International Islamic University Chittagong
Department of Computer Science & Engineering
Spring - 2021

Course Code: CSE-2321
Course Title: Data Structures

Mohammed Shamsul Alam
Professor, Dept. of CSE, IIUC

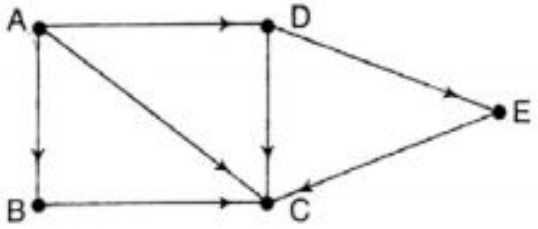
Lecture – 17

Graphs-2

Adjacency List

Let G be a directed graph with m nodes. The sequential representation of G in memory—i.e., the representation of G by its adjacency matrix A—has a number of major drawbacks. First of all, it may be difficult to insert and delete nodes in G. This is 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. Furthermore, if the number of edges is $O(m)$ or $O(m \log_2 m)$, then the matrix A will be sparse (will contain many zeros); hence a great deal of space will be wasted. Accordingly, G is usually represented in memory by a linked *representation*, also called an *adjacency structure*, which is described in this section.

Consider the graph G in Fig. 8.7(a). The table in Fig. 8.7(b) shows each node in G followed by its *adjacency list*, which is its list of adjacent nodes, also called its *successors* or *neighbors*. Figure 8.8 shows a schematic diagram of a linked representation of G in memory. Specifically, the linked representation will contain two lists (or files), a node list NODE and an edge list EDGE, as follows.



(a) Graph G

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

(b) Adjacency list of G

Fig. 8.7

Linked Representation of Graphs

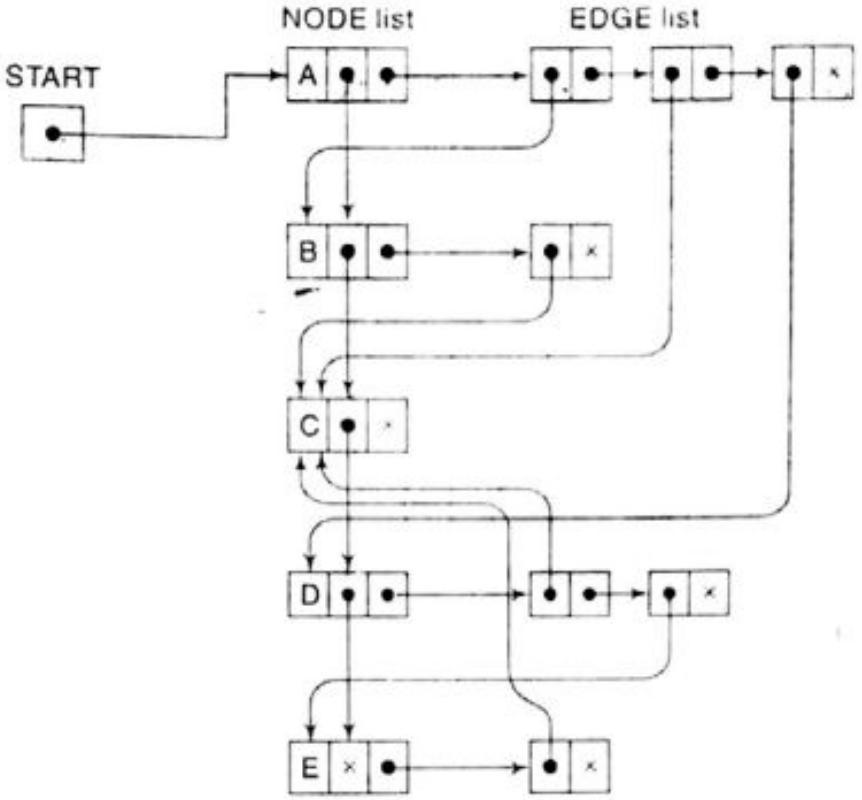


Fig. 8.8

(a) *Node list.* Each element in the list NODE will correspond to a node in G, and it will be a record of the form:

NODE	NEXT	ADJ	

Here 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

Linked Representation of Graphs

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

DEST	LINK	

The field DEST will point to the location in the list NODE of the destination or terminal node of the edge. The field LINK will link together the edges with the same initial node, that is, the nodes in the same adjacency list. The shaded area indicates that there may be other information in the record corresponding to the edge, such as a field EDGE containing the labeled data of the edge when G is a labeled graph, a field WEIGHT containing the weight of the edge when G is a weighted graph, and so on. We also need a pointer variable AVAILE for the list of available space in the list EDGE.

