4.3 Minimum Spanning Trees

- introduction
- edge-weighted graph API
- cut property
- Kruskal’s algorithm
- Prim’s algorithm
4.3 Minimum Spanning Trees

- introduction
- edge-weighted graph API
- cut property
- 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.

![Spanning tree diagram](image-url)
**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.
Minimum spanning tree problem

Input. Connected, undirected graph $G$ with positive edge weights.
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?
Let $T$ be a spanning tree of a connected graph $G$ with $V$ vertices. Which of the following statements are true?

A. $T$ contains exactly $V - 1$ edges.
B. Removing any edge from $T$ disconnects it.
C. Adding any edge to $T$ creates a cycle.
D. All of the above.
Network design

paved, but didn’t want to spend more money than necessary because the city also wanted to build a swimming pool. The mayor therefore specified two conditions:

1. Enough streets must be paved so that it is possible for everyone to travel from their house to anyone else’s house only along paved roads, and
2. The paving should cost as little as possible.

Here is the layout of the city. The number of paving stones between each house represents the cost of paving that route. Find the best route that connects all the houses, but uses as few counters (paving stones) as possible.

Solution: the graph (for another muddy city) and the paving.

Other practical applications based on minimal spanning trees include:

• Taxonomy.
• Cluster analysis: clustering points in the plane, single-linkage clustering, graph-theoretic clustering, and clustering gene expression data.

http://www.utdallas.edu/~besp/teaching/mst-applications.pdf
Rules for Biologically Inspired Adaptive Network Design

Atushi Tero, Seiji Takagi, Tetsu Saigusa, Kentaro Ito, Dan P. Bebber, Mark D. Fricker, Kenji Yumiki, Ryo Kobayashi, Toshiyuki Nakagaki

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

MST is fundamental problem with diverse applications.

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

http://www.utdallas.edu/~besp/teaching/mst-applications.pdf
4.3 Minimum Spanning Trees

- introduction
- edge-weighted graph API
- cut property
- 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);`
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;
    }
}
```

- Constructor
- Either endpoint
- Other endpoint
- Compare edges by weight
Edge-weighted graph API

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

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

Maintain vertex-indexed array of Edge lists.
Edge-weighted graph: adjacency-lists implementation

```java
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];
    }
}
```

same as Graph, but adjacency lists of Edges instead of integers

constructor

add edge to both adjacency lists
4.3 Minimum Spanning Trees

- introduction
- edge-weighted graph API
- cut property
- Kruskal’s algorithm
- Prim’s algorithm
Simplifying assumptions

For simplicity, we assume:

- No parallel edges.
- The graph is connected. \(\Rightarrow\) MST exists.
- The edge weights are distinct. \(\Rightarrow\) MST is unique.  

Note. Algorithms still work even if parallel edges or duplicate edge weights.
**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.
Which is the min weight edge crossing the cut \{ 2, 3, 5, 6 \}?

A. 0–7 (0.16)
B. 2–3 (0.17)
C. 0–2 (0.26)
D. 5–7 (0.28)
**Cut property: correctness proof**

**Def.** A cut is a partition of a graph’s vertices into two (nonempty) sets.

**Def.** A crossing edge connects two vertices in different sets.

**Cut property.** Given any cut, the min-weight crossing edge $e$ is in the MST.

**Pf.** [by contradiction] Suppose $e$ is not in the MST.

- Some other edge $f$ in the MST 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. □

---

![Diagram](image)

the MST does not contain $e$
Application of cut property [warmup for Kruskal’s algorithm]

Def. A cut is a partition of a graph’s vertices into two (nonempty) sets.
Def. A crossing edge connects two vertices in different sets.

Cut property. Given any cut, the min-weight crossing edge \( e \) is in the MST.

Exercise. In any connected graph of \( \geq 3 \) vertices (distinct edge weights; no parallel edges):

- Show that the edge with lowest weight is in the MST.
- Show that the edge with second lowest weight is in the MST.
- Note that the edge with third lowest weight may not be in the MST.
4.3 Minimum Spanning Trees

- introduction
- edge-weighted graph API
- cut property
- Kruskal’s algorithm
- Prim’s algorithm
Consider edges in ascending order of weight.

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

**an edge-weighted graph**

```plaintext
<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: visualization
Kruskal’s algorithm: correctness proof

