4.3 Minimum Spanning Trees

- introduction
- greedy algorithm
- edge-weighted graph API
- Kruskal's algorithm
- Prim's algorithm
4.3 Minimum Spanning Trees

- introduction
- greedy algorithm
- edge-weighted graph API
- Kruskal's algorithm
- Prim's algorithm
**Spanning tree**

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

- A **tree**: connected and acyclic.
- **Spanning**: includes all of the vertices.
**Spanning tree**

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

- A tree: connected and acyclic.
- Spanning: includes all of the vertices.
**Def.** A **spanning tree** of $G$ is a subgraph $T$ that is:

- A tree: connected and acyclic.
- Spanning: includes all of the vertices.

![Spanning tree example](image)
**Spanning tree**

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

- **A tree:** connected and acyclic.
- **Spanning:** includes all of the vertices.

![Diagram of a graph with two spanning trees highlighted]
Minimum spanning tree problem

**Input.** Connected, undirected graph \( G \) with positive edge weights.

\[
\text{edge-weighted digraph } G
\]
Minimum spanning tree problem

Input. Connected, undirected graph $G$ with positive edge weights.
Output. A spanning tree of minimum weight.

Brute force. Try all spanning trees?
Models of nature

MST of random graph

http://algo.inria.fr/broutin/gallery.html
Medical image processing

MST describes arrangement of nuclei in the epithelium for cancer research

http://www.bccrc.ca/ci/ta01_archlevel.html
Slime mold grows network just like Tokyo rail system

Rules for Biologically Inspired Adaptive Network Design

Atsushi Tero, 1,2 Seiji Takagi, 1 Tetsu Saigusa, 1 Kentaro Ito, 1 Dan P. Bebb, 4 Mark D. Fricker, 4 Kenji Yumiki, 2,3 Ryo Kobayashi, 2,3 Toshiyuki Nakagaki 1,3,4

https://www.youtube.com/watch?v=GwKuFREOgmo
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
Simplifying assumptions

For simplicity, we assume

- The graph is connected. \( \Rightarrow \) MST exists.
- The edge weights are distinct. \( \Rightarrow \) MST is unique.

![Graph with edge weights](image_url)
Cut property

**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.
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 has 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.

![Graph Image]

an edge-weighted graph

<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>
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.
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.

MST edges

0–2
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.
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.

MST edges

0–2  5–7
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.

MST edges

0–2  5–7
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.

MST edges

0–2  5–7  6–2
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.
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.

MST edges

- 0-2
- 5-7
- 6-2
- 0-7
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.
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.

**MST edges**

0–2 5–7 6–2 0–7 2–3
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.

MST edges

0–2  5–7  6–2  0–7  2–3
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.

MST edges
0–2  5–7  6–2  0–7  2–3  1–7
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.

MST edges

0-2  5-7  6-2  0-7  2-3  1-7
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.

MST edges

0–2  5–7  6–2  0–7  2–3  1–7  4–5
Greedy MST algorithm: correctness proof

**Proposition.** The greedy algorithm computes the MST.

**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)

![Diagram showing a cut with no black crossing edges](image)

![Diagram showing fewer than \( V - 1 \) edges colored black](image)
Greedy MST algorithm: efficient implementations

**Proposition.** The greedy algorithm computes the MST.

**Efficient implementations.** Find cut? Find min-weight edge?

**Ex 1.** Kruskal's algorithm.  [stay tuned]

**Ex 2.** Prim's algorithm.  [stay tuned]

**Ex 3.** Borůvka's algorithm.
Removing two simplifying assumptions

Q. What if edge weights are not all distinct?
A. Greedy MST algorithm correct even 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.
Gordon Gecko (Michael Douglas) evangelizing the importance of greed (in algorithm design?)
Wall Street (1986)
4.3 Minimum Spanning Trees

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

Edge abstraction needed for weighted edges.

```java
public 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);`
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

```java
class EdgeWeightedGraph {
    public EdgeWeightedGraph(int V) {
        // create an empty graph with V vertices
    }
    public EdgeWeightedGraph(In in) {
        // create a graph from input stream
    }
    void addEdge(Edge e) {
        // add weighted edge e to this graph
    }
    Iterable<Edge> adj(int v) {
        // edges incident to v
    }
    Iterable<Edge> edges() {
        // all edges in this graph
    }
    int V() {
        // number of vertices
    }
    int E() {
        // number of edges
    }
    String toString() {
        // string representation
    }
}
```

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

Maintain vertex-indexed array of Edge lists.

```
adj[]
0 6 0.58 0 2.26 0 4.38 0 7.16
1 1 3.29 1 2.36 1 7.19 1 5.32
2 6 2.40 2 7.34 1 2.36 0 2.26 2 3.17
3 3 6.52 1 3.29 2 3.17
4 6 4.93 0 4.38 4 7.37 4 5.35
5 1 5.32 5 7.28 4 5.35
6 6 4.93 6 0.58 3 6.52 6 2.40
7 2 7.34 1 7.19 0 7.16 5 7.28 5 7.28
```

These references to the same Edge object should be handled carefully to avoid redundant computations and memory usage.
Edge-weighted graph: adjacency-lists implementation

```java
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];
    }
}
```

- **same as Graph, but adjacency lists of Edges instead of integers**
- **constructor**
- **add edge to both adjacency lists**
### Minimum spanning tree API

**Q. How to represent the MST?**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class MST</td>
<td></td>
</tr>
<tr>
<td>MST(EdgeWeightedGraph G)</td>
<td>constructor</td>
</tr>
<tr>
<td>Iterable&lt;Edge&gt; edges()</td>
<td>edges in MST</td>
</tr>
<tr>
<td>double weight()</td>
<td>weight of MST</td>
</tr>
</tbody>
</table>
4.3 MINIMUM SPANNING TREES

- introduction
- greedy algorithm
- edge-weighted graph API
- Kruskal's algorithm
- Prim's algorithm
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](image)

<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>

an edge-weighted graph
Kruskal's algorithm demo

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