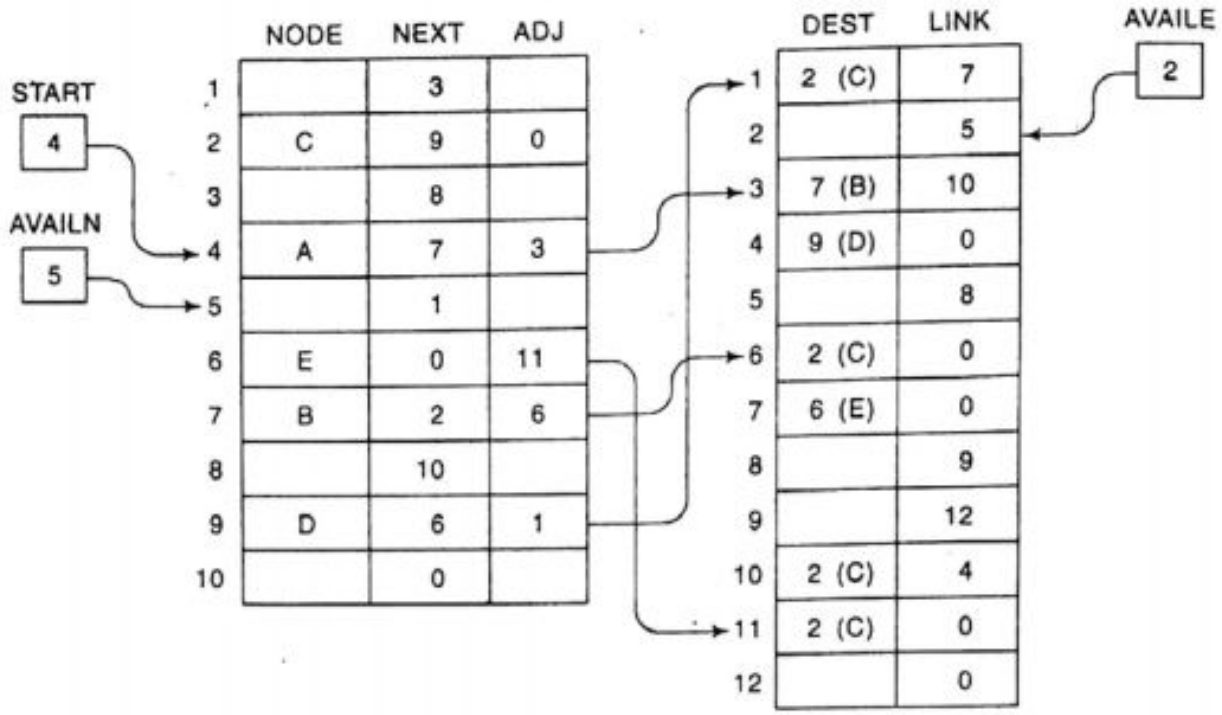


Fig. 8.9

Traversing A Graph

8.7 TRAVERSING A GRAPH

Many graph algorithms require one to systematically examine the nodes and edges of a graph G. There are two standard ways that this is done. One way is called a breadth-first search, and the other is called a depth-first search. The breadth-first search will use a queue as an auxiliary structure to hold nodes for future processing, and analogously, the depth-first search will 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:

STATUS = 1: (Ready state.) The initial state of the node N.
STATUS = 2: (Waiting state.) The node N is on the queue or stack, waiting to be processed.
STATUS = 3: (Processed state.) The node N has been processed.

We now discuss the two searches separately.

