4.3 Minimum Spanning Trees

- introduction
- greedy algorithm
- edge-weighted graph API
- Kruskal's algorithm
- Prim's algorithm
- context

With added notes and slides by Betty O’Neil for cs310
4.3 Minimum Spanning Trees

- introduction
- greedy algorithm
- edge-weighted graph API
- Kruskal's algorithm
- Prim's algorithm
- context
Def. A spanning tree of $G$ is a subgraph $T$ that is:

- Connected.
- Acyclic.
- Includes all of the vertices.

Minimum spanning tree

graph $G$
Def. A spanning tree of $G$ is a subgraph $T$ that is:

- Connected.
- Acyclic.
- Includes all of the vertices.

Minimum spanning tree
Def. A spanning tree of $G$ is a subgraph $T$ that is:

- Connected.
- Acyclic.
- Includes all of the vertices.
**Def.** A spanning tree of $G$ is a subgraph $T$ that is:

- Connected.
- Acyclic.
- Includes all of the vertices.

Minimum spanning tree

![Diagram of a spanning tree and a non-spanning subgraph.]

*not spanning*
Given. Undirected graph $G$ with positive edge weights (connected).
Goal. Find a min weight spanning tree.
Minimum spanning tree

**Given.** Undirected graph $G$ with positive edge weights (connected).

**Goal.** Find a min weight spanning tree.

---

**Brute force.** Try all spanning trees?

---

**minimum spanning tree $T$**

(cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7)
Network design

MST of bicycle routes in North Seattle

http://www.flickr.com/photos/ewedistrict/21980840
Models of nature

MST of random graph

http://algo.inria.fr/broutin/gallery.html
MST describes arrangement of nuclei in the epithelium for cancer research

http://www.bccrc.ca/ci/ta01_archlevel.html
Image processing

MST dithering

http://www.flickr.com/photos/quasimondo/2695389651
Applications

MST is fundamental problem with diverse applications.

- Dithering.
- Cluster analysis.
- Max bottleneck paths.
- Real-time face verification.
- LDPC codes for error correction.
- Image registration with Renyi entropy.
- Find road networks in satellite and aerial imagery.
- Reducing data storage in sequencing amino acids in a protein.
- Model locality of particle interactions in turbulent fluid flows.
- Autoconfig protocol for Ethernet bridging to avoid cycles in a network.
- Approximation algorithms for NP-hard problems (e.g., TSP, Steiner tree).
- Network design (communication, electrical, hydraulic, computer, road).

4.3 Minimum Spanning Trees

- Introduction
- Greedy algorithm
- Edge-weighted graph API
- Kruskal’s algorithm
- Prim’s algorithm
- Context
Simplifying assumptions

- Graph is connected.
- Edge weights are distinct. See page 605 for discussion of this assumption.

**Consequence.** MST exists. It can be shown to be unique.
Def. A cut in a graph is a partition of its vertices into two (nonempty) sets.
Def. A crossing edge connects a vertex in one set with a vertex in the other.

Cut property. Given any cut, the crossing edge of min weight is in the MST.

This property leads to a greedy algorithm to find the MST: just do cuts, add min weight edge to growing MST. A rare use of “greedy” in S&W.
Cut property: correctness proof

Def. A cut in a graph is a partition of its vertices into two (nonempty) sets. Def. A crossing edge connects a vertex in one set with a vertex in the other.

Cut property. Given any cut, the crossing edge of min weight is in the MST.

Pf. Suppose min-weight crossing edge $e$ is not in the MST.

- Adding $e$ to the MST creates a cycle.
- Some other edge $f$ in cycle must be a crossing edge.
- Removing $f$ and adding $e$ is also a spanning tree.
- Since weight of $e$ is less than the weight of $f$, that spanning tree is lower weight.
- Contradiction. ▪
Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

an edge-weighted graph
Cut through middle-right has min edge weight 0.26, so edge 0-2 must be in MST
Could mark crossing edges red here.
Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until \( V - 1 \) edges are colored black.

