Last Time

• Directed graphs and terminology
• Depth- and breadth-first search
• Finding directed cycles
• Topological sort
• Strongly connected components
Strong Connectedness

• Two vertices \( v \) and \( w \) are **strongly connected** if they are mutually reachable.
  – A directed path exists from \( v \) to \( w \).
  – A directed path exists from \( w \) to \( v \).

• A digraph is strongly connected if all its vertices are strongly connected to one another.
Strong Connectedness

• We can also use the idea of strong connectedness to partition a graph into strongly connected components (or strong components) – subsets of the graph that are internally strongly connected.
Kosaraju-Sharir Algorithm

- Start by building a **condensation digraph** – shrink all connected components into a single node for each.
  - The result must be a DAG, because a directed cycle would imply a larger strongly connected component.
Kosaraju-Sharir Algorithm

- Since the condensation digraph is a DAG, we can place its vertices in a topological order.
Kosaraju-Sharir Algorithm

• We want to start at one of the nodes in the sink, and see which other nodes we can visit (and check for cycles).

• Because the graph that currently exists points from source to sink, we want to reverse all of the edges so that they point from sink to source.
Kosaraju-Sharir Algorithm

public KosarajuSharirCC(Digraph G) {
    marked = new boolean[G.V()];
    id = new int[G.V()];
    DepthFirstOrder order = new DepthFirstOrder(G.reverse());
    for (int s : order.reversePostorder) {
        if (!marked(s)) {
            dfs(G, s);
            count++;
        } //if
    } //for
} //KosarajuSharirCC

private void dfs(Digraph G, int v) {
    ...
    id[v] = count;
    ...
} //dfs
Kosaraju-Sharir Algorithm – Visual

DFS in original digraph

DFS in reverse digraph (ReversePost)
Transitive Closure

• The **transitive closure** of a digraph $G$ is another digraph with the same set of vertices, but with an edge from $v$ to $w$ in the transitive closure if and only if $w$ is reachable from $v$ in $G$. 
Transitive Closure

private DirectedDFS[] all;

TransitiveClosure(Digraph G) {
    all = new DirectedDFS[G.V()];
    for (int v = 0; v < G.V(); v++) {
        all[v] = new DirectedDFS(G, v);
    } // for
} // TransitiveClosure (constructor)

boolean reachable(int v, int w) {
    return all[v].marked(w);
} // reachable
Edge-Weighted Graphs

• To this point, we were assuming that all edges are equal.

• An **edge-weighted graph** is a graph model where we associate **weights** with each edge.
  
  – Weights are not necessarily distances (but can be) or costs (but also can be).
  
  – Weights may be zero or negative.
Minimum Spanning Tree

• A **spanning tree** of a graph is a connected subgraph with no cycles that includes all the vertices.

• A **minimum spanning tree** (**MST**) of an edge-weighted graph is a spanning tree whose total weight is no larger than the weight of any other spanning tree.

![Diagram](image-url)
Storing Edge-Weighted Graphs

Bag objects

references to the same Edge object
More terminology

• A **cut** of a graph is a partition of its vertices into two nonempty, disjoint sets.
  - Typically specified as a set of vertices, with the implicit assumption that the missing vertices are a part of the other set.

• A **crossing edge** of a cut is an edge that connects a vertex in one set with a vertex in the other.

• **Greedy algorithms** are a class of algorithms that always take the best available solution at each step, regardless of future consequences.
Prim’s Algorithm

• Basic idea:
  – Attach a new edge to a single growing tree at each step.
  – Start with any vertex as a single vertex tree, then add $V - 1$ edges to it.
  – Always take the remaining minimum weight edge that connects a vertex on the tree to a vertex not yet on the tree.
Prim’s Algorithm

LazyPrimMST(EdgeWeightedGraph G) {
    pq = new MinPQ<Edge>();
    marked = new boolean[G.V()];
    mst = new Queue<Edge>();

    visit(G, 0);
    while (!pq.isEmpty()) {
        Edge e = pq.delMin();
        int v = e.either();
        int w = e.other(v);
        if (marked(v) && marked(w)) continue;
        mst.enqueue(e);
        if (!marked(v)) visit(G, v);
        if (!marked(w)) visit(G, w);
    } //while
} //LazyPrimMST

void visit(EWG G, int v) {
    marked[v] = true;
    for (Edge e: G.adj(v))
        if (!marked(e.other(v))
            pq.insert(e);
} //visit
Prim’s Algorithm – Visual

* marks new entries

crossing edges (ordered by weight)

ineligible edges (gray)

0-7 0.16
0-2 0.26
0-4 0.38
6-0 0.58

2-3 0.17
5-7 0.28
1-3 0.29
1-5 0.32
2-7 0.34
1-2 0.36
4-7 0.37
0-4 0.38
6-2 0.40
6-0 0.58

1-3 0.29
1-5 0.32
2-7 0.34
4-5 0.35
1-2 0.36
4-7 0.37
0-4 0.38
6-2 0.40
3-6 0.52
6-0 0.58

1-2 0.36
4-7 0.37
0-4 0.38
6-2 0.40
3-6 0.52
6-0 0.58
6-4 0.93

04/09/2015
Minimum Spanning Trees
Prim’s Algorithm – Analysis

- **Theorem:** Lazy Prim’s uses space proportional to $E$ and time proportional to $E \times \log(E)$ (in the worst case).

