4.3 Minimum Spanning Trees

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.


**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?

---

**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).

**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
Genetic research

MST of tissue relationships measured by gene expression correlation coefficient

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

Weighted edge API

Edge abstraction needed for weighted edges.

```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 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 double weight() {
        return weight;
    }

    public String toString() {
        return String.valueOf(weight);
    }
}
```

Weighted edge: Java implementation

Idiom for processing an edge e: int v = e.either(), w = e.other(v);
Conventions. Allow self-loops and parallel edges.

Edge-weighted graph API

```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]; }

    public Iterable<Edge> edges()
    { return all of this graph's edges.
    }

    public int V()
    { return number of vertices }

    public int E()
    { return number of edges }

    public String toString()
    { return string representation }
}
```

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]; }

    public Iterable<Edge> edges()
    { return all of this graph's edges.
    }

    public int V()
    { return number of vertices }

    public int E()
    { return number of edges }

    public String toString()
    { return string representation }
}
```

Minimum spanning tree API

```java
public class MST
{
    public MST(EdgeWeightedGraph G)
    { constructor
    }

    public Iterable<Edge> edges()
    { edges in MST
    }

    public double weight()
    { weight of MST
    }
}
```

Q. How to represent the MST?

```
% java MST tinyEWG.txt
0-7 0.16 1-7 0.19 0-2 0.26 2-3 0.27 5-7 0.28 4-5 0.35 6-2 0.40 1.81
```
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
}
```

```java
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.println(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

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. ♦
**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.

**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.
Kruskal’s algorithm

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: proof of correctness

**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 edge $e = v \rightarrow w$ black.
- Cut = set of vertices connected to $v$ (or to $w$) 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?**

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

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;
    private MinPQ<Edge> pq;
    public KruskalMST(EdgeWeightedGraph G) {
        mst = new Queue<Edge>();
        pq = new MinPQ<Edge>(G.edges());
        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.find(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 $O(E \log E)$ time.

**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>del min</td>
<td>E</td>
<td>$\log E$</td>
</tr>
<tr>
<td>union</td>
<td>V</td>
<td>$\log^* V$ †</td>
</tr>
<tr>
<td>find</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 $E \log^* V$.

Prim’s algorithm example

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

**How difficult?**
- $O(E)$ time. try all edges
- $O(V)$ time.
- $O(\log E)$ time. use a priority queue!
- $O(\log^* E)$ time.
- Constant time.

**Proposition.** Prim’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: 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 $e = v \xrightarrow{} w$ 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 $T$

Prim's algorithm: lazy implementation

For each edge $e = v \xrightarrow{} w$ from PQ if $w$ not already in $T$:

add $v$ to $T$.

assumed $G$ is connected

repeatedly delete the min weight edge $e = v \xrightarrow{} w$ from PQ

ignore if both endpoints in $T$

add edge to tree

add v or w to tree

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

public LazyPrimMST(WeightedGraph G)
{
    pq = new MinPQ<Edge>();
    mat = 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;
        mat.enqueue(e);
        if (!marked[v]) visit(G, v);
        if (!marked[w]) visit(G, w);
    }
}

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

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

Prim's algorithm example: lazy implementation
**Proposition.** Lazy Prim’s algorithm computes the MST in time proportional to \( E \log E \) 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>

**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.

**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)`.

**Indexed priority queue implementation**

```java
public class IndexMinPQ<Key extends Comparable<Key>>
{
    public IndexMinPQ(int N)
    {
        // create indexed priority queue with indices 0, 1, ..., N-1
    }

    void insert(int k, Key key)
    {
        // associate key with index k
    }

    void decreaseKey(int k, Key key)
    {
        // decrease the key associated with index k
    }

    boolean contains()
    {
        // is k 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 entries in the priority queue
    }
}
```

**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 \( T \)

**Challenge.** Find min weight edge with exactly one endpoint in \( T \).
Prim's algorithm example: eager implementation

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

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</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_2 V )</td>
<td>( d \log_2 V )</td>
<td>( \log_2 V )</td>
<td>( E \log_2 V )</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>( 1 ) ( ^* )</td>
<td>( \log V ) ( ^* )</td>
<td>( 1 ) ( ^* )</td>
<td>( E + V \log V )</td>
</tr>
</tbody>
</table>

\( ^* \) 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.

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>
</tbody>
</table>
| 20xx | \( E \)    | ???

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 \( \sim c \cdot N \log N \).

Scientific application: clustering

k-clustering. Divide a set of objects 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 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 (stop when $k$ connected components).

Alternate solution. Run Prim’s algorithm and delete $k-1$ max weight edges.
Dendrogram. Tree diagram that illustrates arrangement of clusters.

http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/hierarchical.html

Tumors in similar tissues cluster together.

Reference: Botstein & Brown group