![Graph with edge weights](image)

**an edge-weighted graph**

- Cut on left has min edge weight 0.35, so edge 4-5 must be in MST.
- For more steps, see [slideset at Princeton.edu](https://example.com)
Greedy MST algorithm demo result

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

MST edges

0-2 5-7 6-2 0-7 2-3 1-7 4-5
Proposition. The greedy algorithm computes the MST, for a connected graph.

Pf. Any edge colored black is in the MST (via cut property).

- Fewer than $V-1$ black edges $\Rightarrow$ cut with no black crossing edges.
  (consider cut whose vertices are any one connected component)

- Alternative argument to second bullet: It’s easy to show by induction that a spanning tree (minimal or not) has $V-1$ vertices. Thus if there are fewer than $V-1$ black edges, the graph is not spanning, so there are vertices that can’t be reached from other ones: use these to define a cut.
Proposition. The greedy algorithm computes the MST.

Efficient implementations. Choose cut? Find min-weight edge?
Ex 1. Kruskal's algorithm. [stay tuned]
Ex 2. Prim's algorithm. [stay tuned]
Ex 3. Boruvka's algorithm.

• In this case, the “greedy algorithm” is not itself the best one: we need further guidance in choosing the cuts to get good performance.
• All these specific algorithms can also be called “greedy”: they just are more specific about what cut to choose next.
Removing two simplifying assumptions

Q. What if edge weights are not all distinct?
A. Greedy MST algorithm still correct if equal weights are present!
   (our correctness proof fails, but that can be fixed)

Q. What if graph is not connected?
A. Compute minimum spanning forest = MST of each component.
Greed is good Gordon Gekko is a composite character in the 1987 film Wall Street and its 2010 sequel Wall Street: Money Never Sleeps, both directed by Oliver Stone. Gekko was portrayed by actor Michael Douglas, whose performance in the first film won him an Oscar for Best Actor.
4.3 Minimum Spanning Trees

- introduction
- greedy algorithm
- edge-weighted graph API
- Kruskal's algorithm
- Prim's algorithm
- context
Weighted edge API

Edge abstraction needed for weighted edges.

```java
class Edge implements Comparable<Edge> {
    Edge(int v, int w, double weight) {
        // create a weighted edge v-w
    }
    int either() {
        // either endpoint
    }
    int other(int v) {
        // the endpoint that's not v
    }
    int compareTo(Edge that) {
        // compare this edge to that edge
    }
    double weight() {
        // the weight
    }
    String toString() {
        // string representation
    }
}
```

Idiom for processing an edge e: `int v = e.either(), w = e.other(v);`
Weighted edge: Java implementation

```java
public class Edge implements Comparable<Edge>
{
    private final int v, w;
    private final double weight;

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

    public int either()
    {
        return v;
    }

    public int other(int vertex)
    {
        if (vertex == v) return w;
        else return v;
    }

    public int compareTo(Edge that)
    {
        if (this.weight < that.weight) return -1;
        else if (this.weight > that.weight) return +1;
        else return 0;
    }
}
```
# Edge-weighted graph API

**public class** `EdgeWeightedGraph`

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EdgeWeightedGraph(int V)</code></td>
<td>create an empty graph with V vertices</td>
</tr>
<tr>
<td><code>EdgeWeightedGraph(In in)</code></td>
<td>create a graph from input stream</td>
</tr>
<tr>
<td><code>void addEdge(Edge e)</code></td>
<td>add weighted edge e to this graph</td>
</tr>
<tr>
<td><code>Iterable&lt;Edge&gt; adj(int v)</code></td>
<td>edges incident to v</td>
</tr>
<tr>
<td><code>Iterable&lt;Edge&gt; edges()</code></td>
<td>all edges in this graph</td>
</tr>
<tr>
<td><code>int V()</code></td>
<td>number of vertices</td>
</tr>
<tr>
<td><code>int E()</code></td>
<td>number of edges</td>
</tr>
<tr>
<td><code>String toString()</code></td>
<td>string representation</td>
</tr>
</tbody>
</table>