```
    1
   / \  \\
  7---3---5
       |
      2
    /\  |
   0---4---6
```

in MST → 0-7 0.16
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.
Kruskal's algorithm demo

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

---

0-7 0.16
2-3 0.17
1-7 0.19

---

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.
Consider edges in ascending order of weight.

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

- Add next edge to tree $T$ unless doing so would 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.
Kruskal's algorithm demo

Consider edges in ascending order of weight.

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

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
Kruskal's algorithm demo

Consider edges in ascending order of weight.

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

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td>0.16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-3</td>
<td>0.17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-7</td>
<td>0.19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0-2</td>
<td>0.26</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5-7</td>
<td>0.28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-3</td>
<td>0.29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-5</td>
<td>0.32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-7</td>
<td>0.34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-5</td>
<td>0.35</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-2</td>
<td>0.36</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-7</td>
<td>0.37</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0-4</td>
<td>0.38</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6-2</td>
<td>0.40</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Kruskal's algorithm demo

Consider edges in ascending order of weight.
- Add next edge to tree $T$ unless doing so would 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.
Kruskal's algorithm demo

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

![Graph with labeled edges and weights]

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

a minimum spanning tree
Kruskal's algorithm: visualization
**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 $v$ via black edges in tree $T$.
- No crossing edge is black.
- No crossing edge has lower weight. Why?
Consider edges in ascending order of weight.

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

![A minimum spanning tree](image)
1.5 Union-Find

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
Dynamic connectivity problem

Given a set of N objects, support two operations:

- Connect two objects.
- Is there a path connecting the two objects?

- connect 4 and 3
- connect 3 and 8
- connect 6 and 5
- connect 9 and 4
- connect 2 and 1

- are 0 and 7 connected? ✗
- are 8 and 9 connected? ✓
- connect 5 and 0
- connect 7 and 2
- connect 6 and 1
- connect 1 and 0
- are 0 and 7 connected? ✓
A larger connectivity example

Q. Is there a path connecting $p$ and $q$?

A. Yes.
Modeling the connections

We assume "is connected to" is an equivalence relation:

- Reflexive: $p$ is connected to $p$.
- Symmetric: if $p$ is connected to $q$, then $q$ is connected to $p$.
- Transitive: if $p$ is connected to $q$ and $q$ is connected to $r$, then $p$ is connected to $r$.

Connected component. Maximal set of objects that are mutually connected.

\[
\begin{align*}
\{0\} & \quad \{1, 4, 5\} & \quad \{2, 3, 6, 7\}
\end{align*}
\]

3 connected components
Implementing the operations

**Find.** In which component is object \( p \)?

**Connected.** Are objects \( p \) and \( q \) in the same component?

**Union.** Replace components containing objects \( p \) and \( q \) with their union.

---

After the `union(2, 5)` operation:

- **Before:**
  - 3 connected components: \{0\}, \{1, 4, 5\}, \{2, 3, 6, 7\}

- **After:**
  - 2 connected components: \{0\}, \{1, 2, 3, 4, 5, 6, 7\}
**Union-find data type (API)**

**Goal.** Design efficient data structure for union-find.
- Number of objects $N$ can be huge.
- Number of operations $M$ can be huge.
- Union and find operations may be intermixed.

```java
public class UF {
    UF(int N) {
        // initialize union-find data structure with N singleton objects (0 to N – 1)
    }

    void union(int p, int q) {
        // add connection between p and q
    }

    int find(int p) {
        // component identifier for p (0 to N – 1)
    }

    boolean connected(int p, int q) {
        // are p and q in the same component?
    }
}
```

1-line implementation of `connected()`:
```java
public boolean connected(int p, int q) {
    return find(p) == find(q);
}
```
1.5 Union-Find

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
Quick-find  [eager approach]

Data structure.

- Integer array `id[]` of length `N`.
- Interpretation: `id[p]` is the id of the component containing `p`.

```plaintext
0 1 1 8 8 0 0 1 8 8
```

- `0`, `5`, and `6` are connected
- `1`, `2`, and `7` are connected
- `3`, `4`, `8`, and `9` are connected
Data structure.

- Integer array \( \text{id}[] \) of length \( N \).
- Interpretation: \( \text{id}[p] \) is the id of the component containing \( p \).

Find. What is the id of \( p \)?

Connected. Do \( p \) and \( q \) have the same id?

Union. To merge components containing \( p \) and \( q \), change all entries whose id equals \( \text{id}[p] \) to \( \text{id}[q] \).

Problem: many values can change.
Quick-find demo

id[]

<table>
<thead>
<tr>
<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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Quick-find demo

union(4, 3)

0 1 2 3 4 5 6 7 8 9

id[]

0 1 2 3 3 5 6 7 8 9

↑   ↑
Quick-find demo

union(3, 8)

id[]

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
0 & 1 & 2 & 8 & 8 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]
Quick-find demo

union(6, 5)

id[]

0 1 2 3 4 5 6 7 8 9

0 1 2 8 8 5 5 7 8 9
Quick-find demo

union(9, 4)

![Diagram of Quick-find algorithm]

```
p0 = 0, p1 = 1, p2 = 2, p3 = 3, p4 = 4, p5 = 5, p6 = 6, p7 = 7, p8 = 8, p9 = 9
id[] = [0, 1, 2, 8, 8, 5, 5, 7, 8, 8]
```
Quick-find demo

union(2, 1)
Quick-find demo

connected(8, 9)

0 1 2 3 4 5 6 7 8 9

id[]

0 1 1 8 8 5 5 7 8 8

already connected
Quick-find demo

connected(5, 0)

<table>
<thead>
<tr>
<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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>id[]</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Quick-find demo

union(5, 0)
**Quick-find demo**

union(7, 2)

![Diagram showing the quick-find algorithm with an example union operation.]
Quick-find demo

union(6, 1)
Quick-find demo

```
<table>
<thead>
<tr>
<th>id[]</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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
```
Quick-find: Java implementation

