4.3 Minimum Spanning Trees

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

Minimum spanning tree

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

**Def.** A spanning tree of $G$ is a subgraph $T$ that is connected and acyclic.

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

![graph G](image)

Minimum spanning tree

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

**Def.** A spanning tree of $G$ is a subgraph $T$ that is connected and acyclic.

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

![not connected](image)
Minimum spanning tree

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

**Def.** A **spanning tree** of $G$ is a subgraph $T$ that is connected and acyclic.

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

![Minimum spanning tree](http://www.flickr.com/photos/1eddistrict/21580840)

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

Network design

*MST of bicycle routes in North Seattle*

![Network design](http://www.flickr.com/photos/1eddistrict/21580840)

Models of nature

*MST of random graph*

![Models of nature](http://algo.inria.fr/brotin/gallery.html)
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).

Simplifying assumptions

**Simplifying assumptions.**
- Edge weights are distinct.
- Graph is connected.

**Consequence.** MST exists and is unique.

---

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 is lower weight.
- Contradiction. □

---

Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V-1$ edges are colored black.
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: 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 if cut with no black crossing edges. (consider cut whose vertices are any one connected component)

Greedy MST algorithm: efficient implementations

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

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

- **Ex 1.** Kruskal’s algorithm. [stay tuned]
- **Ex 2.** Prim’s algorithm. [stay tuned]
- **Ex 3.** Borůvka’s algorithm.

Removing two simplifying assumptions

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

**Q.** What if graph is not connected?
- **A.** Compute minimum spanning forest = MST of each component.
Greed is good

Gordon Gecko (Michael Douglas) address to Teldar Paper Stockholders in Wall Street (1986)

Weighted edge API

Edge abstraction needed for weighted edges.

```
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 v)
    {
        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;
    }

    public String toString()
    {
        return v + weight + w;
    }
}
```

Idiom for processing an edge $e$: $v = e$.either(), $w = e$.other(v);

Weighted edge: Java implementation

```
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 v)
    {
        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;
    }

    public String toString()
    {
        return v + weight + w;
    }
}
```
public class EdgeWeightedGraph
{
    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]; }
}

public class MST
{
    MST(EdgeWeightedGraph G)
    {
        // constructor
        edges()
        double weight()
    }

    % java MST tinyEWG.txt
    0-7 0.16
    1-7 0.19
    0-2 0.26
    2-3 0.17
    5-7 0.28
    4-5 0.35
    6-2 0.40
    1.81

Conventions. Allow self-loops and parallel edges.

Minimum spanning tree API

Q. How to represent the MST?

An edge-weighted graph and its MST

Edge-weighted graph representation

Edge-weighted graph: adjacency-lists representation

Maintain vertex-indexed array of Edge lists.

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

same

constructor

constructor

edges in MST

weight of MST
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>

public static void main(String[] args)
{
    In in = new In(args[0]);
    EdgeWeightedGraph G = new EdgeWeightedGraph(in);
    MST mst = new MST(G);
    for (Edge e : mst.edges())
        StdOut.println(e.weight());
}

% java MST tinyEWG.txt
0-7 0.16
1-7 0.19
0-2 0.26
2-3 0.17
5-7 0.28
4-5 0.35
6-2 0.40
1.81

4.3 Minimum Spanning Trees

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

Kruskal's algorithm demo

Consider edges in ascending order of weight.
- Add next edge to tree 7 unless doing so would create a cycle.

http://algs4.cs.princeton.edu

Kruskal's algorithm demo

Consider edges in ascending order of weight.
- Add next edge to tree 7 unless doing so would create a cycle.

http://algs4.cs.princeton.edu
**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 $e = v \rightarrow w$ black.
- Cut = set of vertices connected to $v$ in tree $T$.
- No crossing edge is black.
- No crossing edge has lower weight. Why?

**Kruskal’s algorithm: implementation challenge**

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

**How difficult?**

- $E + V$
- $V$
- $\log V$
- $\log* V$
- 1

**Efficient solution.** Use the union-find data structure.

- Maintain a set for each connected component in $T$.
- If $v$ and $w$ are in same set, then adding $v \rightarrow w$ would create a cycle.
- To add $v \rightarrow w$ to $T$, merge sets containing $v$ and $w$. 

