**Representing a graph:**

- Graph $G(V, E)$: $V$ - set of nodes (vertices); $E$ - set of edges.
- Notation: $n = |V|$ and $m = |E|$. (Vertices are numbered 1 through $n$.)
- Size of $G$ is $|V| + |E| = n + m$. (Note that $m = O(n^2)$.)
- Degree of a node is the number of edges incident on that node.
- In a directed graph, each node has an *out degree* and an *in degree*.
- For dense graphs, $m = \Theta(n^2)$.
- For sparse graphs, $m = o(n^2)$. (Typically, for a sparse graph, $m = O(n)$.)

**Notes:**

1. $O(|V|)$ and $O(n)$ are used interchangeably; similarly, $O(|E|)$ and $O(m)$ are used interchangeably.
2. Edge between nodes $i$ and $j$ in an undirected graph is denoted by $\{i, j\}$.
3. Edge *from* node $i$ *to* node $j$ in a directed graph is denoted by $(i, j)$.
4. For an undirected graph $G(V, E)$,
   $$\sum_{v \in V} \text{degree}[v] = 2m.$$
5. For a directed graph $G(V, A)$,
   $$\sum_{v \in V} \text{Indegree}[v] = \sum_{v \in V} \text{Outdegree}[v] = m.$$

(a) Adjacency list representation:

- Adj[1 .. $n$]: An array of pointers.
- Adj[$i$] points to a list of nodes; each node stores the number of a node that is adjacent to node $i$. 

**Elementary Graph Algorithms**

**Ref:** Chapter 22 of text. (Omit Section 22.5.)
For a directed graph, the list for each node stores the outgoing edges from that node.
- Edge weights can also be stored in the lists.
- Total number of nodes in all lists = 2m.
- Total storage = $O(n + m)$: linear in the size of the graph.

Example: To be presented in class.

Advantages:
1. Size of representation is linear in the size of graph.
2. Particularly suitable for sparse graphs.

Disadvantage: To check whether an edge $\{i, j\} \in E$, $\Theta(n)$ time is used in the worst case.

(b) Adjacency Matrix representation:
- Uses an $n \times n$ Boolean matrix $M$.
- $M[i, j] = 1$ if $\{i, j\}$ is an edge and 0 otherwise. (By convention, $M[i, i] = 0$, $1 \leq i \leq n$.)

- The adjacency matrix of an undirected graph is symmetric. (The adjacency matrix for a directed graph may not be symmetric.)
- If an edge $\{i, j\}$ has a weight, a non-Boolean matrix can be used instead.
- Total storage = $O(n^2)$: may be quadratic in the size of the graph.

Example: To be presented in class.

Advantages:
1. To check whether an edge $\{i, j\} \in E$, $O(1)$ time is sufficient.
2. Suitable for dense graphs.

Disadvantage: If the graph is sparse, size of representation may be quadratic in the size of graph.
Breadth-First Search (BFS):

- Given a connected graph $G$ and a source node $s$, BFS systematically explores $G$ to discover all the nodes reachable from $s$.
- BFS produces a tree (called BFS tree) $T$ such that the path from $s$ to any node $w$ in $T$ is a shortest path (in terms of number of edges) in $G$ from $s$ to $w$.
- BFS expands the frontier between visited and unvisited nodes along the breadth of the frontier; that is, nodes at a distance $k$ from $s$ are discovered before those at a distance of $k + 1$ or more.

Example: To be presented in class.

Implementation of BFS:

- Adjacency list representation for graph.
- A color scheme to remember whether or not a node has been discovered.
  (a) White: A node that has not yet been discovered. (Initially, all nodes are white.)
  (b) Black: All nodes adjacent to a black node have been discovered (i.e., a black node has no adjacent white node).
  (b) Gray: A gray node may have some undiscovered (white) node adjacent to it.

Values stored for each node $u$:

- $\text{Adj}[u]$: List of adjacent nodes.
- $d[u]$: Distance of $u$ from $s$.
- Initially, $d[u] = \infty$ for all nodes in $V - \{s\}$ ($s$: source node).
- At the end, for each $u \in V$, $d[u]$ gives the shortest path length from $s$ to $u$.
- $\pi[u]$: Parent of $u$ in the BFS tree. ($\pi[u]$ is NULL if $u$ has no parent.)
- $\text{Color}[u]$: Color of node $u$.

Auxiliary data structure: FIFO Queue $Q$
(to enforce the breadth-first nature of the search).

Pseudocode: Handout 15.1.

Example: To be presented in class.
Running time of BFS:
- Step 1: \( O(n) \) time.
- Steps 2 and 3: \( O(1) \) time.
- Step 4: \( O(n + m) \) time. (Explanation in class.)
- Overall running time of BFS = \( O(n + m) \).

BFS and shortest paths: To be discussed in class.

Generating shortest paths from BFS tree:
- See pseudocode for \( \text{PRINT-PATH} \) in Handout 15.1.
- Running time of \( \text{PRINT-PATH} = O(n) \). (Each recursive call shortens the path being considered by 1.)

