# CS430

# LECTURE NOTE NO. 7

## GRAPH ALGORITHMS

In this chapter we describe algorithms for graph theoretic problems. We will consider both undirected and directed graphs. For undirected graph a fundamental problem is that of connectivity. Testing a graph for connectedness is of importance in a variety of areas including circuit design, communications etc. However, simple connectivity is not a sufficient criterion in the design of telecommunication networks. It is important that the network be fault tolerant. A graph which provides connectivity after the failure of a node is said to be biconnected. We will see the design of algorithms which test for connectivity and biconnectivity in undirected graphs. Algorithms for deciding whether a graph is $k$-connected, $k \geq 3$ are more complicated and outside the scope of this discussion.

We also consider connectivity in directed graphs. An algorithm will be shown for testing for strong connectivity in graphs.

## 1 Data Structures

Both undirected and directed graphs $G = (V, E)$, where $V = \{v_1, v_2 \ldots v_n\}$ and $E = \{e_1, e_2 \ldots e_m\}$, will be represented by one of the following data structures: For directed graphs each edge $e$ is an ordered pair of vertices.

1. Node-Node Adjacency matrix, $A$: $A$ is a $V$x$V$ matrix where $A(i, j) = 1$ iff the edge $(v_i, v_j)$ exists. For undirected graphs $A(i, j = A(j, i) = 1$ iff the edge $(v_i, v_j)$ exists.

2. Node-Arc Adjacency matrix $B$: $B$ is a $V$x$E$ matrix where

   (i) $B(i, j) = 1$ if $e_j = (v_i, v_k)$ for some $k$.

   (ii) $B(i, j) = -1$ if $e_j = (v_k, v_i)$ for some $k$.

   (iii) $B(i, j) = 0$ if $e_j$ is not incident onto $v_i$.

   For unidirected graphs $B(i, j) = 1$ if $e_j$ is incident onto $v_i$ and is 0 otherwise.

3. Adjacency list $Adj$: $Adj$ is a $V$ size vector where $Adj(i)$ points to a list of edges with one endpoint $v_i$. For directed graphs $Adj(i)$ points to a list of edges of the form $(v_i, v_k)$ for some $k$.

## 2 Breadth First Search

Graph Searching via Breadth-First Search is a useful methodology for determing a path between two vertices. In fact it provides the shortest such path.

Given the graph $G$, BFS starts at a vertex $s$. It uses a FIFO queue to process the vertices.

**Algorithm** $BFS(s)$
**Begin**
$num \leftarrow 0$;
$bfsn(s) \leftarrow 0$;

```
    Add s to FIFO Q;
    WhileQ is non-empty do
        v ← FRONT(Q);
        mark v;
        Forallw ∈ Adj(v) do
        ifw is unmarked then
          bfsn(w) ← bfsn(v) + 1;
          Add w to FIFO Q;
        endif
        endFor
    endWhile
End
```

Breadth first assigns a label/number, termed $bfsn(v)$ to each vertex $v$. This number partitions the graph into levels. Vertex $s$ is at the 0th level. The level number is actually the distance of the shortest path fom $s$ to $v$. This is easy to see by induction. If $v$ is numbered $i$ then $v$ is connected to vertices only at level $i-$ which, by induction, indicates the distance of the shortest-hop path to those vertices.

# 3 Depth First Search

Depth First Search is a recursive methodolgy which allows graph searching in linear time. We first consider depth-first search of undirected graphs.

In this search method we start at a vertex and pick an adjacent untraversed vertex to visit. The graph is searched recursively from this vertex onwards. The vertices are numbered in the order in which they are visited. This number is $dfsn(v)$.