Traversing A

Breadth-First Search

The general idea behind a breadth-first search beginning at a starting node A is as follows. 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. Naturally, we need to keep track of the neighbors of a node, and we need to guarantee that no node is processed more than once. This is accomplished by using a queue to hold nodes that are waiting to be processed, and by using a field STATUS which tells us the current status of any node. The algorithm follows.

Algorithm A: This algorithm executes a breadth-first search on a graph G beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1).
2. Put the starting 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 the status of N to the processed state (STATUS = 3).
5. Add to the rear of QUEUE all the neighbors of N that are in the steady state (STATUS = 1), and change their status to the waiting state (STATUS = 2).

[End of Step 3 loop.]

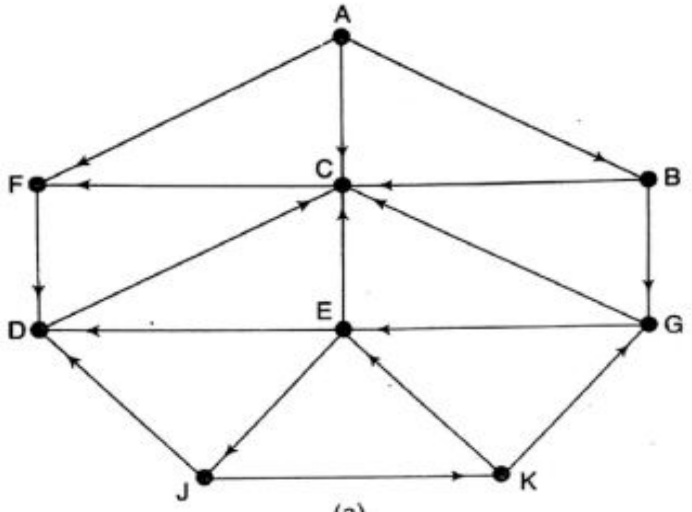
6. Exit.

The above algorithm will process only those nodes which are reachable from the starting node A. Suppose one wants to examine all the nodes in the graph G. Then the algorithm must be modified so that it begins again with another node (which we will call B) that is still in the ready state. This node B can be obtained by traversing the list of nodes.

Traversing A Graph

Example 8.7

Consider the graph G in Fig. 8.14(*a*). (The adjacency lists of the nodes appear in Fig. 8.14(*b*).) Suppose G represents the daily flights between cities of some airline, and suppose we want to fly from city A to city J with the minimum number of stops. In other words, we want the minimum path P from A to J (where each edge has length 1).



(a)

Adjacency lists	
A:	F, C, B
B:	G, C
C:	F
D:	C
E:	D, C, J
F:	D
G:	C, E
J:	D, K
K:	E, G

(b)

Fig. 8.14

(a) Initially, add A to QUEUE and add NULL to ORIG as follows:

FRONT = 1 QUEUE: A
REAR = 1 ORIG : Ø

(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 ORIG : Ø, A, A, A

Note that the origin A of each of the three edges is added to ORIG.

(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 ORIG : Ø, A, A, A, F

Traversing A Graph

(d) Remove the front element C from QUEUE, and add to QUEUE the neighbors of C (which are in the ready state) as follows:

```
FRONT = 4    QUEUE: A, F, C, B, D
REAR  = 5    ORIG  : Ø, A, A, A, F
```

Note that the neighbor F of C is not added to QUEUE, since F is not in the ready state (because F has already been added to QUEUE).

(e) Remove the front element B from QUEUE, and add to QUEUE the neighbors of B (the ones in the ready state) as follows:

```
FRONT = 5    QUEUE: A, F, C, B, D, G
REAR  = 6    ORIG  : Ø, A, A, A, F, B
```

Note that 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, and add to QUEUE the neighbors of D (the ones in the ready state) as follows:

```
FRONT = 6    QUEUE: A, F, C, B, D, G
REAR  = 6    ORIG  : Ø, A, A, A, F, B
```

(g) Remove the front element G from QUEUE and add to QUEUE the neighbors of G (the ones in the ready state) as follows:

```
FRONT = 7    QUEUE: A, F, C, B, D, G, E
REAR  = 7    ORIG  : Ø, A, A, A, F, B, G
```

(h) Remove the front element E from QUEUE and add to QUEUE the neighbors of E (the ones in the ready state) as follows:

```
FRONT = 8    QUEUE: A, F, C, B, D, G, E, J
REAR  = 8    ORIG  : Ø, A, A, A, F, B, G, E
```

We stop as soon as J is added to QUEUE, since J is our final destination. We now backtrack from J, using the array ORIG to find the path *P*. Thus

$$J \leftarrow E \leftarrow G \leftarrow B \leftarrow A$$

is the required path *P*.

