COT 6405: Analysis of Algorithms
Giri NARASIMHAN
Relax Step

(a)

(b)
All Pairs Shortest Path Algorithm

- Invoke Dijkstra’s SSSP algorithm n times.
- Or use dynamic programming. How?
First Variant

- Let $D[i,j,m] = \text{length of the shortest path from } I \text{ to } j \text{ that uses at most } m \text{ edges}$
- $D[i,j,0] = ?$; $D[i,j,1] = ?$
- Recurrence Relation

\[
\begin{align*}
    l_{ij}^{(m)} &= \min_{\substack{1 \leq k \leq n}} \{l_{ij}^{(m-1)} + l_{ik}^{(m-1)} + w_{kj}\} \\
    &= \min_{\substack{1 \leq k \leq n}} \{l_{ik}^{(m-1)} + w_{kj}\}.
\end{align*}
\]
Second Variant

- \( C[i,j,k] = \) length of shortest path from \( i \) to \( j \) that only uses vertices from \( \{1, 2, \ldots, k\} \)

\[
d_{ij}^{(k)} = \begin{cases} 
  w_{ij} & \text{if } k = 0 \\
  \min \left( d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right) & \text{if } k \neq 0
\end{cases}
\]
Figure 14.38
Worst-case running times of various graph algorithms

<table>
<thead>
<tr>
<th>Type of Graph Problem</th>
<th>Running Time</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unweighted</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>Weighted, no negative edges</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>Weighted, negative edges</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>Weighted, acyclic</td>
<td>$O(</td>
<td>E</td>
</tr>
</tbody>
</table>
Figure 25.4 The sequence of matrices $D^{(i)}$ and $\Pi^{(i)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.
FLOYD-WARSHALL($W$)
1  $n = W.rows$
2  $D^{(0)} = W$
3  for $k = 1$ to $n$
4      let $D^{(k)} = d_{ij}^{(k)}$ be a new $n \times n$ matrix
5      for $i = 1$ to $n$
6          for $j = 1$ to $n$
7              $d_{ij}^{(k)} = \min d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$
8  return $D^{(n)}$
Main loops of Floyd-Warshall’s algorithm

\[
\text{for } k \leftarrow 1 \text{ to } n \\
\text{for } i \leftarrow 1 \text{ to } n \\
\text{for } j \leftarrow 1 \text{ to } n \\
\text{if } c_{ij} > c_{ik} + c_{kj} \then c_{ij} \leftarrow c_{ik} + c_{kj}
\]
Time Complexity

- Time Complexity = $O(n^3)$
- Improvements are possible with faster matrix multiplication algorithm.
Connectivity & Biconnectivity
Graph is **connected** if there exists a path between every pair of vertices.

A tree is **minimally connected**

Removing a edge/vertex from a **minimally connected** graph makes it disconnected.

Graph is **biconnected** if there exists 2 or more **disjoint** paths between every pair of vertices.

A cycle is **minimally biconnected**

You need to remove at least 2 vertices/edges to disconnect a **minimally biconnected** graph.

Every node lies on a cycle
Connected & Biconnected Components

- Subgraph $G'(V', E')$ is a connected component of $G(V, E)$ if $V'$ is a maximal subset of $V$ that induces a connected subgraph.

- If a graph is not connected, it can be decomposed into connected components.

- Subgraph $G'(V', E')$ is a biconnected component of $G(V, E)$ if $V'$ is a maximal subset of $V$ that induces a biconnected subgraph.

- If a graph is not biconnected, it can be decomposed into biconnected components.
What does DFS do for us?
Testing for Biconnectivity

- An **articulation point** is a vertex whose removal disconnects a graph.
- A **bridge** is an edge whose removal disconnects a graph.

**Claim:** If a graph is not biconnected, it must have an articulation point. **Proof?** "If and only if"?

How do we look for articulation points (and bridges)?

- Use **DFS**
If root of DFS tree has at least 2 children, it’s an articulation point
- Easy to check!

Non-root vertex \( u \) is an articulation point of \( G \) if and only if \( u \) has a child \( v \) such that there is no back edge from \( v \) or any descendant of \( v \) to a proper ancestor of \( u \)

Compute \( \text{Low}[x] = \) lowest numbered vertex reachable from some descendant of \( x \) (default is \( d[x] \))

Vertex \( u \) is an articulation point if \( \text{Low}[s] \geq d[u] \) for child \( s \) of \( u \)
DFS-VISIT(u)
1. VisitVertex(u)
2. Color[u] ← GRAY
3. Time ← Time + 1
4. d[u] ← Time
5. for each v ∈ Adj[u] do
6. VisitEdge(u, v)
7. if (v ≠ π[u]) then
8. if (color[v] = WHITE) then
  9. π[v] ← u
10. DFS-VISIT(v)
11. Color[u] ← BLACK
12. Low[u] ← min { Low[u], Low[v] } // back edge
13. else Low[u] = min { Low[u], d[v] } // back edge

BCC(G, u) // Compute the biconnected components of G
// starting from vertex u
1. Color[u] ← GRAY
2. Low[u] ← d[u] ← Time ← Time + 1
3. Put u on stack S
4. for each v ∈ Adj[u] do
  5. if (v ≠ π[u]) and (color[v] ≠ BLACK) then
    6. if (TopOfStack(S) ≠ u) then put u on stack S
    7. Put edge (u, v) on stack S
  8. if (color[v] = WHITE) then
    9. π[v] ← u
    10. BCC(G, v)
    11. if (Low[v] ≥ d[u]) then // u is an articulation point
        12. // Output next biconnected component
        13. Pop S until u is reached
        14. Push u back on S
    15. Low[u] = min { Low[u], Low[v] }
  16. else Low[u] = min { Low[u], d[v] } // back edge
Correctness and Complexity

- Theorem: A graph is biconnected if and only if it has no articulation points.
- BCC finds all articulation points:
  - If $\text{Low[child(u)]} \geq u$, then $u$ is an articulation point.
- Correctness follows from theoretical principles.
- Time and Space complexity $= O(n+m)$ Why?
How to detect bridges

- An edge $e$ of $G$ is a bridge if and only if it does not lie on any simple cycle of $G$
  - Use DFS, where every edge is a tree edge or back edge
  - If edge $e$ is a back edge?
    - It cannot be a bridge! Why?
  - If edge $e$ is a tree edge?
    - Let $e = (u,v)$ such that $u$ is the parent of $v$
    - Edge $e$ is a bridge if $\text{Low}[v] = d[v]$
Correctness and Complexity

- Correctness follows from the theoretical principles
- Time and Space complexity to detect all bridges in the graph
  - $O(n+m)$ Why?