A recursive version of the algorithm is as below. One can also implement the algorithm using a stack.

```
Algorithm DFS(v)
Begin
    mark v; dfsn(v) ← num;
    num ← num + 1;
    ∀w ∈ Adj(v) do
        ifw is unmarked then
          DFS(w);
    finish(v) ← fnum;
    fnum ← fnum + 1;
End
```

This algorithm is called from a main program with an arbitrarily chosen starting point.

```
Algorithm Main(G = (V, E))
Begin
    Pick a vertex root;
    num, fnum ← 1;
    DFS(root);
End
```

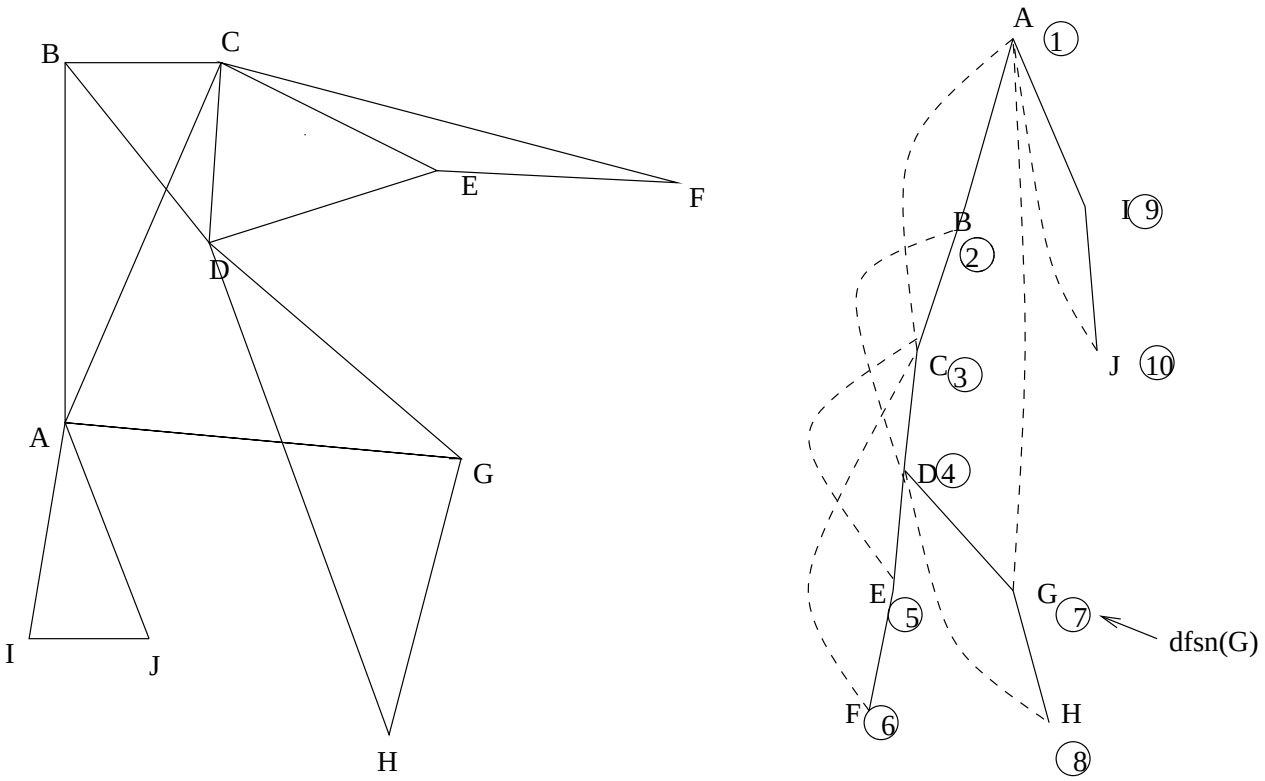


Figure 1: Depth First Search in Undirected Graphs

Depth First Search gives a method of testing for connectivity. If a vertex is left unmarked then the graph is not a connected graph. This is easily proven by contradiction. Suppose the graph is connected but a vertex $w$ is unmarked. There is a path, $P$, from *root* to $w$. W.l.o.g assume that the vertex prior to $w$ on the path is marked. Otherwise $w$ can be chosen to be the first unmarked vertex on $P$. On traversing $w'$, $w$ will be encountered in the adjacency list of $w'$ and will be visited since it is unmarked. This gives us the desired contradiction.

A simple modification to the above algorithm provides us with all the connected components of the graph (left to the reader as an exercise).

