Chapter 9: Graph Traversal

Graph Traversal

- In some cases, what is important is that the vertices are visited in a systematic order, regardless of the input graph. Usually, there are two methods of graph traversal:
  - Depth-first search
  - Breadth-first search

Depth-First Search

- Let $G=(V,E)$ be a directed or undirected graph.
- First, all vertices are marked unvisited.
- Next, a starting vertex is selected, say $v \in V$, and marked visited. Let $w$ be any vertex that is adjacent to $v$. We mark $w$ as visited and advance to another vertex, say $x$, that is adjacent to $w$ and is marked unvisited. Again, we mark $x$ as visited and advance to another vertex that is adjacent to $x$ and is marked unvisited.

Graph Representations

Option 1:
- Class Node
  - String: Name
  - List<Node>: Neighbors
  - List<Integer>: Costs
- End Node

Option 2:
- Class Node
  - String: Name
  - List<Edge>: Links
- End Node
- Class Edge
  - Integer: Cost
  - Node: toNode
  - Node: fromNode
- End Link

Depth-First Traversal

```
Traverse(Node: node)
  <Process node>
  For each edge in Links
    Traverse(edge.toNode)
  end for
End Traverse
```

Depth-First Traversal with Marking

```
Traverse(Node: node)
  <Process node>
  node.Visited = True
  For each edge in Links
    If not edge.toNode.Visited then
      Traverse(edge.toNode)
    end if
  end for
End Traverse
```
Depth-First Search

Example:

Depth-First Traversal with Time-Stamp

Traverse(Node: node)
    <Process node>
    node.StartTime = ++time // time is global
    For Each edge In Links
        If (edge.toNode.StartTime == 0) Then
            Traverse(edge.toNode)
        End If
    End for
    node.FinishTime = ++time // optional
    End Traverse

Non-Recursive DF Traversal

depthFirstTraverse(Node: start_node)
    start_node.Visited = True   // Visit this node.
    stack = new Stack<Node>();    stack.Push(start_node);
    // Repeat as long as the stack isn’t empty.
    While <stack isn’t empty>
        Node node = stack.Pop() // Get the next node from the stack.
        // Process the node’s links.
        For each edge In node.Links
            // if toNode hasn’t been visited…
            If (Not link.toNode.Visited) Then
                // Mark the node as visited.
                link.toNode.Visited = True
                // Push the node onto the stack.
                stack.Push(link.toNode)
            End If
        End for
    Loop // Continue processing the stack until empty
    End depthFirstTraverse

3 stages of a node: not visited (white), in stack (grey), exited stack (black)

Depth-First Search Tree

- When the search is complete, if all vertices are reachable from the start vertex, a spanning tree called the depth-first search spanning tree is constructed whose edges are those inspected in the forward direction.
- As a result of the traversal, the edges of an undirected graph G are classified into the following two types:
  - Tree edges: edges in the depth-first search tree.
  - Back edges: all other edges.

For undirected graphs, the edges are classified into 4 types: Tree edges, Back edges, Forward edges, and Cross edges.

Depth-First Search Forests

Input: An undirected graph G=(V, E);
Output: Preordering of the vertices in the corresponding depth-first search tree.
1. for each vertex v ∈ V
   2. Mark v unvisited;
   3. for each vertex w ∈ V
      4. If w is not marked dfs(w);
      a. dfs(w)
         1. Mark w visited;
         2. for each edge (v, w) ∈ E
            3. If w is not marked { P[w] = v, dfs(w); }
               // P[w] is the parent of v in DFS Tree.

Edge classification by DFS

Edge (u, v) of G is classified as a:

1. Tree edge iff u discovers v during the DFS: P[v] = u
2. If (u, v) is NOT a tree edge then it is a:
   (2) Forward edge iff u is an ancestor of v in the DFS tree
   (3) Back edge iff u is a descendant of v in the DFS tree
   (4) Cross edge iff u is neither an ancestor nor a descendant of v
**Edge classification by DFS**

- Tree edges
- Forward edges
- Back edges
- Cross edges

The edge classification depends on the particular DFS tree!