```java
public class QuickFindUF {
    private int[] id;

    public QuickFindUF(int N) {
        for (int i = 0; i < N; i++) {
            id[i] = i;
        }
    }

    public boolean find(int p) {
        return id[p] == p;
    }

    public void union(int p, int q) {
        int pid = find(p);
        int qid = find(q);
        for (int i = 0; i < id.length; i++) {
            if (id[i] == pid) {
                id[i] = qid;
            }
        }
    }
}
```

- set id of each object to itself (N array accesses)
- return the id of p (1 array access)
- change all entries with id[p] to id[q] (at most 2N + 2 array accesses)
public class QuickFindUF
{
    private int[] id;

    public QuickFindUF(int N)
    {
        id = new int[N];
        for (int i = 0; i < N; i++)
            id[i] = i;
    }

    public boolean find(int p)
    { return id[p]; }

    public void union(int p, int q)
    {
        int pid = id[p];
        int qid = id[q];
        for (int i = 0; i < id.length; i++)
            if (id[i] == pid) id[i] = qid;
    }
}
Quick-find is too slow

Cost model. Number of array accesses (for read or write).

<table>
<thead>
<tr>
<th>algorithm</th>
<th>initialize</th>
<th>union</th>
<th>find</th>
<th>connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

order of growth of number of array accesses

Union is too expensive. It takes $N^2$ array accesses to process a sequence of $N$ union operations on $N$ objects.
Quadratic algorithms do not scale

Rough standard (for now).
- $10^9$ operations per second.
- $10^9$ words of main memory.
- Touch all parts of memory in approx. 1 second.

Ex. Huge problem for quick-find.
- $10^9$ union commands on $10^9$ objects.
- Quick-find takes more than $10^{18}$ operations.
- 30+ years of computer time!

Quadratic algorithms don't scale with technology.
- New computer may be 10x as fast.
- But, has 10x as much memory $\Rightarrow$
  want to solve a problem that is 10x as big.
- With quadratic algorithm, takes 10x as long!
1.5 Union-Find

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
Quick-union  [lazy approach]

Data structure.

- Integer array \( \text{id}[\] \) of length \( N \).
- Interpretation: \( \text{id}[i] \) is parent of \( i \).
- Root of \( i \) is \( \text{id}[\text{id}[\text{id}[\ldots \text{id}[i] \ldots]]] \).

<table>
<thead>
<tr>
<th>( \text{id}[] )</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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{id}[]</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

keep going until it doesn't change (algorithm ensures no cycles)

parent of 3 is 4
root of 3 is 9
**Quick-union  [lazy approach]**

**Data structure.**
- Integer array `id[]` of length N.
- Interpretation: `id[i]` is parent of i.
- Root of i is `id[id[id[...id[i]...]]]`.

<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>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>id[]</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

**Find.** What is the root of p?

**Connected.** Do p and q have the same root?

**Union.** To merge components containing p and q, set the id of p's root to the id of q's root.

<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>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>id[]</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

---

root of 3 is 9  
root of 5 is 6  
3 and 5 are not connected
Quick-union demo
Quick-union demo

union(4, 3)

id[]

0 1 2 3 4 5 6 7 8 9
Quick-union demo

union(4, 3)

id[]

<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>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>id[]</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Quick-union demo

0 1 2 3 5 6 7 8 9

id[] 0 1 2 3 3 5 6 7 8 9
Quick-union demo

union(3, 8)
Quick-union demo

union(3, 8)

0 1 2 3 4 5 6 7 8 9

id[]

0 1 2 8 3 5 6 7 8 9
Quick-union demo

id[]

<table>
<thead>
<tr>
<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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Quick-union demo

union(6, 5)

id[]

<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>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Quick-union demo

union(6, 5)

id[]

0 1 2 3 4 5 6 7 8 9

0 1 2 8 3 5 7 8 9
Quick-union demo

id[]

0 1 2 3 4 5 6 7 8 9
0 1 2 8 3 5 5 7 8 9
Quick-union demo

union(9, 4)

<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>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>id[]</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Quick-union demo

union(9, 4)

id[]

0  1  2  3  4  5  6  7  8  9

0  1  2  8  3  5  5  7  8  8
Quick-union demo
Quick-union demo

union(2, 1)

id[]

<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>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Quick-union demo

union(2, 1)

0  1  2
  |  |
  \|/
1  5  7  8  3  4  6  9

id[]

0 1 1 1 8 3 5 5 7 8 8
Quick-union demo

id[]  0 1 1 8 3 5 5 7 8 8

Diagram: A network of nodes connected by edges, showing the structure of the quick-union algorithm.
Quick-union demo

connected(8, 9)

