Quiz 1

One advantage of adjacency list representation over adjacency matrix representation of a graph is that in adjacency list representation, space is saved for sparse graphs.

A. True
B. False
Quiz 1

One advantage of adjacency list representation over adjacency matrix representation of a graph is that in adjacency list representation, space is saved for sparse graphs.

A. True
B. False
Quiz 2

Traversal of a graph is different from tree because

A. There can be a loop in graph so we must maintain a visited flag for every vertex
B. DFS of a graph uses stack, but inorder traversal of a tree is recursive
C. BFS of a graph uses queue, but a time efficient BFS of a tree is recursive.
D. All of the above
Quiz 2

Traversal of a graph is different from tree because

A. There can be a loop in graph so we must maintain a visited flag for every vertex
B. DFS of a graph uses stack, but inorder traversal of a tree is recursive
C. BFS of a graph uses queue, but a time efficient BFS of a tree is recursive.
D. All of the above
Quiz 3

One possible order of Breadth First Search on the following graph

A. MNOPQR
B. NQMPOR
C. QMNPRO
D. QMNPOR
Quiz 3

One possible order of Breadth First Search on the following graph

A. MNOPQR
B. NQMPOR
C. QMNPOR
D. QMNPRO

A. MNOPQR
B. NQMPOR
C. QMNPOR
D. QMNPRO
Quiz 4

Given two vertices in a graph 1 and 6, which of the two traversals (BFS and DFS) can be used to find if there is path from 1 to 6?

A. Only BFS  
B. Only DFS  
C. Both BFS and DFS  
D. Neither BFS nor DFS
Quiz 4

Given two vertices in a graph 1 and 6, which of the two traversals (BFS and DFS) can be used to find if there is a path from 1 to 6?

A. Only BFS
B. Only DFS
C. Both BFS and DFS
D. Neither BFS nor DFS
Quiz 5

Consider the DAG with Consider V = {1, 2, 3, 4, 5, 6}, shown below. Which of the following is NOT a topological ordering?

A. 1 2 3 4 5 6
B. 1 3 2 4 5 6
C. 1 3 2 4 6 5
D. 3 2 4 1 6 5
Quizzes 5

Consider the DAG with Consider V = {1, 2, 3, 4, 5, 6}, shown below. Which of the following is NOT a topological ordering?

A. 1 2 3 4 5 6
B. 1 3 2 4 5 6
C. 1 3 2 4 6 5
D. 3 2 4 1 6 5
Shortest Paths
Shortest paths

Given an edge-weighted digraph, find the shortest path from $s$ to $t$. 

```
shortest path from 0 to 6
0->2  0.26
2->7  0.34
7->3  0.39
3->6  0.52
```
Shortest path variants

- Which vertices?
  - **Single source**: from one vertex $s$ to every other vertex.
  - **Source-sink**: from one vertex $s$ to another $t$.
  - **All pairs**: between all pairs of vertices.

- Restrictions on edge weights?
  - **Nonnegative weights**.
  - **Arbitrary weights**.

- Cycles?
  - **No directed cycles**.
  - **No "negative cycles."**

- Simplifying assumption: Shortest paths from $s$ to each vertex $v$ exist.
Weighted directed edge

public class DirectedEdge

    DirectedEdge(int v, int w, double weight)

    int from()

    int to()

    double weight()

    String toString()

Idiom for processing an edge e: int v = e.from(), w = e.to();
public class DirectedEdge{
    private final int v, w;
    private final double weight;

    public DirectedEdge(int v, int w, double weight){
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public int from() { return v; }
    public int to() { return w; }
    public double weight() { return weight; }
}
public class EdgeWeightedDigraph

EdgeWeightedDigraph(int V)  

void addEdge(DirectedEdge e)  

Iterable<DirectedEdge> adj(int v)

int V()  

int E()

Iterable<DirectedEdge> edges()

String toString()

