

CSC 378 Lecture 11

July 26, 2001

For another reference, see chapter 23 of the textbook (CLR).

1 Graphs

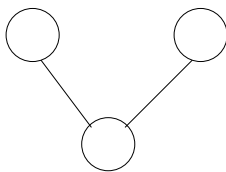
A graph $G = (V, E)$ consists of a set of *vertices* (or *nodes*) V and a set of *edges* E . In general, we let $n = |V|$, the number of nodes, and $m = |E|$, the number of edges. In a *directed* graph, each edge is an ordered pair of nodes (u, v) (so (u, v) is considered different from (v, u)); also, self-loops (edges of the form (u, u)) are allowed. In an *undirected* graph, each edge is a set of two vertices $\{u, v\}$ (so $\{u, v\}$ and $\{v, u\}$ are the same), and self-loops are disallowed. In a *weighted* graph each edge $e \in E$ is assigned a real number $w(e)$ called its *weight*.

An undirected graph is said to be **connected** if there is a path between every two vertices. One way to test whether there is a path between any two vertices u and v is to use a Union-Find ADT:

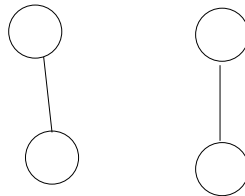
```
For all v in V do
  MAKE-SET(v)
For all (u,v) in E do
  UNION(u,v)
```

Now we can test whether there is a path between u and v by testing $\text{FIND-SET}(u) = \text{FIND-SET}(v)$.

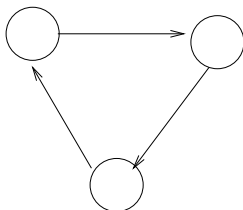
A directed graph is said to be **strongly connected** if, for any two vertices u, v , there is a directed path from u to v . Notice the Union-Find idea does not work for this case.



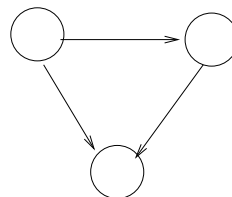
Connected



Not connected



Strongly connected



Not strongly connected

Some standard operations on graphs are:

- Add a vertex; Remove a vertex; Add an edge; Remove an edge.
- Edge Query: given two vertices u, v , find out if the edge (u, v) (if the graph is directed) or the edge $\{u, v\}$ (if it is undirected) is in the graph.
- Neighborhood: given a vertex u in an undirected graph, get the set of vertices v such that $\{u, v\}$ is an edge.
- In-neighborhood, out-neighborhood: given a vertex u in a directed graph, get the set of vertices v such that (u, v) (or (v, u) , respectively) is an edge.
- Degree, in-degree, out-degree: compute the size of the neighborhood, in-neighborhood, or out-neighborhood, respectively.
- Traversal: visit each vertex of a graph to perform some task.

1.1 Data structures for graphs

There are two standard data structures used to store graphs: adjacency matrices, and adjacency lists.

- For an adjacency matrix, let $V = \{v_1, v_2, \dots, v_n\}$. Then, we store information about the edges of the graph in an $n \times n$ array A where

$$A[i, j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

For undirected graphs, the matrix will be symmetric ($A[i, j]$ and $A[j, i]$ will always hold the same value). This requires space $\Theta(n^2)$ but edge queries are $\Theta(1)$. If the graph is weighted, we let $A[i, j]$ store the weight of the edge (v_i, v_j) if that edge exists, and either 0 or ∞ if the edge doesn't exist, depending on the application.

This representation requires space $\Theta(n^2)$ but edge queries are $\Theta(1)$.

- For an adjacency list, we have a 1-dimensional array A of size n . At entry $A[i]$, we store a linked-list of neighbors of v_i (if the graph is directed, we store only the out-neighbors).

The amount of storage required is $\Theta(n + m)$ since each edge (v_i, v_j) of the graph is represented by exactly one linked-list node in the directed case (namely, the node storing v_j in the linked list at $A[i]$), and by exactly two linked-list nodes in the undirected case (node v_j in the list at $A[i]$ and node v_i in the list at $A[j]$). Edge queries can be made $\Theta(\log n)$ (actually, $\Theta(\log(\text{maximum degree}))$) if the lists are stored as balanced trees.

