Basic Traversal and Search Techniques

Definition 1 ***Traversal*** *of a binary tree involves examining every node in the tree.*

Definition 2 ***Search*** *involves visiting nodes in a graph in a systematic manner, and may or may not result into a visit to all nodes.*

Techniques for binary trees

- Traversal produces a linear order for the information in a tree
- Inorder, preorder, and postorder traversals

  Theorem 1 *Let T_n and S_n represent the time and space needed by any one of the traversal algorithms when the input tree t has $n \geq 0$ nodes. If the time and space needed to visit a single node is $\Theta(1)$, then $T_n = \Theta(n)$ and $S_n = O(n)$.*

- Level-order traversal

Techniques for graphs

- Reachability problem in graph theory
  - Determine whether a vertex v is reachable from a vertex u in a graph $G = (V, E)$.
- Breadth first search and traversal
  - *Explore* all vertices adjacent from a starting vertex
  - Explore unexplored vertices that are adjacent to all the explored vertices

```
node bfs ( node v )
{
    queue q;                        // Initialize an empty queue of nodes
    q.enqueue ( v );
    list  visited;                  // List of visited nodes (initially empty)
    while ( ! q.empty() )
    {
        node u = q.dequeue();
        if ( u is vertex being searched for )
            return ( u );

        for each vertex nu adjacent from u
            if ( ! visited.search ( nu ) )
            {
                q.enqueue ( nu );
                visited.insert ( nu );
            }
    }

    // Did not find a solution

    return ( NULL );
}
```

Theorem 2 *Algorithm* `bfs` *visits all vertices reachable from v.*

- Notice the similarity between breadth-first search and level-order traversal

- Depth first search and traversal
  - More like pre-order traversal

```
node dfs ( node v )
{
    static list visited;          // Global list of visited nodes
    if ( ! visited.search ( v ) )
        visited.insert ( v );     // Add v to the list of visited nodes

    if ( v is vertex being searched for )
        return ( v );

    for each vertex u adjacent from v
    {
        node sol = dfs ( u );
        if ( sol != NULL )
            return ( sol );
    }

    return ( NULL )
}
```

Connected components and spanning trees

- G is a connected graph implies that all vertices will be visited by `bfs`
- If G is not connected, you have to make a call to `bfs` for each of the connected components
- The above property can be used to check if a given graph G is connected
- Use `bfs` to compute a spanning tree in a graph
  - The computed spanning tree is *not* a minimum spanning tree
- The check for connected components as well as the computation of spanning tree can be performed using `dfs` as well
  - The spanning trees given by `bfs` and `dfs` are not identical

Biconnected components and depth first search

- Articulation point

  Definition 3 *A vertex v in a connected graph G is an **articulation point** if the deletion of v from G, along with the deletion of all edges incident to v, disconnects the graph into two or more nonempty components.*

- Biconnected graph

  Definition 4 *A graph G is **biconnected** if and only if it contains no articulation points.*

- Algorithm to determine if a connected graph is biconnected

- Identify all the articulation points in a connected graph
- If graph is not biconnected, determine a set of edges whose inclusion makes the graph biconnected
  * Find the maximal subgraphs of G that are biconnected
  * Biconnected component

    Definition 5 $G' = (V', E')$ *is a* ***maximal biconnected subgraph*** *of* G *if and only if* G *has no biconnected subgraph* $G'' = (V'', E'')$ *such that* $V' \subseteq V''$ *and* $E' \subseteq E''$*. A maximal biconnected subgraph is a* ***biconnected component****.*

Lemma 1 *Two biconnected components can have at most one vertex in common and this vertex is an articulation point.*