Depth-First Search (DFS):
- The idea is to search “deeper” in the graph whenever possible.
- The graph generated by DFS is a depth-first spanning tree if the graph \( G \) is connected; otherwise, it is a forest of trees.

Example: To be presented in class.

Implementation of DFS:
- Same color-coding scheme as BFS.
- Global variable ‘time’ is used to generate the values of time stamps.
- For each node \( u \), instead of distance, two time stamp values are stored.
  (a) \( d[u] \): Time when \( u \) is first discovered (i.e., time when the color of \( u \) changes to gray).
  (b) \( f[u] \): Time when the search finishes examining \( u \)’s adjacency list (i.e., time when the color of \( u \) changes to black).

Notes on timestamps:
1. Timestamps are integers in the range \([1 .. 2n]\). (\( 2n \) timestamp values are needed because each node gets ‘discovered’ once and ‘finished’ once.)
2. For each node \( u \), \( d[u] < f[u] \).
3. The color of \( u \) is white before \( d[u] \), gray between \( d[u] \) and \( f[u] \), and black thereafter.
Pseudocode for DFS: Handout 15.1.

Example for DFS: To be presented in class.

Running time of DFS:

- Steps 1 and 2 (Initialization steps): \( O(n) \) time.
- DFS-Visit is called exactly once for each node.
- The call DFS-Visit\((v)\) takes \( O(\text{degree}(v)) \) time.
- So, total time for all calls to DFS-Visit is \( O(\sum_{v \in V} \text{degree}(v)) = O(m). \)
- So, the running time of DFS is \( O(n + m). \)

Classification of edges through DFS

Tree edges: Edges in the DFS tree (or forest).

- DFS on an undirected graph: Edge \( \{u, v\} \) is a tree edge if \( v \) was discovered when \( u \)'s adjacency list was being processed or \( u \) was discovered when \( v \)'s adjacency list was being processed.

- DFS on a directed graph: Edge \((u, v)\) is a tree edge if \( v \) was discovered when \( u \)'s adjacency list was being processed (i.e., while the edge \((u, v)\) was being explored).

Back edges: Such edges are not in the DFS tree (or forest).

- A back edge joins a node \( v \) to an ancestor of \( v \) in the DFS tree.
- Self loops in directed graphs are considered to be back edges.

Forward edges: Such edges are also not in the DFS tree (or forest).

- Forward edges are possible only in directed graphs.
- A forward edge \((u, v)\) joins a node \( u \) to a descendant \( v \) in the DFS tree.
Cross edges: Any edge that is not a tree edge or a back edge or a forward edge.

- Cross edges are possible only in directed graphs.
- A cross edge may join a pair of nodes as long as one is not an ancestor of another in the DFS tree.

Modifying DFS to classify edges:

- For directed graphs only.
- The modification won’t distinguish between forward and cross edges.
- Uses the colors of nodes.

Suppose the directed edge \((u, v)\) is encountered.

(a) If Color\([v]\) is white, then \((u, v)\) is a tree edge.
   (Reason: Vertex \(v\) is just being discovered.)
(b) If Color\([v]\) is gray, then \((u, v)\) is a back edge.
   (Reason: Gray nodes are in the (recursion) stack and \(v\) is deeper in the stack than \(u\).)
(c) If Color\([v]\) is black, then \((u, v)\) is a forward or cross edge.

Example for Case (c): To be presented in class.

Note: In Case (c),

- \((u, v)\) is a forward edge if \(d[u] < d[v]\).
- \((u, v)\) is a cross edge if \(d[u] > d[v]\).

**Theorem 1**: In a DFS of an undirected graph \(G(V, E)\), every edge in \(E\) is either a tree edge or a back edge.

Proof: To be presented in class.

**Applications of DFS**:

1. Finding connected components:
   - For undirected graphs only.
   - Suppose \(G(V, E)\) has \(t\) connected components, numbered 1 through \(t\).
   - Algorithm produces array \(CC[1 .. n]\), where \(CC[u]\) is the number of the connected component containing nodes \(u\).
• **Idea:** Do a DFS. Whenever the recursive calls to DFS-Visit are completed and a new “exploration” is started, a new connected component begins.

**Pseudocode:** Handout 15.2.

**Running time:** $O(m+n)$ (because it involves just a DFS).

2. **Topological sort of a DAG:**

**Note:** DAG – Directed Acyclic Graph.

**Definition:** Given a DAG $G$, a topological sort of $G$ is a linear arrangement of the nodes so that for each directed edge $(u, v)$, $u$ appears before $v$.

• Thus, a topological sort is a listing of the nodes on a line so that each directed edge goes from left to right.

• Such an ordering is not possible if the directed graph contains a cycle.

• Topological sort is used in situations where an order for a set of events needs to be determined given some precedence constraints.

**Examples:** To be discussed in class.

**Pseudocode:** Handout 15.2.

**Running time:** $O(m + n)$.

**Correctness of the algorithm:**

**Observation 2:** When DFS is carried out on a dag, no back edges can arise.

**Reason:** Any back edge indicates the presence of a directed cycle.

**Theorem 3:** When DFS is carried out on a dag, for every directed edge $(u, v)$, $f[u] > f[v]$.

**Proof:** To be presented in class.