Introduction to Graph Theory

- A graph $G = (V,E)$ is a pair of sets:
  - $V$: vertex set.
  - $E$: edge set.
- A graph may be weighted and its edges might be directed.

$V = \{\text{Sea, Sfo, Lax, Msn, Stl, Dfw, Mia, Lga, Pvd}\}$

$E = \{(\text{Sea, Sfo}), (\text{Sfo, Lax}), (\text{Sea, Msn}), \ldots, (\text{Lga, Pvd})\}$
Preliminaries

- Things to know:
  - Path
  - Cycle
  - Sub-graph
  - Degree of a Node
  - Maximum and Minimum Degree
  - Maximum Number of Edges in an Undirected Graph
  - Connected Components of a Graph
  - Shortest Path in a Weighted Graph
  - Tree (rooted tree)
  - Spanning Tree of a Graph:
  - Acyclic Graph
  - Bipartite Graph

Notations:

- Given A graph $G=(V,E)$, where
  - $V$ is its vertex set, $|V|=n$.
  - $E$ is its edge set, with $|E|=m=O(n^2)$.
- If $G$ is connected then for every pair of vertices $u,v$ in $G$ there is path connecting them.
- In an undirected graph an edge $(u,v) = (v,u)$.
- In directed graph $(u,v)$ is different from $(v,u)$.
- In a weighted graph there weights associated with edges or vertices.
- Running time of graph algorithms are usually expressed in terms of $n$ or $m$. 
Graph Representation in terms of Adjacency Matrix

- The adjacency matrix of a graph $G$, denoted by $A_G$ is an $n$ by $n$ defined as follows:

$$A_G[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{if } (i, j) \notin E \end{cases}$$

- If $G$ is undirected then $A_G$ is symmetric.

$$A_G = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Graph Representation in terms of Adjacency List

- In this method for each vertex $v$ in $V$, a list $Adj[v]$ will represent those vertices adjacent to $v$. The size of this list is the degree of $v$.

$$Adj[1] = \{2, 3\}$$
$$Adj[2] = \{3\}$$
$$Adj[3] = \{\}$$
$$Adj[4] = \{3\}$$
Note that:

- Number of 1’s in $A_G$ is $m$.
- Degree of a vertex is the sum of entries in corresponding row of $A_G$
- Sum of all degree is $2m$.
- In a directed graph sum of the out degrees is equal to $m$.

Breadth First Search (BFS)

- Given a graph $G = (V,E)$, BFS starts at some source vertex $s$ and discovers which vertices are reachable from $s$.
- Define the $distance$ between a vertex $v$ and $s$ to be the minimum number of edges on a path from $s$ to $v$. 
BFS (cont’d.)

- BFS discovers vertices in increasing order of distance, and hence can be used as an algorithm for computing shortest paths.

- At any given time there is a *frontier* of vertices that have been discovered, but not yet processed. BFS is so named because it visits vertices across the entire *breadth* of this frontier.

---

BFS (cont’d.)

- We will use the following coloring procedure to show the status of BFS at each instance of time:
  - *Initially all vertices (except the source) are colored white, meaning that they are undiscovered. When a vertex has first been discovered, it is colored gray (and is part of the frontier). When a gray vertex is processed, it becomes black.*
BFS (cont’d.)

- We will also maintain arrays \text{color}[u] which holds the color of vertex \( u \) (either white, gray or black), \text{pred}[u] which points to the predecessor of \( u \) (i.e. the vertex who first discovered \( u \)), and \text{d}[u], the distance from \( s \) to \( u \).

BFS (cont’d.)

- The search makes use of a queue, a first-in-first-out list, where elements are removed in the same order as they are inserted.
- Observe that the predecessor pointers of the BFS search define an inverted tree, with \( s \) as its root. If we reverse these edges we get a tree called a \textit{BFS tree} for \( G \).
These edges of $G$ are called tree edges and the remaining edges of $G$ are called cross edges.

Note that there are many potential BFS trees for a given graph, depending on where the search starts, and in what order vertices are placed on the queue.

BFS Algorithm

```
BFS(G, s) {
    for each $u$ in $V$ // initialization
        { color[$u$] = white; d[$u$] = INFINITY; pred[$u$] = NULL; }
    color[s] = gray; d[s] = 0; // initialize source s
    Q = {s}; // put s in the queue
    while (Q is nonempty) {
        u = Dequeue(Q); // u is the next vertex to visit
        for each v in Adj[u] {
            if (color[v] == white) // if neighbor v undiscovered
                { color[v] = gray; d[v] = d[u] + 1; pred[v] = u;
                    Enqueue(Q, v); // ...put it in the queue
                }
        }
    }
    color[u] = black; // we are done with u
}
```
Example for BFS

Analysis

- Let $n=|V|$ and $m=|E|$. The initialization requires $\Theta(|V|)$ time. Since every vertex will be visited only once, the number of times we go through the while loop is at most $|V|$. The number of iterations through the inner for loop is proportional to $\text{degree}(u) + 1$. Summing up over all vertices we have the running time

$$T(n, m) = n + \sum_{u \in V} (\text{deg}(u) + 1) = n + n + 2m = \Theta(n + m)$$
Depth First Search (DFS)

- Consider the problem of searching a castle for treasure. To solve it you might use the following strategy.
  - As you enter a room of the castle, paint some graffiti on the wall to remind yourself that you were already there.
  - Successively travel from room to room as long as you come to a place you haven't already been.
  - When you return to the same room, try a different door leaving the room (assuming it goes somewhere you haven't already been).
  - When all doors have been tried in a given room, then backtrack.

