4.3 Minimum Spanning Trees

- Introduction
- Greedy algorithm
- Edge-weighted graph API
- Kruskal's algorithm
- Prim's algorithm
- Context

Def. A spanning tree of $G$ is a subgraph $T$ that is:
- Connected.
- Acyclic.
- Includes all of the vertices.

Def. A minimum spanning tree of $G$ is a subgraph $T$ that is:
- Connected.
- Acyclic.
- Includes all of the vertices.

Def. A spanning tree of $G$ is a subgraph $T$ that is:
- Connected.
- Acyclic.
- Includes all of the vertices.

Def. A minimum spanning tree of $G$ is a subgraph $T$ that is:
- Connected.
- Acyclic.
- Includes all of the vertices.

Minimum spanning tree

graph $G$

not connected

not acyclic

not spanning
Minimum spanning tree

Given: Undirected graph $G$ with positive edge weights (connected).
Goal: Find a min weight spanning tree.

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

Brute force. Try all spanning trees?

Network design

MST of bicycle routes in North Seattle

http://www.flickr.com/photos/55668664@N07

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

Image processing

MST thinning

http://www.flickr.com/photos/materials/1305285465
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).

http://www.cs.wisc.edu/~bp/tupelo/geo/met.html

4.3 Minimum Spanning Trees

Cut property (page 606)

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.

This property leads to a greedy algorithm to find the MST: just do cuts, add min weight edge to growing MST. A rare use of “greedy” in S&W.

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.

\[
\begin{align*}
0 & - 7 & 0.26 \\
2 & - 3 & 0.17 \\
1 & - 7 & 0.19 \\
0 & - 2 & 0.26 \\
5 & - 7 & 0.28 \\
1 & - 3 & 0.29 \\
1 & - 5 & 0.32 \\
2 & - 7 & 0.34 \\
4 & - 5 & 0.35 \\
1 & - 2 & 0.36 \\
4 & - 7 & 0.37 \\
0 & - 4 & 0.38 \\
6 & - 2 & 0.40 \\
3 & - 6 & 0.52 \\
6 & - 0 & 0.58 \\
6 & - 4 & 0.93 \\
\end{align*}
\]
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.

\[ \begin{array}{cccc}
0 & 0.35 & 0.37 & 0.38 & 0.39 & 0.40 & 0.41 & 0.42 \\
0 & 0 & 0.35 & 0.37 & 0.38 & 0.39 & 0.40 & 0.41 \\
0 & 0 & 0 & 0.37 & 0.38 & 0.39 & 0.40 & 0.41 \\
0 & 0 & 0 & 0 & 0.38 & 0.39 & 0.40 & 0.41 \\
0 & 0 & 0 & 0 & 0 & 0.39 & 0.40 & 0.41 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.40 & 0.41 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.41 \\
\end{array} \]

A. Q. B. C. D.

Greedy MST algorithm demo result

- 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.

\[ \begin{array}{cccc}
0 & 0.35 & 0.37 & 0.38 & 0.39 & 0.40 & 0.41 & 0.42 \\
0 & 0 & 0.35 & 0.37 & 0.38 & 0.39 & 0.40 & 0.41 \\
0 & 0 & 0 & 0.37 & 0.38 & 0.39 & 0.40 & 0.41 \\
0 & 0 & 0 & 0 & 0.38 & 0.39 & 0.40 & 0.41 \\
0 & 0 & 0 & 0 & 0 & 0.39 & 0.40 & 0.41 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.40 & 0.41 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.41 \\
\end{array} \]

Greedy MST algorithm: correctness proof

**Proposition.** The greedy algorithm computes the MST, for a connected graph.

**Proof.** Any edge colored black is in the MST (via cut property).
- Fewer than \( V - 1 \) black edges \( \Rightarrow \) cut with no black crossing edges. (consider cut whose vertices are any one connected component)

A cut with no black crossing edges

fewer than \( V - 1 \) edges colored black

- Alternative argument to second bullet: It’s easy to show by induction that a spanning tree (minimal or not) has \( V - 1 \) vertices. Thus if there are fewer than \( V - 1 \) black edges, the graph is not spanning, so there are vertices that can’t be reached from other ones: use these to define a cut.

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 holds, but that can be fixed)

Q. What if graph is not connected?
A. Compute minimum spanning forest \( \rightarrow \) MST of each component.

Greedy is good

Gordon Gekko is a composite character in the 1987 film Wall Street and its 2010 sequel Wall Street: Money Never Sleeps, both directed by Oliver Stone. Gekko was portrayed by actor Michael Douglas, whose portrayal was in the words by John Lee and the film itself.
4.3 MINIMUM SPANNING TREES

Weighted edge: Java implementation

```java
public class Edge implements Comparable<Edge> {
    private final int v, w;
    private double weight;
    public Edge(int v, int w, double weight) {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }
    public int compareTo(Edge e) {
        if (this.weight < e.weight) return -1;
        else if (this.weight > e.weight) return 1;
        else return 0;
    }
    public int other(int vertex) {
        if (vertex == v) return w;
        else return e.other(v);
    }
    public int either() {
        return v;
    }
    public String toString() {
        return weight;
    }
}
```

Weighted edge API

```java
public class EdgeWeightedGraph {
    private final int V;
    private final Bag<Edge>[] adj;
    public EdgeWeightedGraph(int V) {
        this.V = V;
        this.adj = new Bag<Edge>[V];
    }
    public void addEdge(Edge e) {
        adj[e.either()].add(e);
    }
    public Iterable<Edge> adj(int v) {
        return this.adj[v];
    }
    public Iterable<Edge> edges() {
        return ListIterable.make(adj, 0, V);
    }
    public String toString() {
        return V + " vertices, " + E + " edges.
    }
}
```

Edge-weighted graph: adjacency-lists representation

```java
import java.util.List;
import java.util.ArrayList;

public class EdgeWeightedGraph {
    private final int V;
    private final Bag<Edge>[] adj;
    public EdgeWeightedGraph(int V) {
        this.V = V;
        this.adj = new Bag<Edge>[V];
    }
    public void addEdge(Edge e) {
        adj[e.either()].add(e);
    }
    public Iterable<Edge> adj(int v) {
        return this.adj[v];
    }
    public String toString() {
        return V + " vertices, " + E + " edges.
    }
}
```

Conventions. Allow self-loops and parallel edges.
How to represent the MST?