Conventions. Allow self-loops and parallel edges.
Edge-weighted digraph: adjacency-lists representation
Edge-weighted digraph implementation

```java
public class EdgeWeightedDigraph{
    private final int V;
    private final Bag<DirectedEdge>[] adj;

    public EdgeWeightedDigraph(int V){
        this.V = V;
        adj = (Bag<DirectedEdge>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<DirectedEdge>();
    }

    public void addEdge(DirectedEdge e){
        int v = e.from();
        adj[v].add(e);
    }

    public Iterable<DirectedEdge> adj(int v){
        return adj[v];
    }
}
```
Single-source shortest paths

What is the shortest distance and path from A to H?
Single-source shortest paths

- **Data structures:** Represent the Shortest Path with two vertex-indexed arrays:
  - \texttt{distTo[v]} is length of shortest path from \texttt{s} to \texttt{v}.
  - \texttt{edgeTo[v]} is last edge on shortest path from \texttt{s} to \texttt{v}.

\begin{verbatim}
public double distTo(int v){
    return distTo[v];
}

public Iterable&lt;DirectedEdge&gt; pathTo(int v){
    Stack&lt;DirectedEdge&gt; path = new Stack&lt;DirectedEdge&gt;();
    DirectedEdge e = edgeTo[v];
    while (e != null){
        path.push(e);
        e = edgeTo[e.from()];
    }
    return path;
}
\end{verbatim}
Edge relaxation

- Relax edge $e = v \rightarrow w$.
  - $\text{distTo}[v]$ is length of shortest known path from $s$ to $v$.
  - $\text{distTo}[w]$ is length of shortest known path from $s$ to $w$.
  - $\text{edgeTo}[w]$ is last edge on shortest known path from $s$ to $w$.
  - If $e = v \rightarrow w$ gives shorter path to $w$ through $v$, update both $\text{distTo}[w]$ and $\text{edgeTo}[w]$.

$v \rightarrow w$ successfully relaxes
Edge relaxation

- Relax edge \( e = v \rightarrow w \).
  - \( \text{distTo}[v] \) is length of shortest known path from \( s \) to \( v \).
  - \( \text{distTo}[w] \) is length of shortest known path from \( s \) to \( w \).
  - \( \text{edgeTo}[w] \) is last edge on shortest known path from \( s \) to \( w \).
  - If \( e = v \rightarrow w \) gives shorter path to \( w \) through \( v \), update both \( \text{distTo}[w] \) and \( \text{edgeTo}[w] \)

```java
private void relax(DirectedEdge e) {
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight()) {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
```
Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.
Repeat until optimality conditions are satisfied:
   Relax any edge.

Efficient implementations: How to choose which edge to relax?
- Dijkstra's algorithm (nonnegative weights).
- Topological sort algorithm (no directed cycles).
- Bellman-Ford algorithm (no negative cycles).
Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex.
Dijkstra's algorithm Demo

Pick vertex in List with minimum distance.

![Graph Diagram]

<table>
<thead>
<tr>
<th>V</th>
<th>distTo[]</th>
<th>edgeTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>--</td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>
Update A’s neighbors
Update D’s neighbors
Update B’s neighbors

No Update
Update E’s neighbors

No Update
Update C’s neighbors
Update G’s neighbors

![Graph with nodes A, B, C, D, E, F, G and edges with weights]

<table>
<thead>
<tr>
<th>V</th>
<th>distTo[]</th>
<th>edgeTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>--</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
<td>G</td>
</tr>
<tr>
<td>G</td>
<td>5</td>
<td>D</td>
</tr>
</tbody>
</table>
Update F’s neighbors

No Update
Dijkstra's algorithm Demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex.
Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges pointing from that vertex.
Dijkstra's algorithm Implementation

public class DijkstraSP{
    private DirectedEdge[] edgeTo;
    private double[] distTo;
    private IndexMinPQ<Double> pq;

    public DijkstraSP(EdgeWeightedDigraph G, int s) {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        pq = new IndexMinPQ<Double>(G.V());
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;
        pq.insert(s, 0.0);
        while (!pq.isEmpty()){
            int v = pq.delMin();
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }
}

Shortest Path Demo
Shortest Path Demo
Shortest Path Demo

\[ \text{A} \rightarrow \text{B} \rightarrow \text{C} \rightarrow \text{D} \]

Weights:
- \(100\) on the edge from \(A\) to \(C\)
- \(-5000\) on the edge from \(C\) to \(D\)
- \(1\) on the edge from \(B\) to \(C\)
- \(1\) on the edge from \(B\) to \(D\)
Acyclic shortest paths

- Consider vertices in topological order. Relax all edges pointing from that vertex.