```
0 1 2 3 4 5 6 7 8 9
id[] 0 1 1 8 3 5 5 7 8 8
```
Quick-union demo

connected(5, 4)
Quick-union demo

union(5, 0)

id[]

\[
\begin{array}{cccccccccccc}
0 & 1 & 1 & 8 & 3 & 5 & 5 & 7 & 8 & 8 \\
\end{array}
\]
Quick-union demo

union(5, 0)

id[]

\[
\begin{array}{cccccccccc}
0 & 1 & 1 & 8 & 3 & 0 & 5 & 7 & 8 & 8 \\
\end{array}
\]
Quick-union demo

![Diagram of quick-union data structure]

<table>
<thead>
<tr>
<th>id[]</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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Quick-union demo

union(7, 2)

0 ─ 5 ─ 6
1 ─ 2
7
8 ─ 3 ─ 9

id[]:

0 1 1 8 3 0 5 7 8 8
Quick-union demo

union(7, 2)
Quick-union demo

id[]

0 1 1 8 3 0 5 1 8 8
Quick-union demo

union(6, 1)

id[]

0 1 1 8 3 0 5 1 8 8
Quick-union demo

union(6, 1)
Quick-union demo
Quick-union demo

union(7, 3)

union(7, 3)
Quick-union demo

union(7, 3)
Quick-union demo

id[]  0 1 2 3 4 5 6 7 8 9
      1 8 1 8 3 0 5 1 8 8
public class QuickUnionUF {
    private int[] id;

    public QuickUnionUF(int N) {
        // set id of each object to itself
        // (N array accesses)
        for (int i = 0; i < N; i++) {
            id[i] = i;
        }
    }

    public int find(int i) {
        // chase parent pointers until reach root
        // (depth of i array accesses)
        while (id[i] != i) {
            i = id[i];
        }
        return i;
    }

    public void union(int p, int q) {
        // change root of p to point to root of q
        // (depth of p and q array accesses)
        int rootP = find(p);
        int rootQ = find(q);
        id[rootP] = rootQ;
    }
}
public class QuickUnionUF
{
    private int[] id;

    public QuickUnionUF(int N)
    {
        id = new int[N];
        for (int i = 0; i < N; i++) id[i] = i;
    }

    public int find(int i)
    {
        while (i != id[i]) i = id[i];
        return i;
    }

    public void union(int p, int q)
    {
        int i = find(p);
        int j = find(q);
        id[i] = j;
    }
}
Quick-union is also too slow

Cost model. Number of array accesses (for read or write).

<table>
<thead>
<tr>
<th></th>
<th>initialize</th>
<th>union</th>
<th>find</th>
<th>connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>quick-union</td>
<td>N</td>
<td>N †</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

† includes cost of finding roots

Quick-find defect.
- Union too expensive ($N$ array accesses).
- Trees are flat, but too expensive to keep them flat.

Quick-union defect.
- Trees can get tall.
- Find/connected too expensive (could be $N$ array accesses).
1.5 Union-Find

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
**Improvement 1: weighting**

**Weighted quick-union.**

- Modify quick-union to avoid tall trees.
- Keep track of size of each tree (number of objects).
- Balance by linking root of smaller tree to root of larger tree.

![Diagram of quick-union and weighted quick-union trees]

**Reasonable alternatives:**
- Union by height or "rank"
Weighted quick-union demo
Weighted quick-union demo
Weighted quick-union demo

union(4, 3)
Weighted quick-union demo

union(4, 3)

0 1 2 3 4 5 6 7 8 9

id[]

0 1 2 4 4 5 6 7 8 9
Weighted quick-union demo
Weighted quick-union demo

union(3, 8)

id[]

0 1 2 3 4 5 6 7 8 9
Weighted quick-union demo

union(3, 8)

weighting: make 8 point to 4 (instead of 4 to 8)

id[]

0 1 2 4 4 5 6 7 4 9
Weighted quick-union demo

```
id[]  0  1  2  3  4  5  6  7  8  9
  0  1  2  4  4  5  6  7  4  9
```
Weighted quick-union demo

union(6, 5)

0 1 2 4 5 6 7 8 9

id[] 0 1 2 4 4 5 6 7 4 9
Weighted quick-union demo

union(6, 5)

id[]

0 1 2 3 4 5 6 7 8 9

0 1 2 4 4 6 6 7 4 9
Weighted quick-union demo

```
id[]   0  1  2  3  4  5  6  7  8  9
       0 1 2 4 4 6 6 7 4 9
```
Weighted quick-union demo

union(9, 4)
Weighted quick-union demo

union(9, 4)

weighting: make 9 point to 4
Weighted quick-union demo
weighted quick-union demo

union(2, 1)

id[]

0 1 2 3 4 5 6 7 8 9

0 1 2 4 4 6 6 7 4 4
Weighted quick-union demo

union(2, 1)

id[]

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
0 & 2 & 2 & 4 & 4 & 6 & 6 & 7 & 4 & 4 \\
\end{array}
\]
Weighted quick-union demo
Weighted quick-union demo

union(5, 0)

id[] = [0, 2, 2, 4, 4, 6, 6, 7, 4, 4]
Weighted quick-union demo

union(5, 0)

weighting: make 0 point to 6 (instead of 6 to 0)
Weighted quick-union demo

id[]

0 1 2 3 4 5 6 7 8 9
6 2 2 4 4 6 6 7 4 4
union(7, 2)
Weighted quick-union demo

union(7, 2)

weighting: make 7 point to 2
Weighted quick-union demo

id[] = [6, 2, 2, 4, 4, 4, 6, 6, 2, 4, 4]
Weighted quick-union demo

union(6, 1)

```
  2
 /|
1 7

  4
 /|
3 8 9

  6
 /|
0 5

id[]  0  1  2  3  4  5  6  7  8  9
     6  2  2  4  4  6  6  2  4  4
```
Weighted quick-union demo

union(6, 1)

![Weighted quick-union demo](image-url)
Weighted quick-union demo
Weighted quick-union demo

union(7, 3)

id[]

<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>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>6</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Weighted quick-union demo

union(7, 3)

weighting: make 4 point to 6 (instead of 6 to 4)
Weighted quick-union demo
Example trees in quick-union versus weighted quick-union

Quick-union and weighted quick-union (100 sites, 88 `union()` operations)

average distance to root: 5.11

average distance to root: 1.52
Weighted quick-union: Java implementation

**Data structure.** Same as quick-union, but maintain extra array \( \text{sz}[i] \) to count number of objects in the tree rooted at \( i \).

**Find/connected.** Identical to quick-union.

**Union.** Modify quick-union to:
- Link root of smaller tree to root of larger tree.
- Update the \( \text{sz}[] \) array.

