4.3 Minimum Spanning Trees

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

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

**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 spanning tree and a graph](attachment:image.png)
Minimum spanning tree problem

Input. Connected, undirected graph $G$ with positive edge weights.

edge-weighted digraph $G$
Minimum spanning tree problem

Input. Connected, undirected graph $G$ with positive edge weights.

Output. A spanning tree of minimum weight.

![Minimum spanning tree](image)

Minimum spanning tree $T$
(weight = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7)

**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.
Image processing

MST dithering

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

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.

- 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.
- Network design (communication, electrical, hydraulic, computer, road).
- Approximation algorithms for NP-hard problems (e.g., TSP, Steiner tree).

4.3 Minimum Spanning Trees

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

For simplicity, we assume:

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

Note. Algorithms still work correctly even if duplicate edge weights.

see Exercise 4.3.3
**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. Suppose \( e \) 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. □
4.3 Minimum Spanning Trees

- introduction
- cut property
- edge-weighted graph API
- Kruskal’s algorithm
- Prim’s algorithm
- context
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
public class EdgeWeightedGraph {
    EdgeWeightedGraph(int V) {
        // create an empty graph with V vertices
    }
    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.

tinyEWG.txt

V

adj[]

E

Bag

objects

references to the
same Edge object
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>public class MST</th>
</tr>
</thead>
<tbody>
<tr>
<td>MST(EdgeWeightedGraph G)</td>
</tr>
<tr>
<td>Iterable&lt;Edge&gt; edges()</td>
</tr>
<tr>
<td>double weight()</td>
</tr>
</tbody>
</table>
4.3 Minimum Spanning Trees

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

Consider edges in ascending order of weight.
- Add next edge to tree $T$ unless doing so would create a cycle.
Kruskal’s algorithm: visualization
Kruskal’s algorithm: correctness proof

**Proposition.** [Kruskal 1956] Kruskal’s algorithm computes the MST.

**Pf.** [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.
Kruskal’s algorithm: correctness proof

**Proposition.** [Kruskal 1956] Kruskal’s algorithm computes the MST.

**Pf.** [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. □
Kruskal’s algorithm: implementation challenge

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

**How difficult to implement?**

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

**Case 1:** $v$ and $w$ in same component

**Case 2:** $v$ and $w$ in different components
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 2: adding $v–w$ creates a cycle

Case 1: add $v–w$ to $T$ and merge sets containing $v$ and $w$
Kruskal’s algorithm: Java implementation

```java
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;
    }
}
```
**Kruskal’s algorithm: running time**

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

**Pf.**

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

$^\dagger$ using weighted quick union
Greed is good

Gordon Gecko (Michael Douglas) evangelizing the importance of greed (in algorithm design?)
Wall Street (1986)
**Maximum Spanning Tree**

**Problem.** Given an undirected graph $G$ with positive edge weights, find a spanning tree that maximizes the sum of the edge weights.

**Running time.** $E \log E$ (or better).

![Diagram of a maximum spanning tree T (weight = 104)]
4.3 Minimum Spanning Trees

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

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

an edge-weighted graph
Prim’s algorithm: visualization
Proposition. [Jarník 1930, Dijkstra 1957, Prim 1959]
Prim's algorithm computes the MST.

Pf. Let \( e = \text{min weight edge with exactly one endpoint in } T \).
- Cut = set of vertices in \( T \).
- No crossing edge is in \( T \).
- No crossing edge has lower weight.
- Cut property \( \Rightarrow \) edge \( e \) is in the MST. •

edge \( e = 7-5 \) added to tree
Prim’s algorithm: implementation challenge

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

How difficult to implement?

A. \( 1 \)
B. \( \log E \)
C. \( V \)
D. \( E \)
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 $T$)
• 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

an edge-weighted graph
Prim’s algorithm: lazy implementation

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

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

        while (!pq.isEmpty() && mst.size() < G.V() - 1) {
            Edge e = pq.delMin();
            int v = e.either(), w = e.other(v);
            if (marked[v] && marked[w]) continue;
            mst.enqueue(e);
            if (!marked[v]) visit(G, v);
            if (!marked[w]) visit(G, w);
        }
    }
}
```

assume G is connected

repeatedly delete the min weight edge e = v–w from PQ
ignore if both endpoints in T
add edge e to tree
add either v or w to tree
Prim’s algorithm: lazy implementation

```
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
**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>
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

<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: 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>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>

MST edges

0–7  1–7  0–2  2–3  5–7  4–5  6–2
**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 $v$; 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 $T$
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 PQ with indices 0, 1, …, n – 1

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

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

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

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

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

    int size()
    // number of keys in the priority queue
```

For Prim’s algorithm, \( n = V \) and index = vertex.
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>

vertex 2 is at heap index 4

decrease key of vertex 2 to C
Prim’s algorithm: which priority queue?

Depends on PQ implementation: \( V \text{ INSERT}, V \text{ DELETE-MIN}, E \text{ DECREASE-KEY}. \)

<table>
<thead>
<tr>
<th>PQ implementation</th>
<th>INSERT</th>
<th>INSERT–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 complete graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.
4.3 Minimum Spanning Trees

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

deterministic compare–based MST algorithms

Remark. Linear-time randomized MST algorithm (Karger–Klein–Tarjan).
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; $n \log n$ using Delaunay triangulation.
**Problem.** Given an edge-weighted graph $G$, find a spanning tree that minimizes the maximum weight of its edges.

**Running time.** $E \log E$ (or better).

Note: not necessarily a MST

**Minimum bottleneck spanning tree $T$ (bottleneck = 9)**