Elementary Graph Algorithms
Master MOSIG - Algorithms and Program Design

Marc Tchiboukdjian - Denis Trystram

2-10-2009

Course

Objectives:

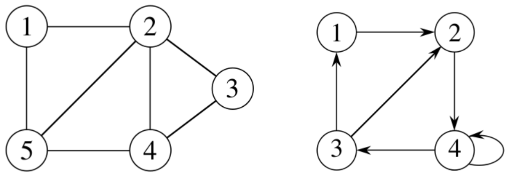
- **Understand adjacency list and adjacency matrix.**
- **Understand breadth first search and depth first search and when to use them.**
- **Analyze globally the cost of an algorithm and not loop by loop.**

<u>**To Remember**</u>

Graph. $G = (V, E)$ with V a set of vertices and E a set of edges. We usually denote the number of vertices by n and the number of edges by m. The degree d_v of a node v is its number of neighbors. The following formula links degrees and number of edges:

$$\sum_{v \in V} d_v = 2m$$

.



(a) Undirected graph (b) Directed graph

Figure 1: Examples of graphs.

Adjacency matrix. A an n by n matrix where $A[i, j] = 1$ iff (i, j) is an edge. Space complexity is $\Theta(n^2)$.

Adjacency list. Adj is an array of n lists. List $Adj[u]$ contains all vertices v s.t. (u, v) is an edge. Space complexity is $\Theta(n + m)$.

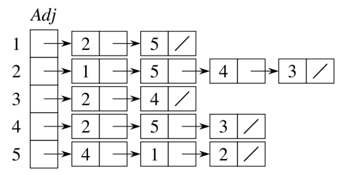
$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

(a) Undirected graph

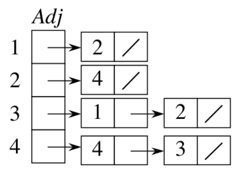
$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 1 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 & 1 \\ \hline \end{array}$$

(b) Directed graph

Figure 2: Adjacency matrices for graphs of fig. 1.



(a) Undirected graph



(b) Directed graph

Figure 3: Adjacency lists for graphs of fig. 1.

Comparison	Winner
Faster to test if (x, y) exists?	matrices
Faster to find vertex degree?	lists
Less memory on small graphs?	lists $(m + n)$ vs. (n^2)
Less memory on big graphs?	matrices (small win)
Edge insertion or deletion?	matrices $O(1)$
Faster to traverse the graph?	lists $m + n$ vs. n^2
Better for most problems?	lists

Adjacency list vs. adjacency matrix. Usually, adjacency list is better and consumes much less memory.

Breadth first search. BFS sends a wave out from a source vertex s. It first hits all vertices 1 edge from s, then hits all vertices 2 edges from s, etc. It uses a FIFO queue Q to maintain the wavefront. $v \in Q$ iff the wave has hit v but has not come out of v yet. BFS outputs $d[v]$, the distance (smallest number of edges) from s to v, for all v and $\pi[v]$, the predecessor of node v in a shortest path from s to v. The complexity is $\Theta(n+m)$. The operations of enqueuing and dequeuing take $\Theta(1)$ time, so the total time devoted to queue operations is $\Theta(n)$. Because the adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(m)$, the total time spent in scanning adjacency lists is $\Theta(m)$. BFS runs in time linear in the size of the adjacency-list representation of G.

```
BFS($G, s$)
    for each vertex $u \in V[G] - \{s\}$
        do $d[u] \leftarrow \infty$
            $\pi[u] \leftarrow$ NIL
    $d[s] \leftarrow 0$
    $\pi[s] \leftarrow$ NIL
    $Q \leftarrow \{s\}$
    while $Q \neq \emptyset$
        do $u \leftarrow head[Q]$
            for each $v \in Adj[u]$
                do if $d[v] = \infty$
                    then $d[v] \leftarrow d[u] + 1$
                         $\pi[v] \leftarrow u$
                         ENQUEUE($Q, v$)
            DEQUEUE($Q$)
```

(a) Pseudo code for BFS

```
DFS($V$, $E$)
for each $u \in V$
    do $color[u] \leftarrow$ WHITE
$time \leftarrow 0$
for each $u \in V$
    do if $color[u]$ = WHITE
        then DFS-VISIT($u$)

DFS-VISIT($u$)
$color[u] \leftarrow$ GRAY            ▷ discover $u$
$time \leftarrow time + 1$
$d[u] \leftarrow time$
for each $v \in Adj[u]$            ▷ explore ($u$, $v$)
    do if $color[v]$ = WHITE
        then DFS-VISIT($v$)
$color[u] \leftarrow$ BLACK
$time \leftarrow time + 1$
$f[u] \leftarrow time$               ▷ finish $u$
```