```java
int i = find(p);
int j = find(q);
if (i == j) return;
if (sz[i] < sz[j]) {
    id[i] = j;
    sz[j] += sz[i];
}
else {
    id[j] = i;
    sz[i] += sz[j];
}
```
Weighted quick-union analysis

Running time.

- Find: takes time proportional to depth of $p$.
- Union: takes constant time, given roots.

Proposition. Depth of any node $x$ is at most $\lg N$. 

N = 11
depth(x) = 3 $\leq$ $\lg N$
Weighted quick-union analysis

Running time.
- Find: takes time proportional to depth of \( p \).
- Union: takes constant time, given roots.

Proposition. Depth of any node \( x \) is at most \( \lg N \).

Pf. What causes the depth of object \( x \) to increase?
Increases by 1 when tree \( T_1 \) containing \( x \) is merged into another tree \( T_2 \).
- The size of the new tree containing \( x \) at least doubles since \( |T_2| \geq |T_1| \).
- Size of tree containing \( x \) can double at most \( \lg N \) times. Why?

\[
\begin{array}{c}
T_2 \\
\downarrow \\
T_1 \\
\downarrow \\
\text{x}
\end{array}
\]

\[
|1| = 2^0, |2| = 2^1, |4| = 2^2, |8| = 2^3, |16| = 2^4, \ldots, |N| = 2^{\lg N}
\]

\( \lg = \text{base-2 logarithm} \)
Weighted quick-union analysis

Running time.
- Find: takes time proportional to depth of \( p \).
- Union: takes constant time, given roots.

Proposition. Depth of any node \( x \) is at most \( \lg N \).

<table>
<thead>
<tr>
<th>algorithm</th>
<th>initialize</th>
<th>union</th>
<th>find</th>
<th>connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>quick-union</td>
<td>N</td>
<td>N ( \dagger )</td>
<td>N</td>
<td>N ( \dagger )</td>
</tr>
<tr>
<td>weighted QU</td>
<td>N</td>
<td>( \lg N ) ( \dagger )</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
</tr>
</tbody>
</table>

\( \dagger \) includes cost of finding roots

Q. Stop at guaranteed acceptable performance?
A. No, easy to improve further.
Improvement 2: path compression

Quick union with path compression. Just after computing the root of $p$, set the $id[]$ of each examined node to point to that root.
Improvement 2: path compression

Quick union with path compression. Just after computing the root of \( p \), set the \( \text{id}[] \) of each examined node to point to that root.
Improvement 2: path compression

Quick union with path compression. Just after computing the root of \( p \), set the \( \text{id}[] \) of each examined node to point to that root.
Improvement 2: path compression

Quick union with path compression. Just after computing the root of $p$, set the $id[]$ of each examined node to point to that root.
Improvement 2: path compression

Quick union with path compression. Just after computing the root of \( p \), set the \( \text{id}[] \) of each examined node to point to that root.

Bottom line. Now, \( \text{find()} \) has the side effect of compressing the tree.
Path compression: Java implementation

**Two-pass implementation:** add second loop to `find()` to set the `id[]` of each examined node to the root.

**Simpler one-pass variant (path halving):** Make every other node in path point to its grandparent.

```java
public int find(int i) {
    while (i != id[i]) {
        id[i] = id[id[i]]; // only one extra line of code!
        i = id[i];
    }
    return i;
}
```

**In practice.** No reason not to! Keeps tree almost completely flat.
Weighted quick-union with path compression: amortized analysis

**Proposition.** [Hopcroft-Ulman, Tarjan] Starting from an empty data structure, any sequence of $M$ union–find ops on $N$ objects makes $\leq c(N + M \lg^* N)$ array accesses.

- Analysis can be improved to $N + M \alpha(M, N)$.
- Simple algorithm with fascinating mathematics.

Linear-time algorithm for $M$ union-find ops on $N$ objects?

- Cost within constant factor of reading in the data.
- In theory, WQUPC is not quite linear.
- In practice, WQUPC is linear.

**Amazing fact.** [Fredman-Saks] No linear-time algorithm exists.
Summary

**Key point.** Weighted quick union (and/or path compression) makes it possible to solve problems that could not otherwise be addressed.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>worst-case time</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>M N</td>
</tr>
<tr>
<td>quick-union</td>
<td>M N</td>
</tr>
<tr>
<td>weighted QU</td>
<td>N + M log N</td>
</tr>
<tr>
<td>QU + path compression</td>
<td>N + M log N</td>
</tr>
<tr>
<td>weighted QU + path compression</td>
<td>N + M ( \lg^* N )</td>
</tr>
</tbody>
</table>

order of growth for \( M \) union-find operations on a set of \( N \) objects

**Ex.** [\( 10^9 \) unions and finds with \( 10^9 \) objects]
- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.
Kruskal's algorithm: implementation challenge

**Challenge.** Would adding edge $v$–$w$ 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 $v$–$w$ would create a cycle.
- To add $v$–$w$ to $T$, merge sets containing $v$ and $w$.

---

**Case 1:** adding $v$–$w$ creates a cycle

**Case 2:** add $v$–$w$ 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;
    }
}
```

- build priority queue (or sort)
- greedily add edges to MST
- edge v–w does not create cycle
- merge connected components
- add edge e to MST
**Kruskal's algorithm: running time**

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

often called fewer than $E$ times
4.3 Minimum Spanning Trees

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

![Diagram of Prim's algorithm]

- Min weight edge with exactly one endpoint in $T$.
- Edges with exactly one endpoint in $T$ (sorted by weight):
  - In MST: 0–7 0.16
  - 0–2 0.26
  - 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
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)**

<table>
<thead>
<tr>
<th>Pair</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<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>2–7</td>
<td>0.34</td>
</tr>
<tr>
<td>4–7</td>
<td>0.37</td>
</tr>
<tr>
<td>0–4</td>
<td>0.38</td>
</tr>
<tr>
<td>6–0</td>
<td>0.58</td>
</tr>
</tbody>
</table>
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
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

**edges with exactly one endpoint in \( T \) (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>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>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>6–0</td>
<td>0.58</td>
</tr>
</tbody>
</table>
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
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

$0-7$  $1-7$  $0-2$
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
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
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
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
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
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

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

- 6-2 0.40
- 3-6 0.52
- 6-0 0.58
- 6-4 0.93
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
Prim’s algorithm: visualization
Kruskal's algorithm: visualization
**Proposition.** [Jarník 1930, Dijkstra 1957, Prim 1959]

Prim's algorithm computes the MST.

**Pf.** Prim's algorithm is a special case of the greedy MST algorithm.

- Suppose edge $e = \text{min weight edge connecting a vertex on the tree to a vertex not on the tree}$. 
- Cut = set of vertices connected on tree.
- No crossing edge is black.
- No crossing edge has lower weight.

![Diagram showing edge e = 7-5 added to tree]
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 $e = v \rightarrow w$ 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 $e$ to $T$ and mark $w$
  - add to PQ any edge incident to $w$ (assuming other endpoint not in $T$)

1-7 is min weight edge with exactly one endpoint in $T$

Priority queue of crossing edges

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<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>2-7</td>
<td>0.34</td>
</tr>
<tr>
<td>4-7</td>
<td>0.37</td>
</tr>
<tr>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>6-0</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.

an edge-weighted graph

<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>
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.

