Last Time

• Minimum spanning trees
• Lazy and Eager Prim Algorithms
• Kruskal’s Algorithm
But Wait, We Did Shortest Path with BFS...

• Yes, we did, but now our edges have weights!
  – A path that takes 4 edges could have a smaller total weight than a path that takes 2 edges.

• A **shortest path** from vertex $s$ to vertex $t$ in an edge-weighted digraph is a directed path from $s$ to $t$ with the property that no other such path has a lower weight.
Shortest Path

<table>
<thead>
<tr>
<th>Path</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-&gt;5</td>
<td>0.35</td>
</tr>
<tr>
<td>5-&gt;4</td>
<td>0.35</td>
</tr>
<tr>
<td>4-&gt;7</td>
<td>0.37</td>
</tr>
<tr>
<td>5-&gt;7</td>
<td>0.28</td>
</tr>
<tr>
<td>7-&gt;5</td>
<td>0.28</td>
</tr>
<tr>
<td>5-&gt;1</td>
<td>0.32</td>
</tr>
<tr>
<td>0-&gt;4</td>
<td>0.38</td>
</tr>
<tr>
<td>0-&gt;2</td>
<td>0.26</td>
</tr>
<tr>
<td>7-&gt;3</td>
<td>0.39</td>
</tr>
<tr>
<td>1-&gt;3</td>
<td>0.29</td>
</tr>
<tr>
<td>2-&gt;7</td>
<td>0.34</td>
</tr>
<tr>
<td>6-&gt;2</td>
<td>0.40</td>
</tr>
<tr>
<td>3-&gt;6</td>
<td>0.52</td>
</tr>
<tr>
<td>6-&gt;0</td>
<td>0.58</td>
</tr>
<tr>
<td>6-&gt;4</td>
<td>0.93</td>
</tr>
</tbody>
</table>

[Diagram of a graph showing shortest path from 0 to 6]

shortest path from 0 to 6

<table>
<thead>
<tr>
<th>Path</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-&gt;2</td>
<td>0.26</td>
</tr>
<tr>
<td>2-&gt;7</td>
<td>0.34</td>
</tr>
<tr>
<td>7-&gt;3</td>
<td>0.39</td>
</tr>
<tr>
<td>3-&gt;6</td>
<td>0.52</td>
</tr>
</tbody>
</table>
Shortest Path Assumptions

• Paths are directed. Undirected edges can be considered as a pair of directed edges. We must respect the direction shown by the edge.
• Weights are (again) not necessarily distances, nor are they necessarily costs (but they could be).
• Not all vertices need to be reachable. If no path exists from \( s \) to \( t \), then it follows that there is no shortest path.
• Negative weights make things tough, so we’ll assume all weights are positive or zero.
• Shortest paths are not necessarily unique.
Relaxation

- **Relaxation** is a process for determining the shortest path by a bit of induction.
  - In *edge relaxation*, we test whether the best known way from $s$ to $w$ is to go from $s$ to $v$, then take the edge from $v$ to $w$, and if so, update our data structures to indicate that this is the case.
  - In *vertex relaxation*, we relax all of the edges from a given vertex at the same time, keeping any of the new shortest paths found in our data structures.
Relaxation

before

after

black edges are in edgeTo[]

weight of v→w is 1.3

distTo[v]

distTo[w]

no changes

still ineligible

now ineligible

edgeTo[w]

no longer in SPT

3.1

4.4

7.2
Dijkstra’s Algorithm

• Basic idea:
  – Build off of the idea behind Eager Prim.
  – Initialize $\text{distTo}[s]$ (starting vertex) to 0, and $\text{distTo}[\text{everything}_\text{else}]$ to positive infinity.
  – Relax and add to the tree a non-tree vertex with the lowest $\text{distTo}[\cdot]$ value.
  – Repeat until all vertices are on the tree, or no non-tree vertex has a finite $\text{distTo}[\cdot]$ value.
Dijkstra’s Algorithm

