

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

- Determine a vertex or a subset of vertices that satisfy a specified property
 - 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

```
struct tree_node_type
{
    item_type      data;    // To hold information at each node
    tree_node_type left_child;
    tree_node_type right_child;
};
```

```
Algorithm inorder ( t )
// Input: t is a binary tree
// Each node of t is of type tree_node_type
{
    if ( t != NULL )
    {
        inorder ( t.left_child );
        visit ( t );    // Perform a function on data in node being traversed
        inorder ( t.right_child );
    }
}
```

```
Algorithm preorder ( t )
// Input: t is a binary tree
// Each node of t is of type tree_node_type
{
    if ( t != NULL )
    {
        visit ( t );    // Perform a function on data in node being traversed
```

```

        preorder ( t.left_child );
        preorder ( t.right_child );
    }
}

Algorithm postorder ( t )
// Input: t is a binary tree
// Each node of t is of type tree_node_type
{
    if ( t != NULL )
    {
        postorder ( t.left_child );
        postorder ( t.right_child );
        visit ( t );    // Perform a function on data in node being traversed
    }
}

```

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 $v_i \in V$ such that there is a path from u to v_i
 - * Solved by starting at vertex u and systematically searching the graph G for vertices reachable from u
- 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 becomes a new unexplored vertex
 - 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

```

Algorithm BFS ( node v )
// A breadth-first search of graph G is carried out starting at vertex v.
// Returns the node if found; else returns NULL
{
    // 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 $G = (V, E)$
 - * $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
 - * $E = \{1-2, 1-3, 2-4, 2-5, 3-6, 3-7, 4-8, 5-8, 6-8, 7-8\}$
- Example: Directed graph $G = (V, E)$
 - * $V = \{1, 2, 3, 4\}$
 - * $E = \{1-2, 2-3, 4-1, 4-3\}$
 - * 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 $S(|V|, |E|) = \Theta(|V|)$.

Proof: Vertices get added to the queue only in the statement `q.enqueue (nu)`;

A vertex gets added to the queue only if it has not been previously visited

As the vertex is added to the queue, it is also added to the list of visited vertices

Hence, each vertex can be added to the queue at most once

Since each vertex gets added to the queue once, at most $|V|$ additions are made, giving a queue space need of $O(n)$

Remaining variables take constant space

Hence, $S(|V|, |E|) = O(|V|)$

If G is an n -vertex graph with the starting vertex v connected to the remaining $n - 1$ vertices, then all $n - 1$ vertices are on the queue at the same time

We need $\Theta(|V|)$ space for the `visited` list

Hence $S(|V|, |E|) = \Theta(|V|)$

The result is independent of whether adjacency matrices or lists are used

- 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 ) )
        return ( NULL );

    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

- BFS vs DFS

- BFS fully explores a node before the exploration of other nodes
 - * The next node to explore is the first unexplored remaining node
- DFS suspends exploration of the node as soon as another unexplored node is encountered
 - * The exploration of the new node is started immediately

Connected components and spanning trees

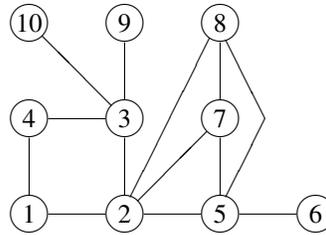
- G is a connected undirected graph implies that all vertices will be visited in the first call to bfs
 - If G is not connected, you have to make a call to bfs for each of the connected components
 - The above property can be used to check if a given graph G is connected
 - All newly visited vertices 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 has a spanning tree iff G is connected
 - * BFS easily determines the existence of a spanning tree
 - The set of edges used in the traversal of the tree are known as *forward edges*
 - * If t is the set of forward edges and G is connected, then t is the spanning tree of G
 - 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** iff 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 $G = (V, E)$



- * Vertex 2 is an articulation point
- * Vertices 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
- If G has no articulation point, the failure of a station i still allows communications between every pair of stations not including station i

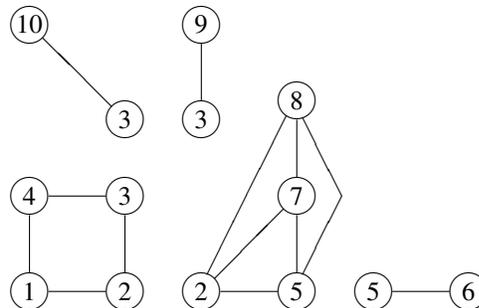
• Algorithm to determine if a connected graph is biconnected

- 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 $V' \subsetneq V''$ and $E' \subsetneq E''$. A maximal biconnected subgraph is a **biconnected component**.

- * Biconnected components of the graph from above



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

- * No edge can be in two different biconnected components; this will require two common vertices
- * Graph G can be transformed into a biconnected graph by using the edge addition scheme in the following algorithm