- **Proof:**
  - For space, in the worst case, all edges will be in the MinPQ at the same time (vertex 0 is connected to every other vertex, which in turn are not connected to each other).
  - For time, in the worst case we need to consider every edge, and the MinPQ takes $\log(E)$ time for insertion and retrieval. ■
Eager Prim

• **Observation:** We don’t need ALL of those edges in our priority queue. We only need the fastest (current) way to access each of the non-tree vertices.
  
  – Let’s put vertices on the priority queue instead of edges, and update their weights as necessary.
Eager Prim

PrimMST(EdgeWeightedGraph G) {
    edgeTo = new Edge[G.V()];
    distTo = new double[G.V()];
    marked = new boolean[G.V()];
    for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;
    pq = new IndexMinPQ<Double>(G.V());
    distTo[0] = 0.0;
    pq.insert(0, 0.0);
    while (!pq.isEmpty())
        visit(G, pq.delMin());
} //PrimMST
Eager Prim

```java
void visit(EdgeWeightedGraph G, int v) {
    marked[v] = true;
    for (Edge e: G.adj(v)) {
        int w = e.other(v);
        if (marked[w])
            continue;
        if (e.weight() < distTo[w]) {
            edgeTo[w] = e;
            distTo[w] = e.weight();
            if (pq.contains(w))
                pq.changeKey(w, distTo[w]);
            else
                pq.insert(w, distTo[w]);
        } //if
    } //for
} //visit
```
Eager Prim – Visual

black: on MST  
red: on pq  
gray: not on MST  
thick red: smallest on pq, next to add to MST
Eager Prim – Analysis

• Theorem: Eager Prim uses extra space proportional to $V$ and extra time proportional to $E \times \log(V)$ (in the worst case).

• Proof:
  – The number of vertices in the priority queue is at most $V$, and there are 3 vertex-indexed arrays.
  – The IndexMinPQ implements operations in time $\log(V)$. ■
Kruskal’s Algorithm

- **Observation:** Instead of worrying about building a single tree vertex-by-vertex, let’s just consider the edges from smallest to largest.
  - Add each edge to the MST as long as it doesn’t form a cycle, and stop after $V - 1$ edges are added.
Kruskal’s Algorithm

KruskalMST(EdgeWeightedGraph G) {
    mst = new Queue<Edge>();
    MinPQ<Edge> pq = new MinPQ<Edge>();
    for (Edge e: G.edges()) {
        pq.insert(e);
        UF uf = new UF(G.V());
    }

    while (!pq.isEmpty() && mst.size() < G.V()-1) {
        Edge e = pq.delMin();
        int v = e.either(), w = e.other(v);
        if (uf.connected(v, w))
            continue;
        uf.union(v, w);
        mst.enqueue(e);
    }
}

4/9/2015

Minimum Spanning Trees
Kruskal’s Algorithm – Visual

- Graph edges sorted by weight
- **MST edge** (black)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td>0.16</td>
</tr>
<tr>
<td>2-3</td>
<td>0.17</td>
</tr>
<tr>
<td>1-7</td>
<td>0.19</td>
</tr>
<tr>
<td>0-2</td>
<td>0.26</td>
</tr>
<tr>
<td>5-7</td>
<td>0.28</td>
</tr>
<tr>
<td>1-3</td>
<td>0.29</td>
</tr>
<tr>
<td>1-5</td>
<td>0.32</td>
</tr>
<tr>
<td>2-7</td>
<td>0.34</td>
</tr>
<tr>
<td>4-5</td>
<td>0.35</td>
</tr>
<tr>
<td>1-2</td>
<td>0.36</td>
</tr>
<tr>
<td>4-7</td>
<td>0.37</td>
</tr>
<tr>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>6-2</td>
<td>0.40</td>
</tr>
<tr>
<td>3-6</td>
<td>0.52</td>
</tr>
<tr>
<td>6-0</td>
<td>0.58</td>
</tr>
<tr>
<td>6-4</td>
<td>0.93</td>
</tr>
</tbody>
</table>

- **Next MST edge** is red
- Grey vertices are a cut defined by the vertices connected to one of the red edge's vertices

- **Obsolete edge** (gray)
Kruskal’s Algorithm – Analysis

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

• **Proof:**
  – We still need to compare all edges.
  – We still have a log-time priority queue.
Can We Do Better?

• No theoretical results say that there *can’t* be a linear-time MST algorithm, but we certainly haven’t found one yet.
• But we’re getting close.
  – Chazelle introduced an algorithm in 1997 that “could never be distinguished from a linear time algorithm in any conceivable practical situation (even though it is provably nonlinear), but it is so complicated that no one would use it in practice.”
• Practically speaking, Prim and Kruskal are also effectively linear algorithms for sparse graphs in most practical situations.
• Also, Prim and Kruskal don’t work for directed graphs. This becomes a much more difficult problem called “minimum cost arborescence.”
Any Questions?