**Conventions.** Allow self-loops and parallel edges.
Edge-weighted graph: adjacency-lists representation

Maintain vertex-indexed array of Edge lists.

```
adj[]
0 1 2 3 4 5 6 7
```

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>0 .58</td>
<td>0</td>
<td>2 .26</td>
<td>0</td>
<td>4 .38</td>
<td>0</td>
<td>7 .16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3 .29</td>
<td>1</td>
<td>2 .36</td>
<td>1</td>
<td>7 .19</td>
<td>1</td>
<td>5 .32</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2 .40</td>
<td>2</td>
<td>7 .34</td>
<td>1</td>
<td>2 .36</td>
<td>0</td>
<td>2 .26</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6 .52</td>
<td>1</td>
<td>3 .29</td>
<td>2</td>
<td>3 .17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>4 .93</td>
<td>0</td>
<td>4 .38</td>
<td>4</td>
<td>7 .37</td>
<td>4</td>
<td>5 .35</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>5 .32</td>
<td>5</td>
<td>7 .28</td>
<td>4</td>
<td>5 .35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>4 .93</td>
<td>6</td>
<td>0 .58</td>
<td>3</td>
<td>6 .52</td>
<td>6</td>
<td>2 .40</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>7 .34</td>
<td>1</td>
<td>7 .19</td>
<td>0</td>
<td>7 .16</td>
<td>5</td>
<td>7 .28</td>
</tr>
</tbody>
</table>
```

References to the same Edge object

Bag objects
public class EdgeWeightedGraph
{
    private final int V;
    private final Bag<Edge>[] adj;

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

    public void addEdge(Edge e)
    {
        int v = e.either(), w = e.other(v);
        adj[v].add(e);
        adj[w].add(e);
    }

    public Iterable<Edge> adj(int v)
    {
        return adj[v];
    }
}
Q. How to represent the MST?

public class MST

MST(EdgeWeightedGraph G) constructor

Iterable<Edge> edges() edges in MST

double weight() weight of MST

% java MST tinyEWG.txt
0-7 0.16
1-7 0.19
0-2 0.26
2-3 0.17
5-7 0.28
4-5 0.35
6-2 0.40
1.8
1

tinyEWG.txt
8
16
V
E
4 5 0.35
4 7 0.37
5 7 0.28
0 7 0.16
1 5 0.32
0 4 0.38
2 3 0.17
1 7 0.19
0 2 0.26
1 2 0.36
1 3 0.29
2 7 0.34
6 2 0.40
3 6 0.52
6 0 0.58
6 4 0.93

MST edge (black)

non-MST edge (gray)
Minimum spanning tree API: another S&W pseudo-interface+constructor

Q. How to represent the MST? (LazyPrimMST or PrimMST or KruskalMST)

```java
public class MST
{
  MST(EdgeWeightedGraph G)  // constructor
  Iterable<Edge> edges()    // edges in MST
  double weight()           // weight of MST
}
```

```java
public static void main(String[] args)
{
  In in = new In(args[0]);
  EdgeWeightedGraph G = new EdgeWeightedGraph(in);
  MST mst = new MST(G);
  for (Edge e : mst.edges())
    StdOut.println(e);
  StdOut.printf("%.2f\n", mst.weight());
}
```

% java MST tinyEWG.txt
0-7 0.16
1-7 0.19
0-2 0.26
2-3 0.17
5-7 0.28
4-5 0.35
6-2 0.40
1.8
1
4.3 Minimum Spanning Trees

- Introduction
- Greedy algorithm
- Edge-weighted graph API
- Kruskal's algorithm
- Prim's algorithm
- Context
Kruskal's algorithm demo

Consider edges in ascending order of weight.
- Add next edge to tree $T$ unless doing so would create a cycle.

Graph edges sorted by weight:

- 0–7 0.16
- 2–3 0.17
- 1–7 0.19
- 0–2 0.26
- 5–7 0.28
- 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
- 6–4 0.93
Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree $T$ unless doing so would create a cycle.

### Graph Edges Sorted by Weight

<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>

In the diagram, the edge 0-7 is highlighted, indicating it is part of the Minimum Spanning Tree (MST). The edge does not create a cycle.
Kruskal's algorithm

demo

Consider edges in ascending order of weight.

• Add next edge to tree $T$ unless doing so would create a cycle.

For more steps, see slideset at Princeton.edu
Consider edges in ascending order of weight.

- Add next edge to tree $T$ unless doing so would create a cycle.