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;
    } //for
    distTo[s] = 0.0;
    pq.insert(s, 0.0);
    while (!pq.isEmpty()) {
        relax(G, pq.delMin());
    } //while
} //DijkstraSP
void relax(EdgeWeightedDigraph G, int v) {
    for (DirectedEdge e: G.adj(v)) {
        int w = e.to();
        if (distTo[w] > distTo[v] + e.weight()) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (pq.contains(w)) {
                pq.changeKey(w, distTo[w]);
            } else {
                pq.insert(w, distTo[w]);
            } //if-else
        } //if
    } //for
} //DijkstraSP
Dijkstra’s Algorithm – Visual

```
red: on pq
black: on SPT
```

```
edgeTo[]
distTo[]
index
priority
```

```
0
1 5→1 0.32 1.05
2 0→2 0.26 0.26
3 7→3 0.37 0.97
4 0→4 0.38 0.38
5 4→5 0.35 0.73
6 7→7 0.34 0.60
0
1 5→1 0.32 1.05
2 0→2 0.26 0.26
3 7→3 0.37 0.97
4 0→4 0.38 0.38
5 4→5 0.35 0.73
6 3→6 0.52 1.49
7 2→7 0.34 0.60
0
1 5→1 0.32 1.05
2 0→2 0.26 0.26
3 7→3 0.37 0.97
4 0→4 0.38 0.38
5 4→5 0.35 0.73
6 3→6 0.52 1.49
7 2→7 0.34 0.60
0
1 5→1 0.32 1.05
2 0→2 0.26 0.26
3 7→3 0.37 0.97
4 0→4 0.38 0.38
5 4→5 0.35 0.73
6 3→6 0.52 1.49
7 2→7 0.34 0.60
```
Dijkstra’s Algorithm – Analysis

• **Theorem:** Dijkstra’s algorithm uses extra space proportional to $V$ and time proportional to $E \times \log(V)$ (in the worst case)

• **Proof:** Identical to the proof for Eager Prim.
Can We Do Better?

• In some cases, yeah.
• What if we know that our digraph is acyclic?
  – Now we can handle negative weights. (why?)
  – We can now solve the problem in linear time.
  – Make use of our knowledge of Topological sort.
  – We can also solve the longest path problem! (how?)
AcyclicSP

AcyclicSP(EdgeWeightedDigraph G, int s) {
    edgeTo = new DirectedEdge[G.V()];
    distTo = new double[G.V()];
    for (int v = 0; v < G.V(); v++) {
        distTo[v] = Double.POSITIVE_INFINITY;
    } //for
    distTo[s] = 0.0;
    Topological top = new Topological(G);
    for (int v: top.order()) {
        relax(G, v);
    } //for
} //AcyclicSP
Acyclic Edge-Weighted Digraph
AcyclicSP – Visual

topological sort

5 1 3 6 4 7 0 2

edgeTo[]

0
1 5->1
2
3
4 5->4
5
6
7 5->7

thick black: on tree

red: add to tree

gray: ineligible

0 4->0
1 5->1
2 6->2
3 1->3
4 5->4
5
6 3->6
7 5->7

0 4->0
1 5->1
2 7->2
3 1->3
4 5->4
5
6 3->6
7 5->7

0 4->0
1 5->1
2 6->2
3 1->3
4 5->4
5
6 3->6
7 5->7
**AcyclicSP – Analysis**

- **Theorem:** By relaxing vertices in topological order, we can solve the shortest path problem for edge-weighted DAGs in time proportional to $E + V$.
- **Proof:**
  - Every edge $v \rightarrow w$ is relaxed exactly once, when $v$ is relaxed, leaving $\text{distTo}[w] \leq \text{distTo}[v] + e\text{.weight}()$.
  - This inequality holds until the algorithm completes, because $\text{distTo}[v]$ never updates (no edge pointing to $v$ will be relaxed after $v$ is relaxed).
  - After all vertices reachable from $s$ have been added to the tree, the shortest-path optimality condition holds.
  - The Topological sort pass takes $E + V$, and the second relaxation pass completes the job by relaxing each edge once: another $E + V$. ■
Let’s Make Things Complicated

• What could happen if we introduce negative edges into a graph with a cycle?
  – (bad things)
  – We can only have a shortest path if there are no **negative cycles** in the graph.
Bellman-Ford Algorithm

Staring at the ceiling, she asked me what I was thinking about.

I should have made something up.

The Bellman-Ford algorithm makes terrible pillow talk.

xkcd.com/69/
General Shortest Paths

• There exists a shortest path from $s$ to $v$ in an edge-weighted digraph iff and only if there exists at least one directed path from $s$ to $v$ and no vertex on any directed path from $s$ to $v$ is on a negative cycle.
Possible Negative Cycle Fixes

• **Option 1**: Let’s traverse the graph to find the smallest edge (the greatest negative value), then add (0-val) to all edges to make them all 0 or positive.
  – Fails because the 0+ graph could potentially have a different shortest path than the original graph.