0 1 4 7 5 2 3 6
Acyclic shortest paths

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.
Longest paths in edge-weighted DAGs

- Formulate as a shortest paths problem in edge-weighted DAGs.
  - Negate all weights.
  - Find shortest paths.
  - Negate weights in result
- Key point. Topological sort algorithm works even with negative weights.

<table>
<thead>
<tr>
<th>longest paths input</th>
<th>shortest paths input</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-&gt;4 0.35</td>
<td>5-&gt;4 -0.35</td>
</tr>
<tr>
<td>4-&gt;7 0.37</td>
<td>4-&gt;7 -0.37</td>
</tr>
<tr>
<td>5-&gt;7 0.28</td>
<td>5-&gt;7 -0.28</td>
</tr>
<tr>
<td>5-&gt;1 0.32</td>
<td>5-&gt;1 -0.32</td>
</tr>
<tr>
<td>4-&gt;0 0.38</td>
<td>4-&gt;0 -0.38</td>
</tr>
<tr>
<td>0-&gt;2 0.26</td>
<td>0-&gt;2 -0.26</td>
</tr>
<tr>
<td>3-&gt;7 0.39</td>
<td>3-&gt;7 -0.39</td>
</tr>
<tr>
<td>1-&gt;3 0.29</td>
<td>1-&gt;3 -0.29</td>
</tr>
<tr>
<td>7-&gt;2 0.34</td>
<td>7-&gt;2 -0.34</td>
</tr>
<tr>
<td>6-&gt;2 0.40</td>
<td>6-&gt;2 -0.40</td>
</tr>
<tr>
<td>3-&gt;6 0.52</td>
<td>3-&gt;6 -0.52</td>
</tr>
<tr>
<td>6-&gt;0 0.58</td>
<td>6-&gt;0 -0.58</td>
</tr>
<tr>
<td>6-&gt;4 0.93</td>
<td>6-&gt;4 -0.93</td>
</tr>
</tbody>
</table>
Longest paths in edge-weighted DAGs

- Parallel job scheduling.
  - Given a set of jobs with durations and precedence constraints, schedule the jobs (by finding a start time for each) so as to achieve the minimum completion time, while respecting the constraints.

<table>
<thead>
<tr>
<th>job</th>
<th>duration</th>
<th>must complete before</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>41.0</td>
<td>1 7 9</td>
</tr>
<tr>
<td>1</td>
<td>51.0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>50.0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>36.0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>38.0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>45.0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>21.0</td>
<td>3 8</td>
</tr>
<tr>
<td>7</td>
<td>32.0</td>
<td>3 8</td>
</tr>
<tr>
<td>8</td>
<td>32.0</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>29.0</td>
<td>4 6</td>
</tr>
</tbody>
</table>

Parallel job scheduling solution
Critical path method

- To solve a parallel job-scheduling problem, create edge-weighted DAG:
  - Source and sink vertices.
  - Two vertices (begin and end) for each job.
  - Three edges for each job.
    - Begin to end (weighted by duration)
    - Source to begin (0 weight)
    - End to sink (0 weight)
  - One edge for each precedence constraint (0 weight).
Critical path method

Use longest path from the source to schedule each job.
Quiz 1

There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra’s shortest path algorithm?

A. SDT
B. SBDT
C. SACDT
D. SACET
There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra’s shortest path algorithm?

A. SDT  
B. SBDT  
C. SACDT  
D. SACET
In an unweighted, undirected connected graph, the shortest path from a node S to every other node is computed most efficiently, in terms of time complexity by

A. Dijkstra’s algorithm starting from S.
B. Performing a DFS starting from S.
C. Performing a BFS starting from S.
D. None of the above
In an unweighted, undirected connected graph, the shortest path from a node S to every other node is computed most efficiently, in terms of time complexity by

A. Dijkstra’s algorithm starting from S.
B. Performing a DFS starting from S.
C. Performing a BFS starting from S.
D. None of the above