We now examine two ways to traverse a graph:

2 Breadth-First Search (BFS)

BFS takes a graph given as an adjacency list. Starting from a specified source vertex $s \in V$, BFS visits every vertex $v \in V$ that can be reached from s , and keeps track of the path from s to v

with the smallest number of edges. BFS works on directed or undirected graphs: we describe it for directed graphs.

To keep track of progress, each vertex is given a color, which is initially white. The first time that a vertex is encountered, its color is changed to gray. When we finish with a vertex, its color is changed to black. At the same time, for each vertex v , we also keep track of the predecessor of v in the BFS tree, $p[v]$, and we keep track of the number of edges from s to v , $d[v]$.

In order to work in a “breadth-first” manner, BFS uses a first-in, first-out (FIFO) queue Q with operations ENQUEUE(Q , v), DEQUEUE(Q) and ISEMPTY(Q).

```

BFS( $G=(V,E),s$ )
  for all vertices  $v$  in  $V$ 
    color[ $v$ ] := white
    d[ $v$ ] := infinity;
    p[ $v$ ] := NIL;
  end for
  initialize an empty queue  $Q$ ;
  color[ $s$ ] := gray;
  d[ $s$ ] := 0;
  p[ $s$ ] := NIL;
  ENQUEUE( $Q,s$ );
  while not ISEMPTY( $Q$ ) do
    u := DEQUEUE( $Q$ );
    for each edge ( $u,v$ ) in  $E$  do
      if (color[ $v$ ] == white) then
        color[ $v$ ] := gray;
        d[ $v$ ] := d[ $u$ ] + 1;
        p[ $v$ ] := u;
        ENQUEUE( $Q,v$ );
      end if
    end for
    color[ $u$ ] := black;
  end while
END BFS

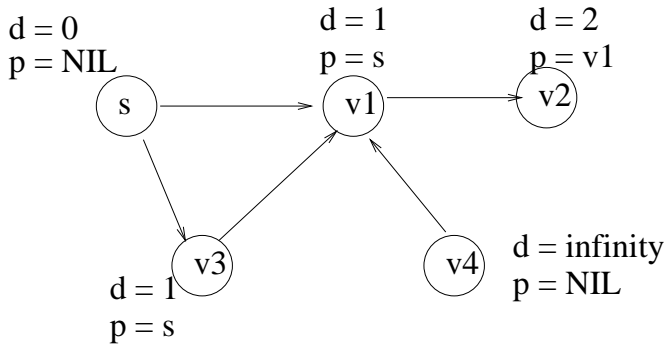
```

Each node is ENQUEUEed at most once, since a node is ENQUEUEed only when it is white, and its color is changed the first time it is ENQUEUEed. In particular, this means that the adjacency list of each node is examined at most once, so that the total running time of BFS is $O(n + m)$, linear in the size of the adjacency list.

Notice that BFS will visit only those vertices that are reachable from s . If the graph is connected (in the undirected case) or strongly-connected (in the directed case), then this will be all the vertices. If not, then we may have to call BFS on more than one start vertex in order to see the whole graph.

For a proof that $d[v]$ really does represent the length of the shortest path (in terms of number of edges) from s to v , consult the text.

Below is a graph showing possible values for d and p . Note that BFS might assign these values slightly differently depending on the order in which the neighbors of each vertex are listed in the adjacency list.



3 Depth-First Search

Just like for BFS, each vertex will be colored white (when it hasn't been "discovered" yet), gray (when it's been encountered but its adjacency list hasn't been completely visited yet), or black (when its adjacency list has been completely visited). The philosophy of DFS is "go as far as possible before backtracking", so we will also keep track of two "timestamps" for each vertex: $d[v]$ will indicate the discovery time (when the vertex was first encountered) and $f[v]$ will indicate the finish time (when it's been completely visited).