**DAGs and back edges**

- Can there be a **back** edge in a DFS on a Directed Acyclic Graph (DAG)?
  - NO! Back edges close a cycle!
- A graph $G$ is a DAG $\iff$ there is no back edge classified by DFS($G$)

**Breadth-First Traversal**

- Nodes close to the starting point are visited before those farther away

**Breadth-First Traversal**

```plaintext
BreadthFirstTraverse(Node: start_node)

// Visit this node.
start_node.Visited = True

// Make a stack and put the start node in it.
Queue[Node]: stack
queue.Add(start_node)

// Repeat as long as the queue isn't empty.
While <queue isn't empty>

  // Get the next node from the stack.
  Node node = queue.RemoveFirst()

  // Process the node's links.
  For Each link In node.Links
    // Use the link only if the destination node hasn't been visited.
    If (Not link.toNode.Visited) Then
      // Mark the node as visited.
      link.toNode.Visited = True
      queue.Add(link.toNode)
    End If
  Next link

End BreadthFirstTraverse
```

**Connectivity Testing**

- Perform a traversal (either DFS or BFS) and see if you reach every node
- Repeat as needed to find connected components
Bipartite Graphs

A graph \( G = (V, E) \) is bipartite if \( V \) can be partitioned into two subsets \( V = A \cup B \) such that \( E \) is a subset \( A \times B \).

Eg. All trees are bipartite.

Claim: A graph is bipartite iff it doesn’t have an odd-length cycle.

Claim: A graph can be 2-colored iff it’s bipartite.

Perform a traversal (either DFS or BFS) and see if you can use 2 colors to color two ends of each edge.

Topological sort

- We have a set of tasks and a set of dependencies (precedence constraints) of form “task A must be done before task B”
- **Topological sort**: An ordering of the tasks that conforms with the given dependencies
- **Goal**: Find a topological sort of the tasks or decide that there is no such ordering

Example

- **Scheduling**: When scheduling task graphs in distributed systems, usually we first need to sort the tasks topologically ...and then assign them to resources.

Topological sort more formally

- Is it possible to execute all the tasks in \( G \) in an order that respects all the precedence requirements given by the graph edges?
- The answer is "yes" if and only if the directed graph \( G \) has no cycle!
- (otherwise we have a deadlock)
- Such a \( G \) is called a Directed Acyclic Graph, or just a DAG

Algorithm for TS

- **TOPOLOGICAL-SORT(G)**:
  1. call DFS(G) to compute finishing times \( f[v] \) for each vertex \( v \)
  2. as each vertex is finished, insert it onto the front of a linked list
  3. return the linked list of vertices

- Note that the result is just a list of vertices in order of decreasing finish times \( f[.] \)
Example:

Finding Articulation Points in a Graph
- A vertex \( v \) in an undirected graph \( G \) with more than two vertices is called an articulation point if there exist two vertices \( u \) and \( w \) different from \( v \) such that any path between \( u \) and \( w \) must pass through \( v \).
- If \( G \) is connected, the removal of \( v \) and its incident edges will result in a disconnected subgraph of \( G \).
- A graph is called biconnected if it is connected and has no articulation points.

Articulation Points
- Example: \( c, b, g, h \) are articulation points.

Finding Articulation Points
- Example: \( c, b, g, h \) are articulation points.

Finding Articulation Points
- For each vertex \( v \) visited, we let \( \alpha[v] \) be the minimum of the following:
  - \( \alpha[v] \)
  - \( \alpha[u] \) for each vertex \( u \) such that \( (v, u) \) is a back edge
  - \( \beta[w] \) for each vertex \( w \) such that \( (v, w) \) is a tree edge

Thus, \( \beta[v] \) is the smallest \( \alpha \) that \( v \) can reach through back edges or tree edges.

Finding Articulation Points
- The articulation points are determined as follows:
  - The root is an articulation point if and only if it has two or more children in the depth-first search tree.
  - A vertex \( v \) other than the root is an articulation point if and only if \( v \) has a child \( w \) with \( \beta[w] \geq \alpha[v] \).
Finding Articulation Points

**Input**: A connected undirected graph \( G = (V, E) \);

**Output**: Boolean array \( \text{artpoint}[1…n] \) indicates the articulation points of \( G \), if any.

1. for each vertex \( v \in V \)
2. \( \text{visited}[v] \leftarrow \text{artpoint}[v] \leftarrow \text{false}; \)
3. time \( \leftarrow 0; \) rootdegree \( \leftarrow 0; \)
4. \( \text{dfs}(s); \) \( // \) \( s \) is the start vertex

