Basic Traversal and Search Techniques

Traversal vs Search

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.*

- Different nodes of a graph may be *visited*, possibly more than once, during traversal or search
- If search results into a visit to all the vertices, it is called traversal

Techniques for binary trees

- Possible problem: Find all nodes in a binary tree with data value less than some specified value
  - Solved by systematically examining all the vertices
  - Does searching for a specified item in a binary search tree result into a traversal?
- Traversal produces a linear order for the information in a tree
  - During traversal, treat each node in the binary tree and its subtrees in the same manner
- 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 $S_n = O(n)$.*

  Proof:

  - Each node in the tree is visited three times, requiring constant amount of work
    1. From parent (or start node, if root)
    2. Return from left subtree
    3. Return from right subtree
  - This gives the total time required for traversal to be $\Theta(n)$
  - Additional space is required for recursion stack
    - If t has depth d, this space is given by $\Theta(d)$
    - For an n-node binary tree, $d \leq n$
    - Hence, $S_n = O(n)$
- 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 $G = (V, E)$; or whether there exists a path from u to v
  - A more general form: Given a vertex $u \in V$, find all vertices v_i such that there is a path from u to v_i
- Breadth first search and traversal

- *Explore* all vertices adjacent from a starting vertex
  - A vertex is said to be explored when the algorithm has visited all the vertices adjacent from it
  - As a vertex is reached or visited, it is said to be unexplored
- Explore unexplored vertices that are adjacent to all the explored vertices
- Breadth-first search operates using a queue to maintain the list of unexplored vertices

```
node bfs ( node v )
{
    // q is the queue of unexplored nodes

    queue q;                          // Initialize an empty queue of nodes
    q.enqueue ( v );

    // visited is the list of nodes visited (explored and unexplored)

    list  visited;                    // List of visited nodes (initially empty)
    visited.insert ( v );

    while ( ! q.empty() )             // There are nodes to be explored
    {
        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 );
}
```

- Example: Undirected graph $G = (V, E)$
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
  - $E = \{12, 13, 24, 25, 36, 37, 48, 58, 68, 78\}$
- Example: Directed graph $G = (V, E)$
  - $V = \{1, 2, 3, 4\}$
  - $E = \{12, 23, 41, 43\}$
  - Starting at vertex 1, vertex 4 is not reachable
- Notice the similarity between breadth-first search and level-order traversal

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

Proof Let $G = (V, E)$ be a directed or undirected graph and let $v \in V$. Prove the theorem by induction on the length of the shortest path from v to every reachable vertex $w \in V$. The length of the shortest path from v to w is denoted by $d(v, w)$.

- Base step: $d(v, w) = 1$. Clearly, all the vertices with $d(v, w) \leq 1$ get visited.

- Induction hypothesis: Assume that all vertices with $d(v, w) \leq r$ get visited.
- Induction step: Show that all vertices w with $d(v, w) = r + 1$ also get visited.
  - Let $w \in V$ such that $d(v, w) = r + 1$
  - Let u be a vertex that immediately precedes w on a shortest v to w path
  - Then, $d(v, u) = r$ and u gets visited by `bfs`
  - It is safe to assume that $u \neq v$ and $r \geq 1$
  - Hence, immediately before u gets visited, it is placed on the queue `q` of unexplored vertices
  - The algorithm continues in the `while` loop until `q` becomes empty
  - Hence, u is dequeued from `q` at some time and all unvisited vertices adjacent from it get visited in the inner `for` loop
  - Hence, w gets visited

  Theorem 3 *Let $T(|V|, |E|)$ and $S(|V|, |E|)$ be the maximum time and maximum additional space taken by algorithm* `bfs` *on any graph $G = (V, E)$. If G is represented by its adjacency lists, $T(|V|, |E|) = \Theta(|V| + |E|)$ and $S(V|, |E|) = \Theta(|V|)$. If G is represented by its adjacency matrix, then $T(|V|, |E|) = \Theta(|V|^2)$ and $S(V|, |E|) = \Theta(|V|)$.*

- Depth first search and traversal
  - Exploration of a vertex u is suspended as soon as a new vertex v is reached
    - Start exploring the new vertex v
    - When v is completely explored, continue exploration of u
    - Terminate search when all the vertices are fully explored
  - Perform `dfs` on example from `bfs`
  - 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 )
}
```

  - `dfs` visits all vertices reachable from vertex u
  - If $T(|V|, |E|)$ and $S(|V|, |E|)$ represent the maximum time and maximum additional space taken by `dfs`, then $S(|V|, |E|) = \Theta(|V|)$ and $T(|V|, |E|) = \Theta(|V| + |E|)$ if adjacency lists are used and $T(|V|, |E|) = \Theta(|V|^2)$ if adjacency matrices are used

Connected components and spanning trees

- G is a connected undirected graph implies that all vertices will be visited in the first call to `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
  - All newly visited vertices on a call to `bfs` from `bft` represent the vertices in a connected component of G
    - Connected components of a graph can be found using `bft`
    - Modify `bfs` so that all newly visited vertices are put onto a list
    - Subgraph formed by the vertices on this list make up a connected component
    - If adjacency lists are used, a breadth-first traversal will obtain the connected components in $\Theta(n + e)$ time
- Use `bfs` to compute a spanning tree in a graph
  - A graph G is a spanning tree iff G is connected
  - The computed spanning tree is *not* a minimum spanning tree
  - Breadth-first spanning tree and depth-first 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

- Consider only the undirected graphs
- 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.*
  - Graph $G = (V, E)$
    - $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
    - $E = \{1-2, 1-4, 2-3, 2-5, 2-7, 2-8, 3-4, 3-9, 3-10, 5-6, 5-7, 5-8, 7-8\}$
    - Vertex 2 is an articulation point
    - Vertex 5 and 3 are also articulation points
- Biconnected graph

  Definition 4 *A graph G is **biconnected** if and only if it contains no articulation points.*
  - Presence of an articulation point may be an undesirable feature in many cases
  - Consider a communications network with nodes as communication stations and edges as communication lines
  - Failure of a communication station may result in loss of communication between other stations as well, if graph is not biconnected
- 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.*