For our subsequent discussion we assume that the graph is connected. Depth first search of the undirected graph results in an interesting partition of the edge set. Edges traversed during the visit of the vertices are termed tree edges since these edges form a rooted directed tree. The other edges are termed *back edges*. These edges have the property that they connect vertices which have an ancestor-descendant relationship in the DFS-Tree (figure **??**). All non-tree edges are back edges in the DFS-Tree of undirected graphs. Each back-edge defines a fundamental cycle in the graph. These cycles will be of considerable use subsequently.

In the case of directed graphs however, non-tree edges need not be restricted to back edges only. Edges can be classified as follows (Figure **??**):

1. Forward Edges: Directed edges $(u, v)$ such that $dfsn(u) < dfsn(v)$ and $u$ is an ancestor of $v$ in the DFS-Tree.

2. Back Edges: Directed edges $(u, v)$ such that $dfsn(v) < dfsn(u)$ and $v$ is an ancestor of $u$ in the DFS-Tree.

3. Cross Edges: Directed edges $(u, v)$ such that $dfsn(v) < dfsn(u)$ and $v$ is not an ancestor of $u$.

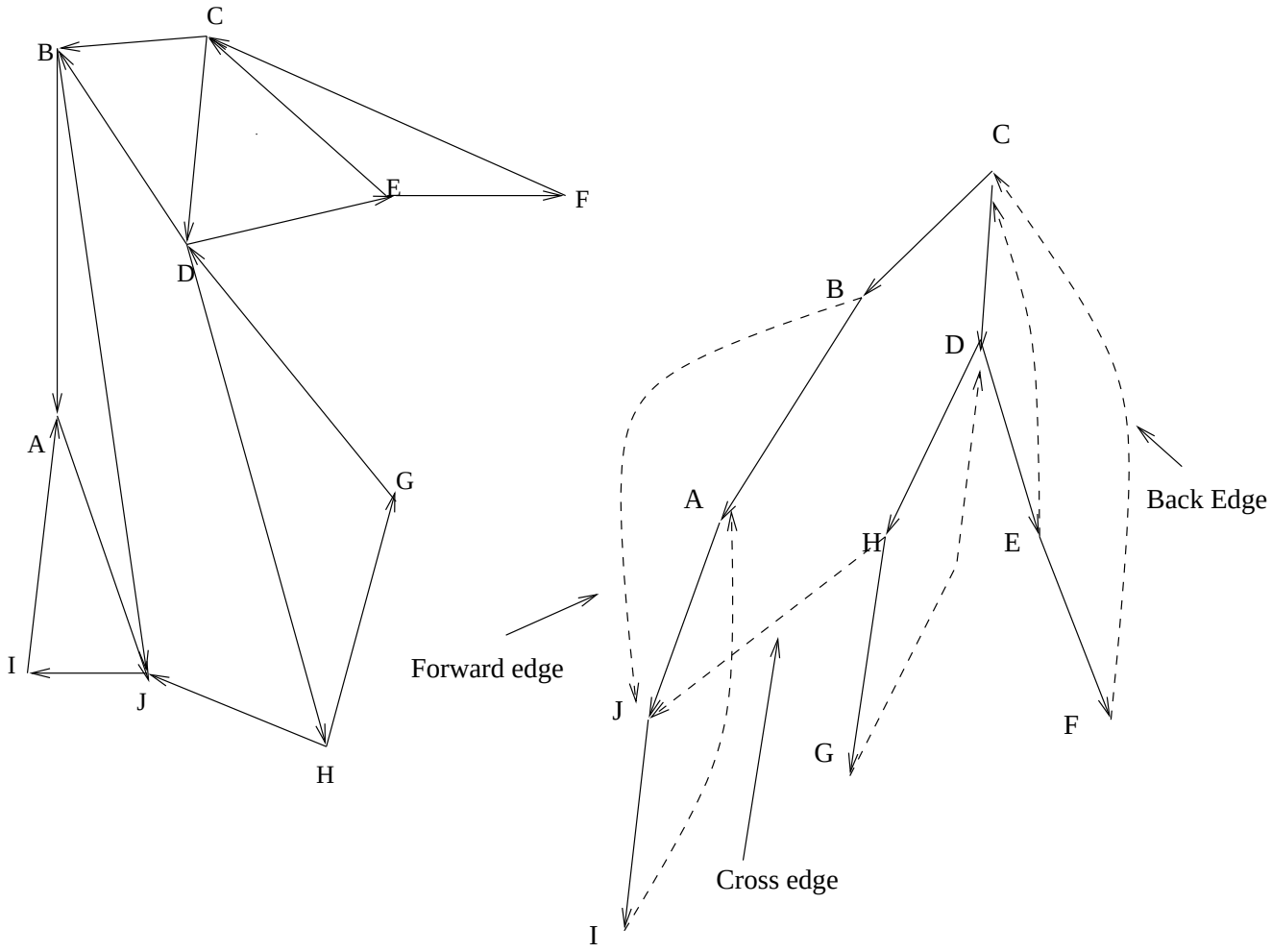


Figure 2: Depth First Search in directed Graphs

In the case of directed graphs also all edges will be of one of the above types. It is easy to see that a directed edge $(u, v)$ where $dfsn(u) < dfsn(v)$ and $u$ is not an ancestor of $v$ does not exist in the DFS.

# 4 Topological Sorting

In an acyclic directed graph, depth first search can be used to topologically sort the graph, i.e. number the vertices $Top(v)$ such that all edges $(u, v)$ satisfy that $Top(u) \leq Top(v)$. We use the number $finish(v)$ alloted in the depth-first algorithm $DFS(v)$.

It is easy to see that arranging the vertices in reverse order of $finish(v)$ provides the topological sort.

# 5 Shortest Paths

Given a directed graph $G(V, \uparrow E)$ with a cost function on the edges $c : E \rightarrow Z$, we define the cost of a path $P = (e_1, e_2 \ldots e_k)$ to be $\Sigma_i c(e_i)$. The following shortest path problems are of interest:

- Single Source Shortest Path (SSSP) Problem: Given a start vertex $s \in V$ find the least cost path from $s$ to all other vertices in the graph.
- All Pairs Shortest Path (APSP) Problem: Find the shortest path between all pairs of vertices in $V$.

We will use dynamic programming principles to solve this problem. The fundamental recurrence for this problem may be expressed as follows:

$$SP(u, v) = min_{w \in Adj(v)}\{SP(w, v) + c(u, w)$$

where $SP(u, v)$ is the shortest path from $u$ to $v$ and $c(u, w)$ the cost of the edge $(u, v)$.

# 6 SSSP Problem

In this problem we will consider two cases; one where all the edge costs are positive and the other where no restriction is imposed on the edge costs. Applying the fundamental recurrence to this problem we get the following useful recurrences:

$$SP(s, v) = min_{w \in Adj(v)}\{SP(s, w) + c(w, v)$$

$$SP(s, v) = min_{w \in Adj(s)}\{c(s, w) + SP(w, v)$$

In order to determine $SP(s, v)$ by dynamic programming we would require $SP(s, w)$. A priori there is no way of ordering the vertices so as to be able to solve the above recurrence. However, the following lemma is of use:

**Lemma 6.1.** *Let $l(w) = c(s, w)$ be a label of vertex $w$ where $(s, w) \in E$. Furthermore $l(w) = \infty$ if no edge $(s, w)$ exists. Then $\exists w$ such that $SP(s, w) = l(w)$.*

*Proof.* Consider a vertex $w'$ and the shortest path to $w'$. This path must use a vertex, say $v$, adjacent to $s$ as the first vertex on the path. Since the shortest path to $w'$ must use the shortest path to $w$ the edge $(s, v)$ is the shortest path to $v$ from $s$. The vertex $v$ is the required vertex $w$. □

**Non-Negative Cost Edges**

We will now consider the case when all the weights in the graph are non-negative.

**Lemma 6.2.** *Let $l(w) = c(s, w)$ be a label of vertex $w$ where $(s, w) \in E$. Furthermore $l(w) = \infty$ if no edge $(s, w)$ exists. Let $w$ be the vertex with the smallest label. Then $SP(s, w) = l(w)$.*

The lemma follows from the fact that the label value asssigned to the vertex with the smallest label will not decrease since all edge costs are non-negative.