```
<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>
```
Kruskal's algorithm: correctness proof

Proposition. [Kruskal 1956] Kruskal's algorithm computes the MST.

Pf. Kruskal's algorithm is a special case of the greedy MST algorithm.
- Suppose Kruskal's algorithm colors the edge $e = v–w$ black.
- Cut = set of vertices connected to $v$ in tree $T$
- No crossing edge is black.
- No crossing edge has lower weight. Why? (because we’ve already considered the lower-weight edges and either made them into black edges or they are connected two ways to them, so they can’t be a crossing edge)

![Diagram of a network with labeled vertices and edges, highlighting the process of adding an edge to the tree.]
Kruskal's algorithm: implementation challenge

**Challenge.** Would adding edge $v\rightarrow w$ to tree $T$ create a cycle? If not, add it.

**How difficult?**

- $E + V$
  - run DFS from $v$, check if $w$ is reachable
    (T has at most $V - 1$ edges)
- $V$
- $\log V$
- $\log* V$
  - use the union-find data structure!
- $1$

![diagram](image-url)
Kruskal's algorithm: implementation challenge

**Challenge.** Would adding edge \( v \rightarrow w \) to tree \( T \) create a cycle? If not, add it.

**Efficient solution.** Use the union-find data structure.
- Maintain a set for each connected component in \( T \).
- If \( v \) and \( w \) are in same set, then adding \( v \rightarrow w \) would create a cycle.
- To add \( v \rightarrow w \) to \( T \), merge sets containing \( v \) and \( w \).

Case 1: adding \( v \rightarrow w \) creates a cycle
Case 2: add \( v \rightarrow w \) to \( T \) and merge sets containing \( v \) and \( w \)
Kruskal's algorithm: Java implementation

```java
public class KruskalMST {
    private Queue<Edge> mst = new Queue<Edge>();

    public KruskalMST(EdgeWeightedGraph G) {
        MinPQ<Edge> pq = new MinPQ<Edge>(G.edges());
        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)) {
                uf.union(v, w);
                mst.enqueue(e);
            }
        }
    }