**Case 1:** adding $v \rightarrow w$ creates a cycle

**Case 2:** add $v \rightarrow w$ to $T$ and merge sets containing $v$ and $w$
Kruskal’s algorithm: Java implementation

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

    public KruskalMST(EdgeWeightedGraph G) {
        MinPQ<Edge> pq = new MinPQ<Edge>();
        for (Edge e : G.edges())
            pq.insert(e);
        UF uf = new UF(G.V());
        while (!pq.isEmpty() && mst.size() < G.V() - 1)
            Edge e = pq.delMin();
            int v = e.other(v);
            if (!uf.connected(v, w))
                uf.union(v, w);
            mst.enqueue(e);
    }

    public Iterable<Edge> edges() {
        return mst;
    }
}
```

Kruskal’s algorithm: running time

### Proposition

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

### Pf.

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>time per op</th>
</tr>
</thead>
<tbody>
<tr>
<td>build pq</td>
<td>1</td>
<td>$E \log E$</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

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

4.3 Minimum Spanning Trees

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

Prim’s algorithm demo

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

An edge-weighted graph: 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
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: proof of correctness

**Proposition.** [Jarník 1930, Dijkstra 1957, Prim 1959]
Prım's algorithm computes the MST.

**Pf.** Prim’s algorithm is a special case of the greedy MST algorithm.
- Suppose edge $e = \min$ 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.

### Prim's algorithm: implementation challenge

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

**How difficult?**
- $E$  
  - try all edges
- $V$
- $\log E$
- $\log^* E$
- 1

(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 to PQ any edge incident to $w$ (assuming other endpoint not in $T$)
  - add $e$ to $T$ and mark $w$

$e = v \rightarrow w$ is min weight edge with exactly one endpoint in $T$

```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);
        }
    }
}
```

**Prim's algorithm (lazy) demo**

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

MST edges
0-7  1-7  0-2  2-3  5-7  4-5  6-2

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

assume G is connected
repeatedly delete the min weight edge $e = v \rightarrow w$ from PQ
ignore if both endpoints in $T$
add edge $e$ to tree
add $v$ or $w$ to tree
Prim's algorithm: lazy implementation

```java
group private void visit(WeightedGraph G, int v) {
    marked[v] = true;
    for (Edge e : G.adj(v))
        if (!marked[e.other(v)])
            pq.insert(e);
}
group 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>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 $T$.

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

- 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 shortest edge connecting $x$ to $T$

Prim's algorithm (eager) demo

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

![an edge-weighted graph]
Prim’s algorithm (eager) demo

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

### Indexed priority queue

Associate an index between 0 and $N - 1$ with each key in a priority queue.
- Supports `insert` and `delete-the-minimum`.
- Supports `decrease-key` given the index of the key.

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

Indexed priority queue implementation

**Binary heap implementation.** [see Section 2.4 of textbook]
- Start with same code as MinPQ.
- Maintain parallel arrays `keys[]`, `pq[]`, and `qp[]` so that:
  - `keys[i]` is the priority of $i$
  - `pq[i]` is the index of the key in heap position $i$
  - `qp[i]` is the heap position of the key with index $i$
- Use `swim(qp[i])` to implement `decreaseKey(i, key)`.

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys[]</td>
<td>A</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>I</td>
<td>N</td>
<td>G</td>
<td>-</td>
</tr>
<tr>
<td>pq[]</td>
<td>-</td>
<td>0</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>qp[]</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>-</td>
</tr>
</tbody>
</table>

Prim’s algorithm: which priority queue?

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

<table>
<thead>
<tr>
<th>PQ implementation</th>
<th>insert</th>
<th>delete-min</th>
<th>decrease-key</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>$V$</td>
<td>$V$</td>
<td>$V$</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 (Johnson 1975)</td>
<td>$d \log_d V$</td>
<td>$d \log_d V$</td>
<td>$d \log_d V$</td>
<td>$E \log_d V$</td>
</tr>
<tr>
<td>Fibonacci heap (Fredman–Tarjan 1984)</td>
<td>$1^\dagger$</td>
<td>$\log V^\dagger$</td>
<td>$1^\dagger$</td>
<td>$E + V \log V$</td>
</tr>
</tbody>
</table>

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

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

### Deterministic Compare-Based MST Algorithms

<table>
<thead>
<tr>
<th>Year</th>
<th>Worst Case</th>
<th>Discovered By</th>
</tr>
</thead>
<tbody>
<tr>
<td>1975</td>
<td>$E \log \log V$</td>
<td>Yao</td>
</tr>
<tr>
<td>1976</td>
<td>$E \log \log V$</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-Gallif-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.

#### 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 $l'$ 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