**dfs(v)**

1. \( \text{visited}[v] = \text{true}; \) time++;
2. \( \alpha[v] \leftarrow \beta[v] \leftarrow \text{time}; \)
3. for each edge \( (v, w) \in E \)
4. if \( \text{visited}[w] = \text{false} \) then
5. \( p[w] \leftarrow v; \) \( \text{dfs}(w); \)
6. if \( v = s \) then
7. rootdegree \( \leftarrow \text{rootdegree} + 1; \)
8. if rootdegree\(=2 \) then \( \text{artpoint}[v] \leftarrow \text{true}; \)
9. else
10. \( \alpha[v] \leftarrow \min\{\alpha[v], \alpha[w]\}; \)
11. if \( \beta[w] \geq \alpha[v] \) then \( \text{artpoint}[v] \leftarrow \text{true}; \)
12. end if;
13. else if \( p[v] \neq w \) \( // (v, w) \) is a back edge
14. then \( \alpha[v] \leftarrow \min\{\alpha[v], \beta[w]\}; \)
15. end if;
16. end for;

Finding Articulation Points

**Example:**

```
    h
   / \
  i   j
 /
  a
 /  \
b   c
    \
  d
```

Strongly Connected Components

- Any directed graph can be partitioned into a unique set of strong components.
- The algorithm for finding the strong components of a directed graph \( G \) uses the transpose of the graph.
- The transpose \( G^T \) has the same set of vertices \( V \) as graph \( G \), but a new edge set consisting of the edges of \( G \) but with the opposite direction.

```
Strongly connected components in a directed graph.
```

Strongly Connected Components

- Execute the depth-first search \( \text{dfs()} \) for the graph \( G \) which creates the list \( \text{dfsList} \) consisting of the vertices in \( G \) in the reverse order of their finishing times.
- Generate the transpose graph \( G^T \).
- Using the order of vertices in \( \text{dfsList} \), make repeated calls to \( \text{dfs}() \) for vertices in \( G^T \). The list returned by each call is a strongly connected component of \( G \).
Strongly Connected Components

dfsList: [A, B, C, E, D, G, F]

Using the order of vertices in dfsList, make successive calls to dfs() for graph G'.

Vertex A: dfs(A) returns the list [A, C, B] of vertices reachable from A in G'.

Vertex E: The next unvisited vertex in dfsList is E. Calling dfs(E) returns the list [E].

Vertex D: The next unvisited vertex in dfsList is D; dfs(D) returns the list [D, F, G] whose elements form the last strongly connected component.

strongComponents()

// find the strong components of the graph;
// each element of component is a LinkedList
// of the elements in a strong component
public static <T> void strongComponents(DiGraph<T> g,
ArrayList<LinkedList<T>> component)
{
  T currVertex = null;
  // list of vertices visited by dfs() for graph g
  LinkedList<T> dfsList = new LinkedList<T> ();
  // list of vertices visited by dfsVisit()
  // for g transpose
  LinkedList<T> dfsGTList = null;
  // used to scan dfsList
  Iterator<T> gIter;
  // transpose of the graph
  DiGraph<T> gt = null;

  // clear the return vector
  component.clear();

  // execute depth-first traversal of g
  dfs(g, dfsList);

  // compute gt
  gt = transpose(g);

  // initialize all vertices in gt to WHITE (unvisited)
  gt.colorWhite();

  // call dfsVisit() for gt from vertices in dfsList
  gIter = dfsList.iterator();
  while(gIter.hasNext())
  {
    currVertex = gIter.next();
    // call dfsVisit() only if vertex
    // has not been visited
    if (gt.getColor(currVertex) == VertexColor.WHITE)
    {
      // create a new LinkedList to hold
      // next strong component
      dfsGTList = new LinkedList<T>();
      // do dfsVisit() in gt for starting
      // vertex currVertex
      dfsVisit(gt, currVertex, dfsGTList, false);
      // add strong component to the ArrayList
      component.add(dfsGTList);
    }
  }

}
Running Time of strongComponents()

- Recall that the depth-first search has running time $O(V+E)$, and the computation for $G^T$ is also $O(V+E)$. It follows that the running time for the algorithm to compute the strong components is $O(V+E)$.

Application of BFS

- Finding shortest distance in an unweighted graph.
- Claim: The distance from the root of BFS Tree to any node is the shortest distance from the root to that node in the graph.

Question: How to find the longest path in a graph?