Topological Sort

• Topological sort is the list of vertices in the reverse order of their finishing times (post-order) of the depth-first search.

Possible topological sort: R37  R137  R51  R63  R53  R151  R173  R263
Running Time for Topological Sort

- The topological sort uses the algorithm for dfs(), so its running time is also $O(V+E)$, where $V$ is the number of vertices in the graph and $E$ is the number of edges.
Strongly Connected Components

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

Components
\[ SC_1: \{ A, B, C \} \]
\[ SC_2: \{ E \} \]
\[ SC_3: \{ D, F, G \} \]

Strongly connected components in a directed graph.
Strongly Connected Components (continued)

• 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 (continued)

- Execute 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`. 
Strongly Connected Components (continued)
Strongly Connected Components (continued)

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

Using the order of vertices in dfsList, make successive calls to dfs() for graph $G^T$:

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

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.
public void SCC() {
    this.initSearch();
    ArrayList<Vertex> dfsList = new ArrayList<Vertex>();
    for (Vertex v : this.getVertices())
        if (!v.discovered) {
            v.discovered = true;
            SCCDFS(v, dfsList);
        }
    System.out.println("Reversed Post Order: " + dfsList);

    Graph G = this.reverse();
    System.out.println("Reversed Graph:\n" + G);
    G.initSearch();
    int n = 0;
    for (int i = dfsList.size()-1; i >= 0; i--) {
        Vertex v = dfsList.get(i);
        if (!v.discovered) {
            if (!v.discovered) {
                G.DFS2(v);
                System.out.println(" <= Component" + (++n));
            }
        }
    }
}
public void SCCDFS(Vertex v, ArrayList<Vertex> dl) {
    for (Vertex w : this.adjacentTo(v))
        if (!w.discovered) {
            w.discovered = true;
            SCCDFS(w, dl);
        }
    dl.add(v);
}

public Graph reverse() { // Create the reversed graph of the current one.
    Graph G = new Graph();
    for (Vertex v : this.getVertices())
        for (Vertex w : this.adjacentTo(v)) {
            G.addEdge(w, v);
        }
    return G;
}
Running Time of stronglyComponents()

- 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 $O(V+E)$. 
Minimum Spanning Tree

• A minimum spanning tree for a connected undirected graph is an acyclic set of edges that connect all the vertices of the graph with the smallest possible total weight.
  – A network connects hubs in a system. The minimum spanning tree links all of the nodes in the system with the least amount of cable.
Minimum Spanning Tree (continued)

Network of Hubs

Minimum spanning tree

Minimum amount of cable = 241
Prim's Algorithm

- The mechanics are very similar to the Dijkstra minimum-path algorithm.
  - The iterative process begins with any starting vertex and maintains two variables minSpanTreeSize and minSpanTreeWeight, which have initial values 0.
  - Each step adds a new vertex to the spanning tree. It has an edge of minimal weight that connects the vertex to those already in the minimal spanning tree.
Prim's Algorithm (continued)

- Prim's algorithm uses a priority queue where elements store a vertex and the weight of an edge that can connect it to the tree.
- When an object comes out of the priority queue, it defines a new edge that connects a vertex to the spanning tree.
Prim's Algorithm (continued)

Connected undirected graph to illustrate Prim’s algorithm.

Min Info (A, 0)

priority queue

<table>
<thead>
<tr>
<th></th>
<th>WHITE</th>
<th>WHITE</th>
<th>WHITE</th>
<th>WHITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>
Prim's Algorithm (continued)

Connected undirected graph to illustrate Prim’s algorithm.

Min Info (B, 2)  Min Info (C, 12)  Min Info (D, 5)
priority queue

<table>
<thead>
<tr>
<th>BLACK</th>
<th>WHITE</th>
<th>WHITE</th>
<th>WHITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

Min Info (B, 2)  Min Info (C, 12)  Min Info (D, 5)
priority queue

<table>
<thead>
<tr>
<th>BLACK</th>
<th>WHITE</th>
<th>WHITE</th>
<th>WHITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

A  B  C  D
Prim's Algorithm (continued)

Connected undirected graph to illustrate Prim’s algorithm.

<table>
<thead>
<tr>
<th>priority queue</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MinInfo (C, 12)</td>
<td>MinInfo (D, 5)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BLACK</th>
<th>BLACK</th>
<th>WHITE</th>
<th>WHITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

A
B
C
D
Prim's Algorithm (continued)

A connected undirected graph to illustrate Prim’s algorithm.