![Diagram of an edge-weighted graph]

<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>

*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.
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
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

<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.

MST edges

0–7
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

MST edges

<table>
<thead>
<tr>
<th></th>
<th>0–7</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>edges on PQ</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(sorted by weight)</td>
<td></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>* 2–7</td>
<td>0.34</td>
</tr>
<tr>
<td>* 4–7</td>
<td>0.37</td>
</tr>
<tr>
<td>0–4</td>
<td>0.38</td>
</tr>
<tr>
<td>6–0</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.

delete 1–7 and add to MST

MST edges

0–7

edges on PQ (sorted by weight)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<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>2–7</td>
<td>0.34</td>
</tr>
<tr>
<td>4–7</td>
<td>0.37</td>
</tr>
<tr>
<td>0–4</td>
<td>0.38</td>
</tr>
<tr>
<td>6–0</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  1–7
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 1

MST edges

0-7 1-7
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 edge 0–2 and add to MST

MST edges

0–7  1–7

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>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>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>6–0</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  1-7  0-2
```

```
edges on PQ (sorted by weight)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<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>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>6-0</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 2**

**edges on PQ**

(sorted by weight)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>2-3</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td>5-7</td>
<td>0.28</td>
</tr>
<tr>
<td></td>
<td>1-3</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>1-5</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td>2-7</td>
<td>0.34</td>
</tr>
<tr>
<td></td>
<td>1-2</td>
<td>0.36</td>
</tr>
<tr>
<td></td>
<td>4-7</td>
<td>0.37</td>
</tr>
<tr>
<td></td>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>*</td>
<td>6-2</td>
<td>0.40</td>
</tr>
<tr>
<td></td>
<td>6-0</td>
<td>0.58</td>
</tr>
</tbody>
</table>

MST edges

0-7  1-7  0-2
**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 2–3 and add to MST**

**edges on PQ (sorted by weight)**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2–3</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td></td>
<td>1–3</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>1–5</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td>2–7</td>
<td>0.34</td>
</tr>
<tr>
<td></td>
<td>1–2</td>
<td>0.36</td>
</tr>
<tr>
<td></td>
<td>4–7</td>
<td>0.37</td>
</tr>
<tr>
<td></td>
<td>0–4</td>
<td>0.38</td>
</tr>
<tr>
<td>*</td>
<td>6–2</td>
<td>0.40</td>
</tr>
<tr>
<td>*</td>
<td>6–0</td>
<td>0.58</td>
</tr>
</tbody>
</table>

**MST edges**

- 0–7
- 1–7
- 0–2
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  0–2  2–3

Edges on PQ (sorted by weight)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<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>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>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 3**

**MST edges**

| 0–7 | 1–7 | 0–2 | 2–3 |

**edges on PQ (sorted by weight)**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<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>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>
</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 5–7 and add to MST

MST edges

0–7 1–7 0–2 2–3
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  0–2  2–3  5–7

**edges on PQ (sorted by weight)**

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<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>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>
</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 5

```
<table>
<thead>
<tr>
<th>edges on PQ (sorted by weight)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-3 0.29</td>
</tr>
<tr>
<td>1-5 0.32</td>
</tr>
<tr>
<td>2-7 0.34</td>
</tr>
<tr>
<td>*4-5 0.35</td>
</tr>
<tr>
<td>1-2 0.36</td>
</tr>
<tr>
<td>4-7 0.37</td>
</tr>
<tr>
<td>0-4 0.38</td>
</tr>
<tr>
<td>6-2 0.40</td>
</tr>
<tr>
<td>3-6 0.52</td>
</tr>
<tr>
<td>6-0 0.58</td>
</tr>
</tbody>
</table>
```

**MST edges**

- 0-7
- 1-7
- 0-2
- 2-3
- 5-7

Diagram of the graph with edges and weights, showing the process of Prim's algorithm.
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 1–3 and discard obsolete edge**

**edges on PQ**

<table>
<thead>
<tr>
<th>(sorted by weight)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–3</td>
</tr>
<tr>
<td>1–5</td>
</tr>
<tr>
<td>2–7</td>
</tr>
<tr>
<td>4–5</td>
</tr>
<tr>
<td>1–2</td>
</tr>
<tr>
<td>4–7</td>
</tr>
<tr>
<td>0–4</td>
</tr>
<tr>
<td>6–2</td>
</tr>
<tr>
<td>3–6</td>
</tr>
<tr>
<td>6–0</td>
</tr>
</tbody>
</table>

**MST edges**

- 0–7
- 1–7
- 0–2
- 2–3
- 5–7
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 1–5 and discard obsolete edge

MST edges

0–7  1–7  0–2  2–3  5–7

<table>
<thead>
<tr>
<th>Edges on PQ (sorted by weight)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–5</td>
</tr>
<tr>
<td>2–7</td>
</tr>
<tr>
<td>4–5</td>
</tr>
<tr>
<td>1–2</td>
</tr>
<tr>
<td>4–7</td>
</tr>
<tr>
<td>0–4</td>
</tr>
<tr>
<td>6–2</td>
</tr>
<tr>
<td>3–6</td>
</tr>
<tr>
<td>6–0</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 2–7 and discard obsolete edge

**MST edges**

0–7 1–7 0–2 2–3 5–7

<table>
<thead>
<tr>
<th>edges on PQ (sorted by weight)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2–7 0.34</td>
</tr>
<tr>
<td>4–5 0.35</td>
</tr>
<tr>
<td>1–2 0.36</td>
</tr>
<tr>
<td>4–7 0.37</td>
</tr>
<tr>
<td>0–4 0.38</td>
</tr>
<tr>
<td>6–2 0.40</td>
</tr>
<tr>
<td>3–6 0.52</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 4–5 and add to MST

- MST edges
  - 0–7, 1–7, 0–2, 2–3, 5–7

<table>
<thead>
<tr>
<th>edges on PQ</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>
Prım'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
- 0–2
- 2–3
- 5–7
- 4–5