```

algorithm mk_biconnected
// Input: Graph G = ( V, E )
{
    for each articulation point a
    {

```

Let B_1, B_2, \dots, B_k be biconnected components containing vertex a ;
 Let $v_i, v_{i+1} \neq a$, be a vertex in $B_i, 1 \leq i \leq k$;
 Add to G the edges $(v_i, v_{i+1}), 1 \leq i < k$;

}

}

- Biconnected components of the previous graph: $\{1, 2, 3, 4\}, \{2, 5, 7, 8\}, \{5, 6\}, \{3, 10\}, \{3, 9\}$
 - * Add edge $(4, 10)$ and $(10, 9)$ corresponding to articulation point 3, edge $(1, 5)$ corresponding to articulation point 2, and edge $(6, 7)$ corresponding to articulation point 5
- In the algorithm `mk_biconnected`, once the edge (v_i, v_{i+1}) is added, vertex a is no longer an articulation point
 - * Since we add an edge corresponding to each articulation point, there are no more articulation points and G is biconnected
- If G has p articulation points and b biconnected components, the above algorithm introduces exactly $b - p$ new edges into G

• Identifying articulation points and biconnected components of a connected graph

- Graph G with $n \geq 2$ vertices
- Depth-first spanning tree of G starting at node 1
 - * Identify the order in which each node is visited by a number
 - * Start with number 1 for node 1, number 2 for node 2, number 3 for node 5, and so on
 - * The number is known as *depth first number* (*dfn*)
 - * The edges used in depth first tree are called *tree edges*; other edges are called *back edges*
 - * (u, v) is a *cross edge* relative to t iff u is not an ancestor of v and v is not an ancestor of u

Lemma 2 *If (u, v) is any edge in G , then relative to the depth first spanning tree t , either u is an ancestor of v or v is an ancestor of u . So, there are no cross edges relative to a depth first spanning tree.*

Proof: Assume that $(u, v) \in E(G)$ is a cross edge

- (u, v) cannot be a tree edge as otherwise u is a parent (or child) of v
- So, (u, v) must be a back edge
- We can assume that $dfn[u] < dfn[v]$
- Since u is visited first, its exploration cannot be complete unless v is visited
- From the definition of DFS, it follows that u is an ancestor of all the vertices visited until u is completely explored
- Hence u is an ancestor of v in t and (u, v) cannot be a cross edge QED

Lemma 3 *The root node of a depth first spanning tree is an articulation point iff it has at least two children. Furthermore, if u is any other vertex, then it is not an articulation point iff from every child w of u it is possible to reach an ancestor of u using only a path made up of descendants of w and a back edge.*

- If it cannot be done for some child w of u , the deletion of vertex u leaves behind at least two nonempty components
 - one containing the root and the other containing vertex w
- Identifying articulation points using the above observation
 - * For each vertex u , define $L[u]$ as

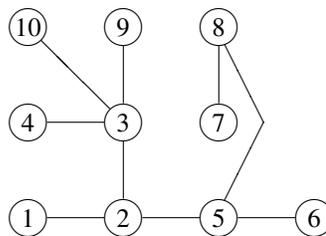
$$L[u] = \min \left\{ \begin{array}{l} dfn[u], \\ \min \left\{ \begin{array}{l} L[w] \quad | \quad w \text{ is a child of } u, \\ \min\{dfn[w] \quad | \quad (u, w) \text{ is a back edge} \} \end{array} \right\} \end{array} \right\}$$

- * $L[u]$ is the lowest *dfn* reachable from u using a path of descendants followed by at most one back edge

- * If u is not the root, then u is an articulation point iff u has a child w such that $L[w] \geq dfn[u]$

– Example:

- * Spanning tree



- * $dfn[1 : 10] = \{1, 2, 7, 10, 3, 4, 6, 5, 8, 9\}$
- * $L[1 : 10] = \{1, 2, 1, 1, 2, 4, 6, 2, 8, 9\}$
- * 2 is an articulation point because $dfn[2] = 2$ and $L[5] = 2$
- * 5 is an articulation point because $dfn[5] = 3$ and $L[6] = 4$
- * 8 is an articulation point because $dfn[8] = 5$ and $L[7] = 6$
- * 3 is an articulation point because $dfn[3] = 7$ and $L[9] = 8$

– $L[u]$ can be easily computed by visiting the vertices of depth first spanning tree in postorder

– Algorithm Articulate to perform a depth first search of graph G and visit the nodes in resulting depth first spanning tree in postorder

