4.3 Minimum Spanning Trees

- edge-weighted graph API
- greedy algorithm
- Kruskal's algorithm
- Prim's algorithm
- advanced topics
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

**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

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

**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.

Brute force. Try all spanning trees?

![Graph with edge weights and minimum spanning tree](image-url)
Network design

MST of bicycle routes in North Seattle

http://www.flickr.com/photos/ewedistrict/21980840
Medical image processing

MST describes arrangement of nuclei in the epithelium for cancer research

http://www.bccrc.ca/ci/ta01_archlevel.html
MST of tissue relationships measured by gene expression correlation coefficient

http://riodb.ibase.aist.go.jp/CELLPEDIA
Applications

MST is fundamental problem with diverse applications.

• 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, cable, computer, road).

• edge-weighted graph API
• greedy algorithm
• Kruskal's algorithm
• Prim's algorithm
• advanced topics
Weighted edge API

Edge abstraction needed for weighted edges.

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**

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

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

Maintain vertex-indexed array of Edge lists (use Bag abstraction).

```
adj[]

0: 6 0 .58 → 0 2 .26 → 0 4 .38 → 0 7 .16
   1 3 .29 → 1 2 .36 → 1 7 .19 → 1 5 .32

1: 6 2 .40 → 2 7 .34 → 1 2 .36 → 0 2 .26 → 2 3 .17
   3 6 .52 → 1 3 .29 → 2 3 .17

2: 6 4 .93 → 0 4 .38 → 4 7 .37 → 4 5 .35
   1 5 .32 → 5 7 .28 → 4 5 .35

3: 6 4 .93 → 6 0 .58 → 3 6 .52 → 6 2 .40
   2 7 .34 → 1 7 .19 → 0 7 .16 → 5 7 .28
   5 7 .28

4: 0 .26

5: 0 .16

6: 0 .32

7: 0 .17

V: 8 16
E: 4 5 0 .35
   4 7 0 .37
   5 7 0 .28
   0 7 0 .16
   1 5 0 .32
   0 4 0 .38
   2 3 0 .17
   1 7 0 .19
   0 2 0 .26
   1 2 0 .36
   1 3 0 .29
   2 7 0 .34
   6 2 0 .40
   3 6 0 .52
   6 0 0 .58
   6 4 0 .93
```

References to the same Edge object.
public class EdgeWeightedGraph
{
    private final int V;
    private final Bag<Edge>[] adj;

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

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

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

```java
public class MST

MST(EdgeWeightedGraph G)  // constructor

Iterable<Edge> edges()  // edges in MST

double weight()  // weight of MST
```

An edge-weighted graph and its MST:

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

The minimum spanning tree API includes:
- The `MST` class with a constructor that takes an `EdgeWeightedGraph` as an argument.
- An `edges()` method that returns an `Iterable<Edge>` containing the edges in the minimum spanning tree.
- A `weight()` method that returns the weight of the minimum spanning tree.

The example graph includes a table with edge-weighted values and a diagram illustrating the minimum spanning tree with the specified edges and weights.
Minimum spanning tree API

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

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

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

% java MST tinyEWG.txt
0-7 0.16
1-7 0.19
0-2 0.26
2-3 0.17
5-7 0.28
4-5 0.35
6-2 0.40
1.81
• edge-weighted graph API
• greedy algorithm
• Kruskal's algorithm
• Prim's algorithm
• advanced topics
**Cut property**

**Simplifying assumptions.** Edge weights are distinct; graph is connected.

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

Simplifying assumptions. Edge weights are distinct; graph is connected.

Def. A cut in a graph is a partition of its vertices into two (nonempty) sets. 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. Let $e$ be the min-weight crossing edge in cut.

• 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 is lower weight.
• Contradiction. $\blacksquare$
**Proposition.** The following algorithm computes the MST:

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Continue until $V - 1$ edges are colored black.
Greedy MST algorithm: correctness proof

**Proposition.** The following algorithm computes the MST:

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Continue until $V-1$ edges are colored black.

**Pf.**

- Any edge colored black is in the MST (via cut property).
- If fewer than $V-1$ black edges, there exists a cut with no black crossing edges. (consider cut whose vertices are one connected component)
**Proposition.** The following algorithm computes the MST:

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Continue until $V - 1$ edges are colored black.

**Efficient implementations.** How to choose cut? How to 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)
- edge-weighted graph API
- greedy algorithm
- Kruskal's algorithm
- Prim's algorithm
- advanced topics
Kruskal's algorithm. [Kruskal 1956] Consider edges in ascending order of weight. Add the next edge to the tree $T$ unless doing so would create a cycle.
Kruskal's algorithm: visualization
**Proposition.** Kruskal's algorithm computes the MST.