Traversing A Graph

Depth-First Search

The general idea behind a depth-first search beginning at a starting node A is as follows. 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. (This algorithm is similar to the inorder traversal of a binary tree, and the algorithm is also similar to the way one might travel through a maze.) The algorithm is very similar to the breadth-first search except now we use a stack instead of the queue. Again, a field STATUS is used to tell us the current status of a node. The algorithm follows.

Algorithm B: This algorithm executes a depth-first search on a graph G beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1).
2. Push the starting 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).

   [End of Step 3 loop.]
6. Exit.

Again, the above algorithm will process only those nodes which are reachable from the starting node A. Suppose one wants to examine all the nodes in G. Then the algorithm must be modified so that it begins again with another node which we will call B—that is still in the ready state. This node B can be obtained by traversing the list of nodes.

Traversing A Graph

Example 8.8

Consider the graph *G* in Fig. 8.14(a). Suppose we want to find and print all the nodes reachable from the node J (including J itself). One way to do this is to use a depth-first search of *G* starting at the node J. The steps of our search follow.

(a) Initially, push J onto the stack as follows:

STACK: J

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

Print J STACK: D, K

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

Print K STACK: D, E, G

(d) Pop and print the top element G, and then push onto the stack all the neighbors of G (those in the ready state) as follows:

Print G STACK: D, E, C

Note that only C is pushed onto the stack, since the other neighbor, E, is not in the ready state (because E has already been pushed onto the stack).

(e) Pop and print the top element C, and then push onto the stack all the neighbors of C (those in the ready state) as follows:

Print C STACK: D, E, F

Traversing A Graph

(f) Pop and print the top element F, and then push onto the stack all the neighbors of F (those in the ready state) as follows:

Print F STACK: D, E

Note that the only neighbor D of F is not pushed onto the stack, since D is not in the ready state (because D has already been pushed onto the stack).

(g) Pop and print the top element E, and push onto the stack all the neighbors of E (those in the ready state) as follows:

Print E STACK: D

(Note that none of the three neighbors of E is in the ready state.)

(h) Pop and print the top element D, and push onto the stack all the neighbors of D (those in the ready state) as follows:

Print D STACK:

The stack is now empty, so the depth-first search of *G* starting at J is now complete. Accordingly, the nodes which were printed,

J, K, G, C, F, E, D

are precisely the nodes which are reachable from J.

Differences between BFS and DFS

Basis for comparison	BFS	DFS
Definition	BFS stands for Breadth First Search	DFS stands for Depth First Search.
Basic	Vertex-based algorithm	Edge-based algorithm
Data structure used to store the nodes	Queue	Stack
Source	BFS is better when target is closer to Source.	DFS is better when target is far from source.
Suitability for decision tree	BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop.
Memory consumption	More memory BFS uses a larger amount of memory because it expands all children of a vertex and keeps them in memory. It stores the pointers to a level's child nodes while searching each level to remember where it should go when it reaches a leaf node.	Less memory DFS has much lower memory requirements than BFS because it iteratively expands only one child until it cannot proceed anymore and backtracks thereafter. It has to remember a single path with unexplored nodes.
Structure of the constructed tree	Wide and short	Narrow and long

Differences between BFS and DFS

Basis for comparison	BFS	DFS
Traversing fashion	Oldest unvisited vertices are explored at first.	Vertices along the edge are explored in the beginning.
Optimality	Optimal for finding the shortest distance, not in cost.	Not optimal
Application	Examines bipartite graph, connected component and shortest path present in a graph.	Examines two-edge connected graph, strongly connected graph, acyclic graph and topological order.
Time Complexity	The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.	The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.