    public Iterable<Edge> edges() {
        return mst;
    }
}
```
Kruskal's algorithm: running time

**Proposition.** Kruskal's algorithm computes MST in time proportional to $E \log E$ (in the worst case).

**Pf.**

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>time per op</th>
</tr>
</thead>
<tbody>
<tr>
<td>build pq</td>
<td>1</td>
<td>$E$</td>
</tr>
<tr>
<td>delete-min</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
<tr>
<td>union</td>
<td>$V$</td>
<td>$\log^* V$†</td>
</tr>
<tr>
<td>connected</td>
<td>$E$</td>
<td>$\log^* V$†</td>
</tr>
</tbody>
</table>

† amortized bound using weighted quick union with path compression

Conclusion: it’s practical to run this on graphs with billions of edges

recall: $\log^* V \leq 5$ in this universe

**Remark.** If edges are already sorted, order of growth is $E \log^* V$. 

42
4.3 Minimum Spanning Trees

- Introduction
- Greedy algorithm
- Edge-weighted graph API
- Kruskal’s algorithm
- Prim’s algorithm
- Context
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V - 1$ edges.

an edge-weighted graph
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V - 1$ edges.
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V-1$ edges.

![Graph](image)

**MST edges**

0–7
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V-1$ edges.

MST edges

0–7

min weight edge with exactly one endpoint in $T$

edges with exactly one endpoint in $T$
(sorted by weight)

in MST

1–7 0.19
0–2 0.26
5–7 0.28
2–7 0.34
4–7 0.37
0–4 0.38
6–0 0.58
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V - 1$ edges.

**MST edges**

0–7  1–7

For more steps, see [slide set at Princeton.edu](http://slide.set.at.Princeton.edu)
Prim's algorithm demo

• Start with vertex 0 and greedily grow tree \( T \).
• Add to \( T \) the min weight edge with exactly one endpoint in \( T \).
• Repeat until \( V - 1 \) edges.

MST edges

0–7  1–7  0–2  2–3  5–7  4–5  6–2
Proposition. [Jarník 1930, Dijkstra 1957, Prim 1959]
Prim's algorithm computes the MST.

\textbf{Pf.} Prim's algorithm is a special case of the greedy MST algorithm.
\begin{itemize}
  \item Suppose edge \( e = \min \text{ weight edge connecting a vertex on the tree to a vertex not on the tree.} \)
  \item Cut = set of vertices connected on tree.
  \item No crossing edge is black.
  \item No crossing edge has lower weight.
\end{itemize}
**Challenge.** Find the min weight edge with exactly one endpoint in $T$.

**How difficult?**

- $E$ → try all edges
- $V$
- $\log E$ → use a priority queue!
- $\log^* E$
- $1$

1-7 is min weight edge with exactly one endpoint in $T$
Prim's algorithm: lazy implementation

**Challenge.** Find the min weight edge with exactly one endpoint in \( T \).

**Lazy solution.** Maintain a PQ of edges with (at least) one endpoint in \( T \).
- Key = edge; priority = weight of edge.
- Delete-min to determine next edge \( e = v-w \) to add to \( T \).
- Disregard if both endpoints \( v \) and \( w \) are marked (both in \( T \)).
- Otherwise, let \( w \) be the unmarked vertex (not in \( T \)):
  - add to PQ any edge incident to \( w \) (assuming other endpoint not in \( T \))
  - add \( e \) to \( T \) and mark \( w \)

1-7 is min weight edge with exactly one endpoint in \( T \).
Prim's algorithm (lazy) demo

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V-1$ edges.

![an edge-weighted graph]
Prim's algorithm: lazy implementation demo

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V - 1$ edges.

**add to PQ all edges incident to 0**

<table>
<thead>
<tr>
<th>edges on PQ (sorted by weight)</th>
</tr>
</thead>
<tbody>
<tr>
<td>* 0-7 0.16</td>
</tr>
<tr>
<td>* 0-2 0.26</td>
</tr>
<tr>
<td>* 0-4 0.38</td>
</tr>
<tr>
<td>* 6-0 0.58</td>
</tr>
</tbody>
</table>
Prim's algorithm: lazy implementation demo

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V - 1$ edges.

**delete 0-7 and add to MST**

**edges on PQ**

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td>0.16</td>
</tr>
<tr>
<td>0-2</td>
<td>0.26</td>
</tr>
<tr>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>6-0</td>
<td>0.58</td>
</tr>
</tbody>
</table>
Prim's algorithm: lazy implementation demo

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V - 1$ edges.

MST edges
0–7

edges on PQ (sorted by weight)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>0–4</td>
<td>0.38</td>
</tr>
<tr>
<td>6–0</td>
<td>0.58</td>
</tr>
</tbody>
</table>
Prim's algorithm: lazy implementation demo

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V - 1$ edges.

**add to PQ all edges incident to 7**

![Graph showing Prim's algorithm steps](image)

**MST edges**

0–7

<table>
<thead>
<tr>
<th>Edges on PQ (sorted by weight)</th>
</tr>
</thead>
<tbody>
<tr>
<td>* 1–7 0.19</td>
</tr>
<tr>
<td>0–2 0.26</td>
</tr>
<tr>
<td>* 5–7 0.28</td>
</tr>
<tr>
<td>* 2–7 0.34</td>
</tr>
<tr>
<td>* 4–7 0.37</td>
</tr>
<tr>
<td>0–4 0.38</td>
</tr>
<tr>
<td>6–0 0.58</td>
</tr>
</tbody>
</table>
Prim's algorithm: lazy implementation demo

- Start with vertex 0 and greedily grow tree \( T \).
- Add to \( T \) the min weight edge with exactly one endpoint in \( T \).
- Repeat until \( V - 1 \) edges.

**delete 1-7 and add to MST**

![Graph with edges and weights]

**MST edges**

0–7

<table>
<thead>
<tr>
<th>Edges on PQ (sorted by weight)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-7 0.19</td>
</tr>
<tr>
<td>0-2 0.26</td>
</tr>
<tr>
<td>5-7 0.28</td>
</tr>
<tr>
<td>2-7 0.34</td>
</tr>
<tr>
<td>4-7 0.37</td>
</tr>
<tr>
<td>0-4 0.38</td>
</tr>
<tr>
<td>6-0 0.58</td>
</tr>
</tbody>
</table>
Prim's algorithm: lazy implementation demo

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V - 1$ edges.

MST edges
0-7 1-7

For more steps, see slideset at Princeton.edu, middle part
Prim's algorithm (lazy) demo: final result

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V - 1$ edges.

**MST edges**

0–7 1–7 0–2 2–3 5–7 4–5 6–2
Prim's algorithm: lazy implementation