In order to implement the depth-first strategy, gray vertices will be stored in a stack instead of a queue. This makes it very natural to write DFS using a recursive algorithm.

```

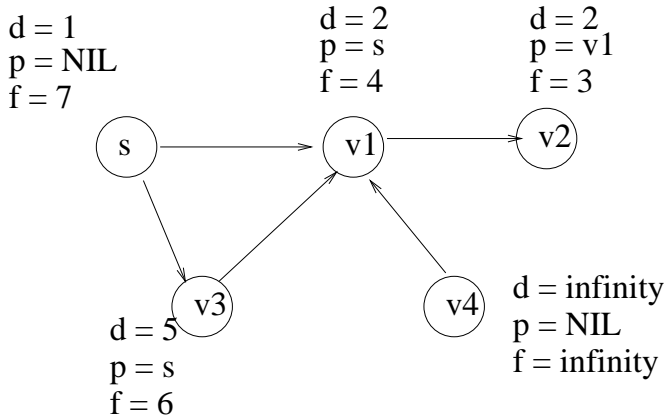
DFS(G=(V,E),s)
  for each vertex v in V
    color[v] := white;
    d[v] := infinity;
    f[v] := infinity;
    p[v] := NIL;
  end for
  time := 0; /* global */
  DFS-VISIT(G,s);
END DFS

DFS-VISIT(G=(V,E),u)
  color[u] := gray;
  time := time + 1;
  d[u] := time;
  for each edge (u,v) in E
    if (color[v] == white) then
      p[v] := u;
      DFS-VISIT(G,v);
    (*) end if
  end for
  color[u] := black;
  time := time + 1;
  f[u] := time;
END DFS-VISIT

```

As for BFS, since `DFS-VISIT` is called on a vertex only when it is white, and vertices become gray the first time they are visited, `DFS-VISIT` is called at most once on each vertex. Also, for each vertex, we visit its adjacency list at most once, so the total running time is just like for BFS, $\Theta(n + m)$ (linear in the size of the adjacency list). As with BFS, DFS will visit only those vertices that are reachable from s .

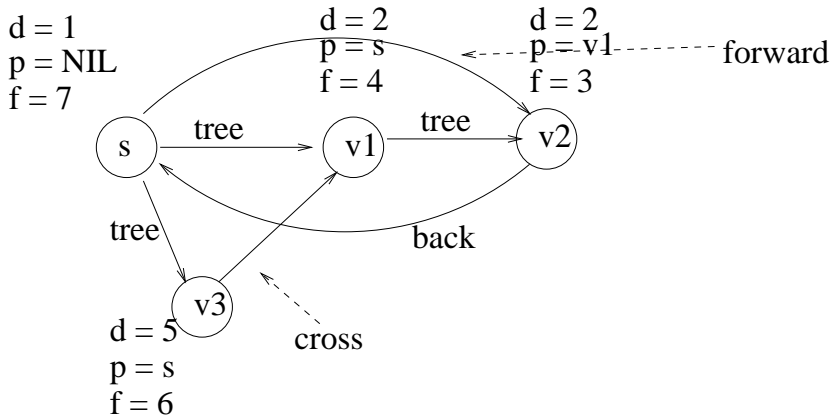
Below is a graph showing possible values for d , p and f . Again, DFS might assign these values slightly differently depending on the order of the adjacency lists.



Note that DFS constructs a "DFS-tree" for the graph, by keeping track of a predecessor $p[v]$ for each node v . For certain applications, we need to distinguish between different types of edges in E :

- Tree Edges are the edges in the DFS tree.
- Back Edges are edges from a vertex u to an ancestor of u in the DFS tree.
- Forward Edges are edges from a vertex u to a descendent of u in the DFS tree.
- Cross Edges are all the other edges that are not part of the DFS tree (from a vertex u to another vertex v that is neither an ancestor nor a descendent of u in the DFS tree).

The following diagram gives one possible output for DFS and labels the types of edges:



One application of DFS is determining whether a graph directed graph G , given as an adjacency matrix, has any cycles in it. A cycle in a directed graph is a directed path from a vertex u to itself. It is not hard to see that there is a cycle in G if and only if there are any back edges when DFS is run. To detect a back edge during the execution of DFS, we can add a test after the line marked by (*) in DFS-VISIT. If the color of v is gray instead of white, then we know that we have seen v before on the current path from the source s . This means that the edge (u, v) is a back edge and therefore forms a cycle.