### Edges on PQ (sorted by weight)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<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>
</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 4

MST edges

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

edges on PQ (sorted by weight)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<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>
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 1–2 and discard obsolete edge

**MST edges**

$0-7$  $1-7$  $0-2$  $2-3$  $5-7$  $4-5$

**edges on PQ**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<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>
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 4–7 and discard obsolete edge

MST edges

<table>
<thead>
<tr>
<th>0–7</th>
<th>1–7</th>
<th>0–2</th>
<th>2–3</th>
<th>5–7</th>
<th>4–5</th>
</tr>
</thead>
</table>

edges on PQ (sorted by weight)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<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>
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 0–4 and discard obsolete edge

MST edges

0–7  1–7  0–2  2–3  5–7  4–5

edges on PQ (sorted by weight)

<table>
<thead>
<tr>
<th>edge</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<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>
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 6–2 and add to MST

**MST edges**

0–7  1–7  0–2  2–3  5–7  4–5

**edges on PQ (sorted by weight)**

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<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>
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 6–2 and add to MST

MST edges

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–7</td>
<td></td>
</tr>
<tr>
<td>1–7</td>
<td></td>
</tr>
<tr>
<td>0–2</td>
<td></td>
</tr>
<tr>
<td>2–3</td>
<td></td>
</tr>
<tr>
<td>5–7</td>
<td></td>
</tr>
<tr>
<td>4–5</td>
<td></td>
</tr>
<tr>
<td>6–2</td>
<td></td>
</tr>
</tbody>
</table>

edges on PQ (sorted by weight)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<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>
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.

stop since $V - 1$ edges

**MST edges**

0–7  1–7  0–2  2–3  5–7  4–5  6–2

edges on PQ (sorted by weight)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<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>
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.

![Graph with MST 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

while not finished
    repeatedly delete the min weight edge e = v–w from PQ
    add edge e to tree
    add either v or w to tree
```
Prim's algorithm: lazy implementation

```java
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;
}
```

- add v to T
- for each edge e = v–w, add to PQ if w not already in T
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**

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

**Observation.** For each vertex \( v \), need only lightest edge connecting \( v \) to \( T \).
- MST includes at most one edge connecting \( v \) to \( T \). Why?
- If MST includes such an edge, it must take lightest such edge. Why?
Prim's algorithm: eager 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.

an edge-weighted graph
Prim's algorithm: eager 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.
Prim's algorithm: eager 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.

```
vertices on PQ
(sorted by weight)

<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>4</td>
<td>0–4</td>
<td>0.38</td>
</tr>
<tr>
<td>6</td>
<td>6–0</td>
<td>0.58</td>
</tr>
</tbody>
</table>
```

found connections to 7, 2, 4, and 6
(add to PQ)
Prim's algorithm: eager 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.

### Vertices on PQ
(sorted by weight)

<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>4</td>
<td>0–4</td>
<td>0.38</td>
</tr>
<tr>
<td>6</td>
<td>6–0</td>
<td>0.58</td>
</tr>
</tbody>
</table>
Prim's algorithm: eager 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**

$0-7$

<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0-7</td>
<td>0.16</td>
</tr>
<tr>
<td>2</td>
<td>0-2</td>
<td>0.26</td>
</tr>
<tr>
<td>4</td>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>6</td>
<td>6-0</td>
<td>0.58</td>
</tr>
</tbody>
</table>
Prim's algorithm: eager 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.

![Graph and table showing Prim's algorithm steps]

- Found a connection to 1 (add to PQ).
- Found a connection to 5 (add to PQ).
- A better connection to 4 (decrease key of 4).
- MST edges: 0–7.
- Vertices on PQ (sorted by weight): 1, 2, 4, 5, 6, 7.
Prim's algorithm: eager 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

<table>
<thead>
<tr>
<th>$v$</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1–7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>5</td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>4–7</td>
<td>0.37</td>
</tr>
<tr>
<td>6</td>
<td>6–0</td>
<td>0.58</td>
</tr>
</tbody>
</table>

Vertices sorted by weight (sorted by weight)
Prim's algorithm: eager 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.

### Prim's algorithm

**MST edges**

0–7 1–7
Prim's algorithm: eager 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.
Prim's algorithm: eager 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.

![Graph](image)

<table>
<thead>
<tr>
<th>$v$</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1–7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>5</td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td>3</td>
<td>1–3</td>
<td>0.29</td>
</tr>
<tr>
<td>4</td>
<td>4–7</td>
<td>0.37</td>
</tr>
<tr>
<td>6</td>
<td>6–0</td>
<td>0.58</td>
</tr>
</tbody>
</table>

**MST edges**

0–7  1–7
Prim's algorithm: eager 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.

```
<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1–7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>5</td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td>3</td>
<td>1–3</td>
<td>0.29</td>
</tr>
<tr>
<td>4</td>
<td>4–7</td>
<td>0.37</td>
</tr>
<tr>
<td>6</td>
<td>6–0</td>
<td>0.58</td>
</tr>
</tbody>
</table>
```

MST edges

- 0–7
- 1–7
- 0–2
Prim's algorithm: eager 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

$0 \rightarrow 7 \quad 1 \rightarrow 7 \quad 0 \rightarrow 2$
Prim's algorithm: eager 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.

Prim's edges:

0–7  1–7  0–2  2–3

<table>
<thead>
<tr>
<th>$v$</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1–7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>3</td>
<td>2–3</td>
<td>0.17</td>
</tr>
<tr>
<td>5</td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>4–7</td>
<td>0.37</td>
</tr>
<tr>
<td>6</td>
<td>6–2</td>
<td>0.40</td>
</tr>
</tbody>
</table>
Prim's algorithm: eager 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  0–2  2–3

<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1–7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>3</td>
<td>2–3</td>
<td>0.17</td>
</tr>
<tr>
<td>5</td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>4–7</td>
<td>0.37</td>
</tr>
<tr>
<td>6</td>
<td>6–2</td>
<td>0.40</td>
</tr>
</tbody>
</table>
Prim's algorithm: eager 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 0–2 2–3
Prim's algorithm: eager 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   0–2   2–3

<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1–7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>3</td>
<td>2–3</td>
<td>0.17</td>
</tr>
<tr>
<td>5</td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>4–7</td>
<td>0.37</td>
</tr>
<tr>
<td>6</td>
<td>6–2</td>
<td>0.40</td>
</tr>
</tbody>
</table>
Prim's algorithm: eager 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.