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

- Suppose Kruskal's algorithm colors the edge \( e = v-w \) black.
- Cut = set of vertices connected to \( v \) in tree \( T \).
- No crossing edge is black.
- No crossing edge has lower weight. Why?
Kruskal's algorithm: implementation challenge

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

**How difficult?**

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

![Diagram](image.png)

- **add edge to tree**
- **adding edge to tree would create a cycle**

run DFS from $v$, check if $w$ is reachable
(T has at most $V - 1$ edges)

use the union-find data structure!
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 $w$.
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);

        UnionFind uf = new UnionFind(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;  }
}
**Proposition.** Kruskal's algorithm computes MST in time proportional to $E \log E$ (in the worst case).

**Pf.**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Frequency</th>
<th>Time per op</th>
</tr>
</thead>
<tbody>
<tr>
<td>build pq</td>
<td>1</td>
<td>$E$</td>
</tr>
<tr>
<td>delete-min</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
<tr>
<td>union</td>
<td>$V$</td>
<td>$\log^* V$</td>
</tr>
<tr>
<td>connected</td>
<td>$E$</td>
<td>$\log^* V$</td>
</tr>
</tbody>
</table>

† amortized bound using weighted quick union with path compression

**Remark.** If edges are already sorted, order of growth is $E \log^* V$. recall: $\log^* V \leq 5$ in this universe
edge-weighted graph API
• greedy algorithm
• Kruskal's algorithm
• Prim's algorithm
• advanced topics
Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]

Start with vertex 0 and greedily grow tree $T$. At each step, add to $T$ the min weight edge with exactly one endpoint in $T$. 
Prim’s algorithm: visualization
Prim's algorithm: implementation challenge

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

**How difficult?**

- $O(E)$ time.  
  - try all edges

- $O(V)$ time.  
  - use a priority queue!

- $O(\log E)$ time.

- $O(\log^* E)$ time.

- Constant time.

![Graph with edge weights](image_url)
Proposition. Prim's algorithm computes the MST.

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

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

- Delete min to determine next edge $e = v$–$w$ to add to $T$.
- Disregard if both endpoints $v$ and $w$ are in $T$.
- Otherwise, let $v$ be vertex not in $T$:
  - add to PQ any edge incident to $v$ (assuming other endpoint not in $T$)
  - add $v$ to $T$

![Graph with edges and weights](image)
Prim's algorithm demo: lazy implementation

Use $\text{MinPQ}$: key = edge, prioritized by weight.
(lazy version leaves some obsolete edges on the PQ)
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())
        {
            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 implementation

repeatedly delete the min weight edge e = v–w from PQ
ignore if both endpoints in T
add edge e to tree
add v or w to tree
assume G is connected
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>
Challenge. Find min weight edge with exactly one endpoint in $T$.

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

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

Use IndexMinPQ: key = edge weight, index = vertex.
(eager version has at most one PQ entry per vertex)
 Indexed priority queue

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

- Client can insert and delete-the-minimum.
- Client can change the key by specifying the index.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>IndexMinPQ(int N)</code></td>
<td>create indexed priority queue with indices 0, 1, ..., N-1</td>
</tr>
<tr>
<td><code>void insert(int k, Key key)</code></td>
<td>associate key with index k</td>
</tr>
<tr>
<td><code>void decreaseKey(int k, Key key)</code></td>
<td>decrease the key associated with index k</td>
</tr>
<tr>
<td><code>boolean contains()</code></td>
<td>is k an index on the priority queue?</td>
</tr>
<tr>
<td><code>int delMin()</code></td>
<td>remove a minimal key and return its associated index</td>
</tr>
<tr>
<td><code>boolean isEmpty()</code></td>
<td>is the priority queue empty?</td>
</tr>
<tr>
<td><code>int size()</code></td>
<td>number of entries in the priority queue</td>
</tr>
</tbody>
</table>
Indexed priority queue implementation

Implementation.

• 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[k]) implement decreaseKey(k, key).

```
keys[i]  A  S  O  R  T  I  N  G  -
pq[i]    -  0  6  7  2  1  5  4  3
qp[i]    1  5  4  8  7  6  2  -
```

```
  A
 / \
(2) N O S
 / \
(4) R 5
  \
  \
  8 
```

```
  G
 / \
(3) I T
 / \
  6 
```
Prim’s algorithm: running time

 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>array</td>
<td>1</td>
<td>( V )</td>
<td>1</td>
<td>( V^2 )</td>
</tr>
<tr>
<td>binary heap (Johnson 1975)</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>( d \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.
› edge-weighted graph API
› greedy algorithm
› Kruskal's algorithm
› Prim's algorithm
› advanced topics
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>20xx</td>
<td>optimal</td>
<td>Pettie-Ramachandran</td>
</tr>
<tr>
<td></td>
<td>$E$</td>
<td>???</td>
</tr>
</tbody>
</table>

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

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

Brute force. Compute $\sim N^2 / 2$ distances and run Prim's algorithm.

Ingenuity. Exploit geometry and do it in $\sim c N \log N$. 