<table>
<thead>
<tr>
<th>Min Info (C, 12)</th>
<th>Min Info (C, 7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority queue</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BLACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

A

<table>
<thead>
<tr>
<th>BLACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

B

<table>
<thead>
<tr>
<th>WHITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>D</td>
</tr>
</tbody>
</table>

C

<table>
<thead>
<tr>
<th>BLACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

D
Prim's Algorithm (continued)

Original Graph

Vertex A
minSpanTreeSize = 1
minTreeWeight = 0
(a)

Minimum spanning tree: adding vertices A, B, D, and C.
Running Time for Prim's Algorithm

- Like Dijkstra's algorithm, Prim's algorithm has running time $O(V + E \log_2 V)$. 
The Shortest Path Algorithm

• Each vertex must maintain a record of its parent and its path length from sVertex. The DiGraph class provides methods that allow the programmer to associate two fields of information with a vertex.
  
  – One field identifies the parent of a vertex and the other field is an integer dataValue associated with the vertex. The method initData() assigns a representation for $\infty$ to each dataValue field of the graph vertices.
Dijkstra's Minimum-Path Algorithm

• The minimum path problem is to determine a path of minimum weight from a starting vertex $v_s$ to each reachable vertex in the graph.
Dijkstra's Minimum-Path Algorithm

Three paths, A-B-E, A-C-E, and A-D-E, have path length 2, with weights 15, 17, and 13 respectively. The minimum path is A-C-D-E, with weight 11 but path length 3.
Dijkstra's Minimum-Path Algorithm (continued)

• Define the distance from every node to the starting node: Initially the distance is 0 to the starting node and infinity to other nodes.

• Use a priority queue to store all the nodes with distance (the minimal distance node on the top).

• Each step in the algorithm removes a node from the priority queue and update the distance to other nodes in the queue via this node.
Dijkstra's Minimum-Path Algorithm (continued)

• Since no subsequent step could find a new path to the node with a smaller weight (because the weights are positive), we have found the minimum path to this node.

• Repeat this above step and the algorithm terminates whenever the priority queue becomes empty.
Dijkstra's Minimum-Path Algorithm (continued)

(a) Setup: Push MI(A, 0) into the priority queue.
(b) Pop MI(A, 0). Push MI(B, 3), MI(C, 4), and MI(D, 8) into the priority queue.
(c) Pop MI(B, 3). Push MI(E, 15) into the priority queue.
(d) Pop MI(C, 4). Push MI(D, 6) into the priority queue.
(e) Pop MI(D, 6). Push MI(E, 11) into the priority queue.
(f) Pop MI(D, 8)
(g) Pop MI(E, 11)
Running Time of Dijkstra's Algorithm

• Dijkstra's algorithm has running time $O(V + E \log_2 V)$. 
Minimum Path in Acyclic Graphs

• When the weighted digraph is acyclic, the problem of finding minimum paths is greatly simplified.
  – The depth-first search creates a list of vertices in topological order.
    dfsList : \([v_0, v_1, \ldots, v_i, \ldots, v_{n-1}]\)
  – Assume \(v_i\) is the starting vertex for the minimum-path problem. Vertices in the list \(v_0\) to \(v_{i-1}\) are not reachable from \(v_i\).
Minimum Path in Acyclic Graphs (continued)

- After initializing the data value for all of the vertices to $\infty$, set the data value for $v_i$ to 0 and its parent reference to $v_i$. A scan of the vertices in dfsList will find that $v_0$ through $v_{i-1}$ have value $\infty$ and will not be considered. The algorithm discovers $v_i$ and iteratively scans the tail of dfsList, proceeding much like Dijkstra's algorithm.
Minimum Path in Acyclic Graphs (continued)

- At a vertex v in the sequential scan of dfsList, its current data value is the minimum path weight from vᵢ to v. For there to be a better path, there must be an unvisited vertex, v', reachable from vᵢ that has an edge to v. This is not possible, since topological order guarantees that v' will come earlier in dfsList.
Minimum Path in Acyclic Graphs (continued)
Running Time for Minimum Path in Acyclic Graphs

• The algorithm first creates a topological sort of the vertices with running time $O(V+E)$. A loop visits all of the vertices in the graph once and examines edges that emanate from each vertex only once. Access to all of the vertices and edges has running time $O(V+E)$ and so the total running time for the algorithm is $O(V+E)$. 