(b) Pseudo code for DFS

Depth first search. DFS explores deeper in the graph whenever possible. Edges are explored out of the most recently discovered vertex v that still has unexplored edges. When all of v's edges have been explored, backtrack to the vertex from which v was discovered. Every vertex has a color, WHITE if undiscovered, GRAY if discovered but no finished, BLACK if finished. DFS outputs 2 timestamps on each vertex, $d[v]$ the discovery time and $f[v]$ the finishing time. The complexity is $\Theta(n + m)$. Procedure DFS takes time $\Theta(n)$ exclusive of the time to execute the calls to DFS-VISIT. The procedure DFS-VISIT is called exactly once for each vertex v, since DFS-VISIT is invoked only on white vertices and the first thing it does is paint the vertex gray. During an execution of DFS-VISIT(v), the for loop on lines is executed $degree(v)$ times. So the total cost of executing all calls to DFS-VISIT is $\Theta(m)$.

Parenthesis theorem for BFS. For all u, v exactly one of the following holds:

- $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither of u and v is a descendant of the other in the depth-first forest.
- $d[u] < d[v] < f[v] < f[u]$ and v is a descendant of u in a depth-first tree.
- $d[v] < d[u] < f[u] < f[v]$ and u is a descendant of v in a depth-first tree.

$d[u] < d[v] < f[u] < f[v]$ cannot happen.

Classification of edges for BFS.

- **Tree edges**: in the depth-first forest.
- **Back edges**: (u, v) where u is a descendant of v in a depth-first tree.
- **Forward edges**: (u, v) where v is a descendant of u in a depth-first tree but not a tree edge.
- **Cross edges**: any other edge.

Example

Directed Acyclic Graph

A directed acyclic graph (DAG) is a directed graph with no cycles. How to check efficiently if a graph is a DAG, i.e. contains no cycles? A graph is acyclic if a DFS yields no back edges.

Proof. Suppose that there is a back edge (u, v). Then, vertex v is an ancestor of vertex u in the depth-first forest. There is thus a path from v to u in G, and the back edge (u, v) completes a cycle.

Suppose now that G contains a cycle c. We show that a depth-first search of G yields a back edge. Let v be the first vertex to be discovered in c, and let (u, v) be the preceding edge in c. At time $d[v]$, the vertices of c form a path of unexplored vertices from v to u, thus vertex u becomes a descendant of v in the depth-first forest. Therefore, (u, v) is a back edge. □

So we can detect a cycle in a graph in $\Theta(n + m)$.

Topological Sort

A topological sort of a dag G is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.

To perform topological sort, call DFS and output each vertex in order of decreasing finish times.

Proof. To prove the correctness of this algorithm, it suffices to show that for any two distinct vertices u and v, if there is an edge in G from u to v, then $f[v] < f[u]$. Consider any edge (u, v) explored by DFS(G). When this edge is explored, v cannot be gray, since then v would be an ancestor of u and (u, v) would be a back edge (and thus we would have a cycle in an acyclic graph). Therefore, v must be either white or black. If v is white, it becomes a descendant of u, and so $f[v] < f[u]$. If v is black, it has already been finished, so that $f[v]$ has already been set. Because we are still exploring from u, we have yet to assign a timestamp to $f[u]$, and so once we do, we will have $f[v] < f[u]$ as well. Thus, for any edge (u, v) in the dag, we have $f[v] < f[u]$, proving the correctness of the algorithm. □

Exercises

BFS with adjacency matrix representation

What is the running time of BFS if its input graph is represented by an adjacency matrix and the algorithm is modified to handle this form of input?

Iterative version of depth first search

Rewrite DFS with a stack to eliminate recursion.

Rivalry

There are two types of professional wrestlers: "good guys" and "bad guys." Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an $\Theta(n+r)$-time algorithm that determines whether it is possible to designate some of the wrestlers as good guys and the remainder as bad guys such that each rivalry is between a good guy and a bad guy. If is it possible to perform such a designation, your algorithm should produce it.

Another algorithm for topological sort

Another way to perform topological sorting on a directed acyclic graph is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $\Theta(n+m)$. What happens to this algorithm if the graph has cycles?