```java
public class LazyPrimMST {
    private boolean[] marked; // MST vertices
    private Queue<Edge> mst; // MST edges
    private MinPQ<Edge> pq; // PQ of edges

    public LazyPrimMST(WeightedGraph G) {
        pq = new MinPQ<Edge>();
        mst = new Queue<Edge>();
        marked = new boolean[G.V()];
        visit(G, 0);
    }

    while (!pq.isEmpty() && mst.size() < G.V() - 1) {
        Edge e = pq.delMin();
        int v = e.either(), 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);
    }
}
```

- Assume G is connected
- Repeatedly delete the min weight edge e = v–w from PQ
- Ignore if both endpoints in T
- Add edge e to tree
- Add v or w to tree
private void visit(WeightedGraph G, int v) {
    marked[v] = true;
    for (Edge e : G.adj(v))
        if (!marked[e.other(v)])
            pq.insert(e);
}

public Iterable<Edge> mst() {
    return mst;
}
**Proposition.** Lazy Prim's algorithm computes the MST in time proportional to $E \log E$ and extra space proportional to $E$ (in the worst case).

**Pf.**

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>binary heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete min</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
<tr>
<td>insert</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
</tbody>
</table>
Prim's algorithm: eager implementation: we’re skipping this (it’s only a little faster, and takes more memory)

**Challenge.** Find min weight edge with exactly one endpoint in $T$.

**Eager solution.** Maintain a PQ of *vertices* connected by an edge to $T$, where priority of vertex $v$ = weight of shortest edge connecting $v$ to $T$.

- Delete min vertex $v$ and add its associated edge $e = v\rightarrow w$ to $T$.
- Update PQ by considering all edges $e = v\rightarrow x$ incident to $v$
  - ignore if $x$ is already in $T$
  - add $x$ to PQ if not already on it
  - decrease priority of $x$ if $v\rightarrow x$ becomes shortest edge connecting $x$ to $T$
Prim's algorithm (eager) demo:

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V - 1$ edges.

an edge-weighted graph
Indexed priority queue

Associate an index between 0 and $N - 1$ with each key in a priority queue.

- Supports **insert** and **delete-the-minimum**.
- Supports **decrease-key** given the index of the key.

```java
public class IndexMinPQ<Key extends Comparable<Key>> {
    IndexMinPQ(int N) {
        // create indexed priority queue with indices 0, 1, ..., N - 1
    }

    void insert(int i, Key key) {
        // associate key with index i
    }

    void decreaseKey(int i, Key key) {
        // decrease the key associated with index i
    }

    boolean contains(int i) {
        // is i an index on the priority queue?
    }

    int delMin() {
        // remove a minimal key and return its associated index
    }

    boolean isEmpty() {
        // is the priority queue empty?
    }

    int size() {
        // number of keys in the priority queue
    }
}
```
Indexed priority queue implementation

**Binary heap implementation.** [see Section 2.4 of textbook]

- Start with same code as MinPQ.
- Maintain parallel arrays `keys[]`, `pq[]`, and `qp[]` so that:
  - `keys[i]` is the priority of `i`
  - `pq[i]` is the index of the key in heap position `i`
  - `qp[i]` is the heap position of the key with index `i`
- Use `swim(qp[i])` to implement `decreaseKey(i, key)`.

