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
- Stack-backed
- 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 $v$ and is marked unvisited. We treat $w$ in the same way as we did to $v$, and go on.

Graph Representations

Option 1:

Class Node

String: Name
Boolean: Visited
List<Node>: Neighbors
List<Integer>: Costs
End Node

Option 2:

Class Node

String: Name
Boolean: Visited
List<Edge>: Links
End Node

Class Edge

Integer: Cost
Node: toNode
Node: fromNode
End Link

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

Complexity: $O(n + m)$, $n$ and $m$ are the numbers of nodes and edges, resp.

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

Color of a node: white if StartTime is undefined; gray if StartTime is defined but FinishTime is undefined; black if FinishTime is defined.
**Depth-First Search**

- Example:

![Diagram of Depth-First Search Example]

**Non-Recursive DF Traversal**

```python
DepthFirstTraverse(Node: start_node):

start_node.Visited = True  # Visit this node.

Stack[Node]: stack;  // Make a stack and put the start node in it.
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 and set StartTime
link.toNode.Visited = True
stack.Push(link.toNode)
End If

End for  // Set FinishTime of node.
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 Forests**

- **Input**: (undirected or directed) 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 v ∈ V
4. if v is not marked dfs (v);

**Dfs (v)**

1. Mark v visited;
2. for each edge (v, w) ∈ E
3. if w is unvisited  { 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:

1. **Tree** edge if u discovers v during the DFS: P[v] = u
2. If (u,v) is NOT a tree then it is:
3. **Forward** edge if u is an ancestor of v in the DFS tree i.e., u.StartTime < v.StartTime and P[v] != u
4. **Back** edge if u is a descendant of v in the DFS tree i.e., u.FinishTime < v.FinishTime
5. **Cross** edge if u is neither an ancestor nor a descendant of v i.e. v.FinishTime < u.StartTime (v is black).
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 form a cycle!
- A graph \( G \) is a DAG \( \iff \) there is no back edge classified by DFS(\( G \))

Breadth-First Traversal

Complexity: \( O(n + m) \)

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 of $A \times B$.

E.g. 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.

Example:

```
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.
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

- To find the set of articulation points, we perform a depth-first search traversal on \( G \).
- During the traversal, we maintain two labels with each vertex \( v \in V \):
  - \( \alpha[v] \)
  - \( \beta[v] \)
- \( \alpha[v] \) is simply \( v \)'s start time in the depth-first search algorithm.
- \( \beta[v] \) is initialized to \( \alpha[v] \), but may change later on during the traversal.

Finding Articulation Points

- For each vertex \( v \) visited, we let \( \beta[v] \) 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

  \( \beta[v] \) is the smallest \( \alpha \) of those points 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 $artpoint[1...n]$ indicates the articulation points of $G$, if any.

1. for each vertex $v \in V$
2. { $\alpha[v] \leftarrow 0; \text{artpoint}[v] \leftarrow false; $}
3. $time \leftarrow 0; \text{rootdegree} \leftarrow 0;$
4. $dfs2(s); // s$ is the start vertex

$dfs2(v)$

2. $\alpha[v] \leftarrow \beta[v] \leftarrow ++time$
3. for each edge $(v, w) \in E$
4. if ($\alpha[w] == 0$) then // (v, w) is a tree edge
5. $p[w] \leftarrow v; dfs2(w);$
6. if $\text{v is the root}$
7. $\text{rootdegree} \leftarrow \text{rootdegree}+1;$
8. if rootdegree==2 then $\text{artpoint}[v] \leftarrow true;$
9. else // $v$ is not the root
10. $\text{if } \beta[w] \geq \alpha[v] \text{ then } \text{artpoint}[v] \leftarrow true;$
11. else // (v, w) is a back edge
12. $\beta[v] \leftarrow \text{min}\{\beta[v], \alpha[w]\};$
13. $\beta[w] \leftarrow \text{min}\{\beta[w], \alpha[v]\};$
14. end if;
15. end if;
16. end for;

Example:

$$
\begin{array}{c}
\text{A} \\
\text{B} \\
\text{C} \\
\text{D} \\
\text{E} \\
\text{F} \\
\text{G} \\
\text{H} \\
\text{I} \\
\text{J} \\
\end{array}
$$

Strongly Connected Components

Any directed graph can be partitioned into a unique set of strong components.

Use the depth-first search $dfs()$ for the graph $G$ which creates the list $dfsList$ 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 $dfsList$, make repeated calls to $dfs()$ for vertices in $G^T$. The list returned by each call is a strongly connected component of $G$. 

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

$\text{dfsList: } \{A, B, C, E, D, G, F\}$

Using the order of vertices in $\text{dfsList}$, make successive calls to $\text{dfs()}$ for graph $G^T$.

Vertex A: $\text{dfs(A)}$ returns the list $\{A, C, B\}$ of vertices reachable from A in $G^T$.

Vertex E: The next unvisited vertex in $\text{dfsList}$ is E. Calling $\text{dfs(E)}$ returns the list $\{E\}$.

Vertex D: The next unvisited vertex in $\text{dfsList}$ is D; $\text{dfs(D)}$ 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;

    strongComponents()
    (continued)
    // 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();

    strongComponents()
    (continued)
    // 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 $\text{strongComponents()}$

- 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?