```

Algorithm Articulate ( u, v )
// u is a starting vertex for depth first search
// v is parent of u, if any in the depth first spanning tree
// Global array dfn is initialized to 0
// Global variable num is initialized to 1
// n is the number of vertices in G
// Initial call is: Articulate ( 1, 0 )
{
    dfn[u] = num;           // Assign dfn to newly visited node
    L[u] = num;
    num = num + 1;
    for each vertex w adjacent from u // Compute L[i] for unvisited vertices
    {
        if ( dfn[w] = 0 ) // w is unvisited
        {
            Articulate ( w, u );
            L[u] = min ( L[u], L[w] );
        }
        else
            if ( w != v )
                L[u] = min ( L[u], dfn[w] );
    }
}

```

– After computing $L[1 : n]$, articulation points can be identified in $O(e)$ time

– Complexity of Articulate is $O(n + e)$, allowing determination of articulation points in G in $O(n + e)$ time

– Finding biconnected components of G

- * If at any point in Articulate, we find that $L[w] \geq dfn[u]$, u is either a root or an articulation point

- * Modify Articulate to find biconnected components

```

Algorithm bi_comp ( u, v )
// u is a starting vertex for depth first search

```

```

// v is parent of u, if any in the depth first spanning tree
// Global array dfn is initialized to 0
// Global variable num is initialized to 1
// n is the number of vertices in G
// Initial call is: bi_comp ( 1, 0 )
{
    dfn[u] = num;           // Assign dfn to newly visited node
    L[u] = num;
    num = num + 1;
    for each vertex w adjacent from u // Compute L[i] for unvisited vertices
    {
        if ( ( v != w ) && ( dfn[w] < dfn[u] ) )
            push edge ( u, w ) to the top of stack s;

        if ( dfn[w] = 0 ) // w is unvisited
        {
            if ( L[w] >= dfn[u] )
            {
                write ( "New bicomponent" );
                repeat
                {
                    pop an edge (x,y) from the top of stack s;
                    write ( x, y );
                } until ( ( ( x, y ) = ( u, w ) ) ||
                    ( ( x, y ) = ( w, u ) ) );
            }
            bi_comp ( w, u );
            L[u] = min ( L[u], L[w] );
        }
        else
            if ( w != v )
                L[u] = min ( L[u], dfn[w] );
    }
}

```

Theorem 4 Algorithm *bi_comp* correctly generates the biconnected components of the connected graph G when G has at least two vertices.

Proof by induction on the number of biconnected components in G

Base case

- For all biconnected graphs G , root u of the depth-first spanning tree has only one child w
- w is the only vertex for which $L[w] \geq dfn[u]$ in *bi_comp*
- By the time w has been explored, all edges in G have been output as one biconnected component

Induction hypothesis

- Assume that *bi_comp* works correctly for all connected graphs G with at most m biconnected components

Induction step

- Let G be a graph with $m + 1$ connected components
- Consider the first time that $L[w] \geq dfn[u]$ in the line whose success writes `New bicomponent`
- At this time no edges have been output and so all edges in G incident to the descendants of w are on the stack and above the edge (u, w)
- Since none of the descendants of u is an articulation point and u is, the set of edges above (u, w) on the stack form a biconnected component together with (u, w)
- Once these edges have been deleted from the stack and output, the algorithm behaves essentially the same as it would on the graph G' obtained by deleting from G the biconnected component just output

- The behavior of algorithm on G differs from that on G' only in that during the exploration of vertex u , some edge (u, r) such that (u, r) is in the component just considered may be considered
 - However, for all such edges, $dfn[r] \neq 0$ and $dfn[r] > dfn[u] \geq L[u]$
 - Hence, the edges only result in a vacuous iteration of the `for` loop and do not materially affect the algorithm
 - It can be easily established that G' has at least two vertices
 - Since G' has exactly m biconnected components, it follows from the induction hypothesis that remaining components are correctly generated QED
- `Articulate` will work with any spanning tree relative to which the given graph has no cross edges
- * Graphs can have cross edges relative to breadth first spanning trees; hence `Articulate` cannot be adapted to BFS