```
<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys[i]</td>
<td>A</td>
<td>S</td>
<td>0</td>
<td>R</td>
<td>T</td>
<td>I</td>
<td>N</td>
<td>G</td>
<td>-</td>
</tr>
<tr>
<td>pq[i]</td>
<td>-</td>
<td>0</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>qp[i]</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>-</td>
</tr>
</tbody>
</table>
```
Prim's algorithm: which priority queue?

Depends on PQ implementation: $V$ insert, $V$ delete-min, $E$ decrease-key.

<table>
<thead>
<tr>
<th>PQ implementation</th>
<th>insert</th>
<th>delete-min</th>
<th>decrease-key</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>$V$</td>
<td>1</td>
<td>$V^2$</td>
</tr>
<tr>
<td>binary heap</td>
<td>$\log V$</td>
<td>$\log V$</td>
<td>$\log V$</td>
<td>$E \log V$</td>
</tr>
<tr>
<td>d-way heap</td>
<td>$\log_d V$</td>
<td>$d \log_d V$</td>
<td>$\log_d V$</td>
<td>$E \log_{E/V} V$</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>$1 \dagger$</td>
<td>$\log V \dagger$</td>
<td>$1 \dagger$</td>
<td>$E + V \log V$</td>
</tr>
</tbody>
</table>

† amortized

Bottom line.
- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.
4.3 Minimum Spanning Trees

- Introduction
- Greedy algorithm
- Edge-weighted graph API
- Kruskal’s algorithm
- Prim’s algorithm
- Context
Does a linear-time MST algorithm exist?

## Deterministic Compare-Based MST Algorithms

<table>
<thead>
<tr>
<th>Year</th>
<th>Worst Case</th>
<th>Discovered By</th>
</tr>
</thead>
<tbody>
<tr>
<td>1975</td>
<td>$E \log \log V$</td>
<td>Yao</td>
</tr>
<tr>
<td>1976</td>
<td>$E \log V$</td>
<td>Cheriton-Tarjan</td>
</tr>
<tr>
<td>1984</td>
<td>$E \log* V, E + V \log V$</td>
<td>Fredman-Tarjan</td>
</tr>
<tr>
<td>1986</td>
<td>$E \log (\log* V)$</td>
<td>Gabow-Galil-Spencer-Tarjan</td>
</tr>
<tr>
<td>1997</td>
<td>$E \alpha(V) \log \alpha(V)$</td>
<td>Chazelle</td>
</tr>
<tr>
<td>2000</td>
<td>$E \alpha(V)$</td>
<td>Chazelle</td>
</tr>
<tr>
<td>2002</td>
<td>optimal</td>
<td>Pettie-Ramachandran</td>
</tr>
<tr>
<td>20xx</td>
<td>$E$</td>
<td>???</td>
</tr>
</tbody>
</table>

Remark. Linear-time randomized MST algorithm (Karger-Klein-Tarjan 1995).
Euclidean MST

Given $N$ points in the plane, find MST connecting them, where the distances between point pairs are their Euclidean distances.

Brute force. Compute $\sim N^2/2$ distances and run Prim's algorithm.
Ingenuity. Exploit geometry and do it in $\sim c N \log N$. 
Scientific application: clustering

**k-clustering.** Divide a set of objects classify into $k$ coherent groups.

**Distance function.** Numeric value specifying "closeness" of two objects.

**Goal.** Divide into clusters so that objects in different clusters are far apart.

Applications.

- Routing in mobile ad hoc networks.
- Document categorization for web search.
- Similarity searching in medical image databases.
- Skycat: cluster $10^9$ sky objects into stars, quasars, galaxies.

outbreak of cholera deaths in London in 1850s (Nina Mishra)
Single-link clustering

**k-clustering.** Divide a set of objects classify into \( k \) coherent groups.

**Distance function.** Numeric value specifying "closeness" of two objects.

**Single link.** Distance between two clusters equals the distance between the two closest objects (one in each cluster).

**Single-link clustering.** Given an integer \( k \), find a \( k \)-clustering that maximizes the distance between two closest clusters.
Single-link clustering algorithm

“Well-known” algorithm in science literature for single-link clustering:

- Form $V$ clusters of one object each.
- Find the closest pair of objects such that each object is in a different cluster, and merge the two clusters.
- Repeat until there are exactly $k$ clusters.

Observation. This is Kruskal's algorithm.
(stopping when $k$ connected components)

Alternate solution. Run Prim; then delete $k - 1$ max weight edges.
Tumors in similar tissues cluster together.

Reference: Botstein & Brown group

Dendrogram of cancers in human