**Proposition.** Kruskal’s algorithm computes the MST. 

**Pf.** Let $T$ be the “tree” at some point during execution, and $e$ the next edge considered.

[Case 1] Kruskal’s algorithm adds edge $e = v–w$ to $T$.
- Vertices $v$ and $w$ are in different connected components of $T$.
- Cut = set of vertices connected to $v$ in $T$.
- By construction of cut, no edge crossing cut is in $T$.
- No edge crossing cut has lower weight. Why?
- Cut property $\Rightarrow$ edge $e$ is in the MST.
- $\Rightarrow$ Kruskal’s algorithm correctly adds $e$ to $T$.

[Case 2] Kruskal’s algorithm discards edge $e = v–w$.
- From Case 1, all edges in $T$ are in the MST.
- The MST can’t contain a cycle.
- $\Rightarrow$ Kruskal’s algorithm correctly discards $e$. 

Recall: increasing order of edge weights

![Diagram showing Kruskal's algorithm](image)
Challenge. Would adding edge $v-w$ to tree $T$ create a cycle? If not, add it.

How difficult to implement? (Worst case order of growth of best impl.)

A. 1
B. $\log V$
C. $V$
D. $E + V$
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 $w$. 

![Diagram](image_url)
public class KruskalMST {
    private Queue<Edge> mst = new Queue<Edge>();

    public KruskalMST(EdgeWeightedGraph G) {
        DirectedEdge[] edges = G.edges();
        Arrays.sort(edges);
        UF uf = new UF(G.V());

        for (int i = 0; i < G.E(); i++) {
            Edge e = edges[i];
            int v = e.either(), w = e.other(v);
            if (uf.find(v) != uf.find(w)) {
                uf.union(v, w);
                mst.enqueue(e);
            }
        }
    }

    public Iterable<Edge> edges() {
        return mst;
    }
}
Proposition. Kruskal’s algorithm computes MST in time proportional to $E \log V$ (in the worst case).

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>time per op</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sort</strong></td>
<td>1</td>
<td>$E \log E$</td>
</tr>
<tr>
<td><strong>Union</strong></td>
<td>$V - 1$</td>
<td>$\log V$ $\dagger$</td>
</tr>
<tr>
<td><strong>Find</strong></td>
<td>$2E$</td>
<td>$\log V$ $\dagger$</td>
</tr>
</tbody>
</table>

$\dagger$ using weighted quick union

same as $E \log V$ if no parallel edges

See Piazza post @519 for a detailed explanation
[https://piazza.com/class/jrp35q44vo35p2?cid=519](https://piazza.com/class/jrp35q44vo35p2?cid=519)
4.3 Minimum Spanning Trees

- introduction
- edge-weighted graph API
- cut property
- Kruskal’s algorithm
- Prim’s algorithm

Only lazy implementation covered; see textbook / videos for eager implementation
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: visualization
Proposition. Prim’s algorithm computes the MST.

Pf.

• Cut = set of vertices in $T$.
• The edges crossing this cut are precisely those considered by Prim’s algorithm (edges with exactly one endpoint in $T$).
• Cut property $\Rightarrow$ edge added by Prim’s algorithm must be in the MST.

edge $e = 7-5$ added to tree
Challenge. Find the min weight edge with exactly one endpoint in $T$.

How difficult to implement?

A. 1
B. $\log E$
C. $V$
D. $E$

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 $T$.
- If both endpoints $v$ and $w$ are marked (both in $T$), disregard.
- 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 $T$)

![Diagram of a graph with priority queue of crossing 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.

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

```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);
        }
    }
}
```

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;  }
Lazy Prim’s algorithm: running time

**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><strong>DELETE-MIN</strong></td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
<tr>
<td><strong>INSERT</strong></td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
</tbody>
</table>
# MST: algorithms of the day

<table>
<thead>
<tr>
<th>algorithm</th>
<th>visualization</th>
<th>bottleneck</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kruskal</td>
<td><img src="image" alt="Kruskal Visualization" /></td>
<td>sorting union–find</td>
<td>$E \log V$</td>
</tr>
<tr>
<td>Prim</td>
<td><img src="image" alt="Prim Visualization" /></td>
<td>priority queue</td>
<td>$E \log V$</td>
</tr>
</tbody>
</table>