Sanjukta M Dhanty

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.

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).

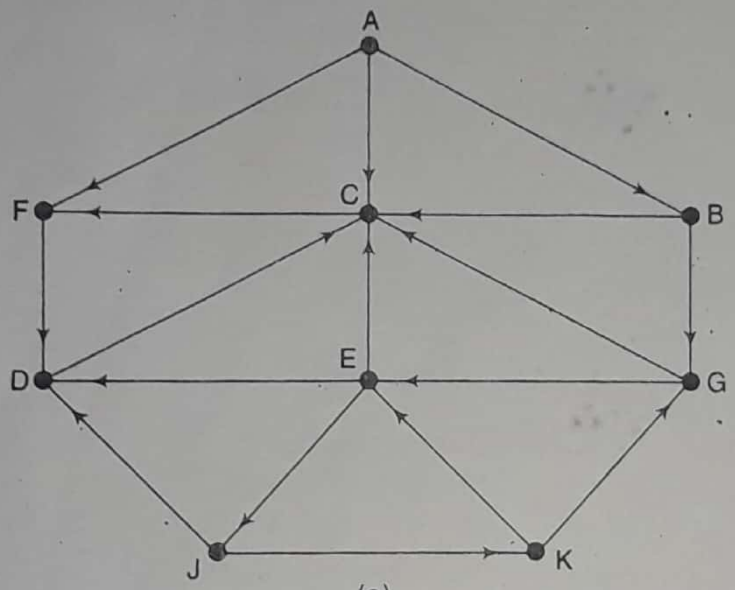
[End of Step 3 loop.]

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.

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

The minimum path P can be found by using a breadth-first search beginning at city A and ending when J is encountered. During the execution of the search, we will also keep track of the origin of each edge by using an array ORIG together with the array QUEUE. The steps of our search follow.

(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

(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*.

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.

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

(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.