• **Option 2**: Let’s adapt Dijkstra somehow to allow negative edges.
  – Fails because Dijkstra relies on examining vertices in increasing distance from the source. An edge with a negative weight will make the path to w in s→v→w shorter than the path to v in s→v, so the assumption that Dijkstra is built upon fails.

• **Option 3**: Whether or not there are negative cycles, there must exist a shortest simple path from s to t, so let’s just define an algorithm to seek that path.
  – We can... but the best algorithm we have for that runs in exponential time in the worst case.
Bellman-Ford Solution

• Assign a shortest-path weight of +INF to vertices that are not reachable from the source. (easy)

• Assign a shortest-path weight of −INF to vertices that are on a negative cycle. (easy?)

• Compute the shortest-path tree for all other vertices. (easy)
Identifying Negative Cycles

• Let’s call this “naïve Bellman-Ford” (doesn’t work with negative cycles)
  – Initialize distTo[s] to 0 and other distTo[] to +INF.
  – Considering the digraph edges in any order, relax all edges. Repeat $V$ times.

• Why does this work?
  – If a shortest path exists from $s$ to $t$, then it consists of a sequence of vertices $v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k$.
  – Since there are no negative cycles, this path has a maximum length of $V - 1$.
  – By induction, after pass $i$, we will find the shortest path from $s$ to $v_i$.
    • The base case shortest path to $s=0$.
    • We relax every edge on the $i^{th}$ pass, including edge $v_i \rightarrow v_{i+1}$, so distTo[v_{i+1}] is no greater than distTo[v_i] + edge weight $v_i \rightarrow v_{i+1}$.
    • Before the $i^{th}$ pass, distTo[v_i] is the smallest possible. After, distTo[v_{i+1}] is the smallest possible.
Naïve Bellman-Ford

• **Theorem:** Naïve Bellman-Ford uses extra space proportional to $V$ and time proportional to $E \times V$ (in all cases)

• **Proof:**
  – We still need to keep track of $V$ distTo values.
  – On each of the $V$ passes, we relax $E$ edges. ■
Queue-Based Bellman-Ford

• We can easily determine that numerous edges will not lead to a successful relaxation in a given pass – the only edges that will change $\text{distTo}[w]$ are those leaving a vertex that updated $\text{distTo}[v]$ on the previous pass.

• We can use a Queue like we did with BFS to order these vertices:
  – If vertex $v$ is successfully relaxed at pass $i$, then all vertices $w$ can be relaxed at pass $i + 1$. They will hence come afterward in the Queue.
Queue-Based Bellman-Ford

BellmanFordSP(EDW G, int S) {
    distTo = new double[G.V()];
    edgeTo = new double[G.V()];
    onQ = new boolean[G.V()];
    queue = new Queue<Integer>();
    for (int v=0; v<G.V(); v++) {
        distTo[v] = Double.POSITIVE_INFINITY;
    }
    distTo[s]=0;
    queue.enqueue(s);
    onQ[s]=true;

    while (!queue.isEmpty() && !hasNegativeCycle()) {
        int v = queue.dequeue();
        onQ[v] = false;
        relax(G, v);
    }
}

04/11/2016
Queue-Based Bellman-Ford

```
relax(EDW G, int V) {
    for (DirectedEdge e : G.adj(v)) {
        int w = e.to();
        if (distTo[w] > distTo[v] + e.weight()) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (!onQ[w]) {
                queue.enqueue(w);
                onQ[w] = true;
            }//if
        }//if
    } //for
    if (cost++ % G.V() == 0) {
        findNegativeCycle();
    } //if
} //relax
```
Queue-Based Bellman-Ford

• Queue-Based Bellman-Ford will either:
  – Find a shortest path tree from $s$
  – Find a negative cycle reachable from $s$

• ...in time proportional to $E \times V$ and extra space proportional to $V$.

• If there is no negative cycle, the algorithm terminates after all relaxations corresponding to the $(V - 1)^{st}$ pass of naive BF. Same proof.

• If there is a negative cycle, the queue never empties, so we have a check to see if we ever hit pass $V$. 
Finding a Negative Cycle

```java
findNegativeCycle() {
    int V = edgeTo.length;
    EdgeWeightedDigraph spt = new EWD();
    for (int v=0; v<V; v++) {
        if (edgeTo[v] != null) {
            spt.addEdge(edgeTo[v]);
        } //if
    } //for

    EdgeWeightedDirectedCycle cf = new EWDC(spt);
    cycle = cf.cycle();
} //findNegativeCycle
```
Any Questions?