Notice that this algorithm is described recursively. In particular, when you enter a new room, you are beginning a new search. This is the general idea behind DFS.
DFS Algorithm

- We assume we are given an directed graph $G = (V,E)$. The same algorithm works for undirected graphs. We use four auxiliary arrays.
  - A color for each vertex: white means undiscovered, gray means discovered but not finished processing, and black means finished.
  - We also store predecessor pointers, pointing back to the vertex that discovered a given vertex.

- We will also associate two numbers with each vertex. These are time stamps. When we first discover a vertex $u$ store a counter in $d[u]$ and when we are finished processing a vertex we store a counter in $f[u]$. (Note: not confuse the discovery time $d[v]$ with the distance $d[v]$ from BFS.)
DFS (Cont’d.)

DFS(G) {
    for each u in V {
        color[u] = white;
        pred[u] = nil;
    }
    time = 0;
    for each u in V
        if (color[u] == white)
            DFSVisit(u);
}

DFSVisit(u) {
    color[u] = gray;
    d[u] = ++time;
    for each v in Adj(u) do
        if (color[v] == white) {
            pred[v] = u;
            DFSVisit(v);
        }
    color[u] = black;
    f[u] = ++time;
}

Example

DFS(a) DFS(b) DFS(c)
DFS(d) DFS(e) DFS(f)
DFS(g)

DFS(a) DFS(b) DFS(c)
DFS(d) DFS(e) DFS(f)
DFS(g)

DFS(a) DFS(b) DFS(c)
DFS(d) DFS(e) DFS(f)
DFS(g)

DFS(a) DFS(b) DFS(c)
DFS(d) DFS(e) DFS(f)
DFS(g)
DFS Tree Structure

- DFS imposes a tree structure (actually a collection of trees, or a forest) on the structure of the graph. This is just the recursion tree, where the edge \((u,v)\) arises when processing vertex \(u\) we call DFSVisit\((v)\) for some neighbor \(v\).

DFS Tree Structure

- For directed graphs the other edges of the graph can be classified as follows:
  - **Back edges**: \((u,v)\) where \(v\) is a (not necessarily proper) ancestor of \(u\) in the tree.
  - **Forward edges**: \((u,v)\) where \(v\) is a proper descendent of \(u\) in the tree.
  - **Cross edges**: \((u,v)\) where \(u\) and \(v\) are not ancestors or descendents of one another (in fact, the edge may go between different trees of the forest).
DFS Tree Structure

- For directed graphs the other edges of the graph can be classified as follows:
  - **Back edges**: \((u,v)\) where \(v\) is a (not necessarily proper) ancestor of \(u\) in the tree.
  - **Forward edges**: \((u,v)\) where \(v\) is a proper descendent of \(u\) in the tree.
  - **Cross edges**: \((u,v)\) where \(u\) and \(v\) are not ancestors or descendents of one another (in fact, the edge may go between different trees of the forest).

Parenthesis Structure

- By looking at the time stamps of two nodes we can find out about their ancestor/descendent relation:
Cycles in a Graph

- The time stamps given by DFS allow us to determine a number of things about a graph or digraph. For instance you can determine whether the graph contains any cycles:
  
  - Given a digraph \( G = (V,E) \), consider any DFS forest of \( G \), and consider any edge \((u,v)\) in \( E\):
    - If this edge is a tree, forward, or cross edge, then \( f[u] > f[v] \).
    - If the edge is a back edge then \( f[u] \leq f[v] \).

- Consider a digraph \( G = (V,E) \) and any DFS forest for \( G \). \( G \) has a cycle if and only the DFS forest has a back edge.

Directed Acyclic Graph

- A directed acyclic graph is often called a DAG for short. DAG's arise in many applications where there are precedence or ordering constraints.

- In general a precedence constraint graph is a DAG in which vertices are tasks and the edge \((u,v)\) means that task \( u \) must be completed before task \( v \) begins.
Topological Sort of Directed Acyclic Graphs

- A directed acyclic graph is often called a DAG for short. DAG's arise in many applications where there are precedence or ordering constraints.

- In general a precedence constraint graph is a DAG in which vertices are tasks and the edge \((u,v)\) means that task \(u\) must be completed before task \(v\) begins.

Example

- Initially we only have a huge graph with dependencies:
Topological Sort

- If $G$ is a DAG then we can build a topologic order of its vertices:
  - perform a DFS on $G$.
  - as each vertex is finished, insert it in front of a linked list
  - Return the linked list of the vertices.