```
+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 |
+---+---+---+---+---+
| 5 | 7 | 0 | 6 | 6 |
+---+---+---+---+---+
```

MST edges

- 0–7
- 1–7
- 0–2
- 2–3
- 5–7

<table>
<thead>
<tr>
<th>$v$</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1–7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>3</td>
<td>2–3</td>
<td>0.17</td>
</tr>
<tr>
<td>5</td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>4–7</td>
<td>0.37</td>
</tr>
<tr>
<td>6</td>
<td>6–2</td>
<td>0.40</td>
</tr>
</tbody>
</table>
Priming's algorithm: eager implementation demo

- Start with vertex 0 and greedily growing 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

### Table: Graph Connectivity

<table>
<thead>
<tr>
<th>$v$</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0-7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1-7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0-2</td>
<td>0.26</td>
</tr>
<tr>
<td>3</td>
<td>2-3</td>
<td>0.17</td>
</tr>
<tr>
<td>5</td>
<td>5-7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>4-7</td>
<td>0.35</td>
</tr>
</tbody>
</table>

- **4**
- **5**
- **6**

- Better connection to 4 (decrease key of vertex 4)

- No longer the best connection to 4 (discard)
Prim's algorithm: eager 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  0–2  2–3  5–7
Prim's algorithm: eager 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.


\[
\begin{align*}
\text{MST edges} & \quad 0-7 \quad 1-7 \quad 0-2 \quad 2-3 \quad 5-7 \quad 4-5
\end{align*}
\]
Prim's algorithm: eager 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**

- $0-7$
- $1-7$
- $0-2$
- $2-3$
- $5-7$
- $4-5$

<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0-7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1-7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0-2</td>
<td>0.26</td>
</tr>
<tr>
<td>3</td>
<td>2-3</td>
<td>0.17</td>
</tr>
<tr>
<td>5</td>
<td>5-7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>4-5</td>
<td>0.35</td>
</tr>
<tr>
<td>6</td>
<td>6-2</td>
<td>0.40</td>
</tr>
</tbody>
</table>

already a better connection to 6 (discard)
Prim's algorithm: eager 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  0–2  2–3  5–7  4–5

<table>
<thead>
<tr>
<th>$v$</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1–7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>3</td>
<td>2–3</td>
<td>0.17</td>
</tr>
<tr>
<td>5</td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>4–5</td>
<td>0.35</td>
</tr>
<tr>
<td>6</td>
<td>6–2</td>
<td>0.40</td>
</tr>
</tbody>
</table>
Prim's algorithm: eager 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  0–2  2–3  5–7  4–5  6–2

<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1–7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>3</td>
<td>2–3</td>
<td>0.17</td>
</tr>
<tr>
<td>5</td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>4–5</td>
<td>0.35</td>
</tr>
<tr>
<td>6</td>
<td>6–2</td>
<td>0.40</td>
</tr>
</tbody>
</table>
Prim's algorithm: eager 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 0–2 2–3 5–7 4–5 6–2

<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1–7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>3</td>
<td>2–3</td>
<td>0.17</td>
</tr>
<tr>
<td>5</td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>4–5</td>
<td>0.35</td>
</tr>
<tr>
<td>6</td>
<td>6–2</td>
<td>0.40</td>
</tr>
</tbody>
</table>
Prim's algorithm: eager implementation

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 \( v = \text{weight of lightest edge connecting } v \text{ to } T \).

- 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 lightest edge connecting \( x \) to \( T \)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1-7</td>
<td>0.19</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0-2</td>
<td>0.26</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1-3</td>
<td>0.29</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0-4</td>
<td>0.38</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5-7</td>
<td>0.28</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6-0</td>
<td>0.58</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0-7</td>
<td>0.16</td>
<td></td>
</tr>
</tbody>
</table>

- black: on MST
- red: on PQ

PQ has at most one entry per vertex
Indexed priority queue

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

- Insert a key associated with a given index.
- Delete a minimum key and return associated index.
- Decrease the key associated with a given index.

```java
public class IndexMinPQ<Key extends Comparable<Key>> {
    IndexMinPQ(int N) {
        // create indexed priority queue
        // with indices 0, 1, ..., N - 1
        associate key with index i
    }
    void insert(int i, Key key) {
        // remove a minimal key and return its associated index
        decrease the key associated with index i
    }
    int delMin() {
        is i an index on the priority queue?
    }
    void decreaseKey(int i, Key key) {
        is the priority queue empty?
    }
    boolean contains(int i) {
        number of keys in the priority queue
    }
    boolean isEmpty() {
    }
    int size() {
    }
}
```
Indexed priority queue: implementation

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

- Start with same code as MinPQ.
- Maintain parallel arrays so that:
  - keys[i] is the priority of vertex i
  - qp[i] is the heap position of vertex i
  - pq[i] is the index of the key in heap position 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>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>
<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>
</tbody>
</table>

decrease key of vertex 2 to C

vertex 2 is at heap index 4
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>

\( ^\dagger \) 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.
Computing a spanning tree in a graph

Does this algorithm seem familiar?
- Prim's algorithm is essentially the same algorithm as Dijkstra's.
- Both are in a family of algorithms that compute a spanning tree.

Main distinction: rule used to choose next vertex for the tree.
- Prim: Closest vertex to the tree (via an undirected edge).
- Dijkstra: Closest vertex to the source (via a directed path).

Note: DFS and BFS are also in this family of algorithms.
Does a linear-time MST algorithm exist?

<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>$E \log \log V$</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 $N \log N$ time.
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.
**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.
Dendrogram of cancers in human

Tumors in similar tissues cluster together.

Reference: Botstein & Brown group