This lemma can be extended to the following by an analogous proof:

**Lemma 6.3.** *Let $S$ be a set of vertices to which the shortest path from $s$ has been found. Let $l(v), v \in S$ denote the cost of the shortest path $SP(s, v)$. Let $l(w), w \in V - S$ be the label of vertices such that $l(v)$ is the cost of the shortest path to $v$ from $s$ using vertices in $S$. Let $w \in V - S$ be the vertex with the smallest label. Then $SP(s, w) = l(w)$.*

This leads to the following algorithm developed by Dijkstra:

**Algo ShortP(G,s)**

$S \leftarrow \emptyset$;
$l(s) \leftarrow 0$; $l(v) \leftarrow \infty, v \in V - \{s\}$
Repeat
    begin
        Find vertex $v$ with least label in $V - S$;
        $SP(s, v) \leftarrow l(v)$;
        $S \leftarrow S + \{v\}$;
        /* Update labels of vertices in $V - S$ */
        $l(w) \leftarrow min\{l(w), c(v, w) + SP(s, v)\}$;
    end
Until $|S| = n$;

An implementation of this scheme using Fibonacci heaps gives an $O(E + VlogV)$ algorithm for finding single source shortest paths in graphs where the edge costs are non-negative.

**Arbitrary Edge Costs**

For graphs with arbitrary edge costs Lemma **??** assures us that at least one vertex has $Sp(s, v)$ determined after labels have been computed from the source. Determining this vertex is not an easy task. The algorithm developed by Bellman-Ford does not even attempt to do this. Instead it relies on the fact that vertices $w$ which have $v$ as the adjacent vertex in $SP(w)$ will have their shortest paths determined after $SP(v)$ has been found. Thus at successive steps the number of vertices to which the shortest path has been found increases.

The algorithm is simple to state: Apply the following relaxation step $n$ times:

$$\forall e = (u, v) \in E \quad bfdol(v) \leftarrow min\{l(v), l(u) + c(u, v)\}$$

At some stage in the algorithm the label will be set to the shortest path cost and would not decrease further. The following lemma helps us prove the correctness of the algorithm:

**Lemma 6.4.** *Suppose $SP(u, s)$ has $k$ edges in it. Then $l(u) = SP(u, s)$ in $k$ relaxation steps.*

**Proof:** The proof is by induction. For vertices which are distance 1 from $s$ the label is set correctly at the first step itself. This label will not change at successive relaxation steps. Consider a vertex $w$ whose shortest paths are of edge length $k$. Assuming that the lemma is true for vertices with shortest path edge lengths less than $k$, we note that the shortest path to $w$ uses as its last intermediate vertex a vertex whose shortest path edge length is $k - 1$. The label of this vertex will be set to its shortest path cost at the $k - 1$st relaxation step. Thus at the $k$th relaxation step the label of $w, l(w)$ will be set to $SP(s, w)$.

Consequently the algorithm requires at most $n-1$ relaxation steps provided the graph does not contain a negative cycle. Negative cycle are detected if any label shows a decrease even at the $n$th relaxation step.

# 7 APSP Problem

For the All Pairs problem it is considerably easier to establish a recurrence which can be solved efficiently via dynamic programming.

Suppose we have numbered the vertices $v_1, v_2 \ldots v_n$. This labelling is arbitrary. We define $SP(i,j,k)$ to be the length of the shortest path from $v_i$ to $v_j$ using vertices in the set $\{v_1, v_2 \ldots v_k\}$. Thus $SP(i,j,n)$ gives us the length of the desired shortest path between $v_i$ and $v_j$. The following recurrence characterizes $SP(i,j,k)$ (assuming no self-loops of negative cost):

$$SP(i,j,k) = min_{V_l \in V}\{SP(i,l,k-1) + SP(l,j,k-1)\}$$

$$SP(i,j,0) = c(i,j)$$

$$SP(i,i,k) = 0$$

This recurrence is "convergent" in a simple way.

In fact it is easy to solve the recurrence in $O(n^3)$ steps by maintaining an $O(n^2)$ matrix of shortest path lengths (Details are left to the reader). In this matrix the $(i,j)$th entry will represent $SP(i,j,k)$ for some $k$. Moreover negative cycles are easily detected since a diagonal entry of the matrix will become negative at the discovery of a negative cycle.