**Minimum spanning tree API**

```java
public class MST {
    public MST(EdgeWeightedGraph G) {
        // Constructor...
    }

    public Iterable<Edge> edges() {
        // Method to get edges...
    }

    public double weight() {
        // Method to get weight...
    }
}
```

**Minimum spanning tree API: another S&W pseudo-interface+constructor**

```java
public class MST {
    public MST(EdgeWeightedGraph G) {
        // Constructor...
    }

    public Iterable<Edge> edges() {
        // Method to get edges...
    }

    public double weight() {
        // Method to get weight...
    }
}
```

### Kruskal's algorithm demo

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

#### Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.
Kruskal's algorithm demo
Consider edges in ascending order of weight.
- Add next edge to tree \( T \) unless doing so would create a cycle.

- \( 0-7 \) 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
- \( 2-3 \) 0.34
- \( 4-5 \) 0.35
- \( 1-2 \) 0.36
- \( 4-2 \) 0.37
- \( 0-4 \) 0.38
- \( 6-2 \) 0.40
- \( 3-6 \) 0.52
- \( 6-5 \) 0.58
- \( 6-4 \) 0.93

A minimum spanning tree

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.
- \( C' \) set of vertices connected to \( v \) in tree \( T \).
- No crossing edge is black.
- No crossing edge has lower weight. Why? (because we've already considered the lower-weight edges and either made them into black edges or they are connected two ways to them, so they can't be a crossing edge)

Kruskal's algorithm: implementation challenge

**Challenge.** Would adding edge \( v \rightarrow w \) 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 \( v \rightarrow w \) would create a cycle.
- To add \( v \rightarrow w \) to \( T \), 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>(G.edges());
        UF uf = new UF(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);
            }
        }
        return mst;
    }

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

Kruskal's algorithm: running time

**Proposition.** Kruskal's algorithm computes MST in time proportional to \( E \log V \) (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>

\( \log^* \) is a function defined such that \( \log^* n \) is the number of times \( \log \) must be applied to \( n \) before the result is less than or equal to 1. It is used to count the number of times a logarithm operation is performed.

\( \log^* E \) is used to approximate the time complexity of running Kruskal's algorithm on a universe of edges.

**Remark.** If edges are already sorted, order of growth is \( E \log^* V \).
**4.3 Minimum Spanning Trees**

- Kruskal's algorithm
- Prim's algorithm

**Prim's Algorithm**

- 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.

**Example**

![Graph](image)

**MST edges**

0-7

For more steps, see slide set at Princeton.edu
Prim's algorithm demo

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

![Diagram of a graph with edges and MST edges labeled.](image)

**MST edges**
- 0-7 0.16
- 2-3 0.17
- 1-2 0.19
- 0-2 0.26
- 5-2 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
- 6-6 0.58
- 6-4 0.93

**Hosted graph**
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7

Prim's algorithm: proof of correctness

**Proposition.** (Jarník 1930, Dijkstra 1957, Prim 1959)

**Prim's algorithm computes the MST.**

**Pf.** Prim's algorithm is a special case of the greedy MST algorithm.
- Suppose edge $e = (u, v)$ is added to $T$.
- Cut $=\{v\}$.
- No crossing edge is added to $T$.
- No crossing edge has lower weight.

![Diagram of a graph with edges and a chosen edge labeled.](image)

**Add 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?**
- $E$ (at least)
- $V$ (at most)
- $\log E$ (at most)
- $\log^* E$ (at most)
- $1$

![Diagram of a priority queue with edges and weight.](image)

**1-7 is min weight edge with exactly one endpoint in T**

**4-7 u 0.49**
- 0-7 0.26
- 5-7 0.28
- 2-7 0.34
- 4-7 0.37
- 0-4 0.38
- 6-0 0.58

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 $T$.
- $V$ - edge priority weight of edge.
- Delete-min to determine next edge $e = (u, v)$ to add to $T$.
- Disregard if both endpoints are not in $T$.
- Otherwise, let $u$ be the unmarked vertex (not in $T$):
  - $v$ is an edge incident to $u$ with (assuming other endpoint not in $T$)
  - Add $e$ to $T$ and mark $u$.

![Diagram of a priority queue with edges and weights.](image)

**Add to PQ all edges incident to 0**

0-7 0.16
2-3 0.17
1-2 0.19
0-2 0.26
5-2 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
6-6 0.58
6-4 0.93

Prim's algorithm (lazy) demo

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

![Diagram of a graph with edges.](image)

**Add to PQ all edges incident to 0**

**Priority queue with edges sorted by weight**
- 0-7 0.16
- 0-2 0.26
- 0-4 0.38
- 6-0 0.58
**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.

**Prim's algorithm: lazy implementation demo**

- Add to the PQ all edges incident to 7.

**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.

For more steps, see slides at Princeton.edu, middle part.
Prim's algorithm: lazy implementation

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

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

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

    public Iterator<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).

**Proof.**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Frequency</th>
<th>Data structure</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) 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 IndexedMIntPQ<Key> extends MIntPQ<Key> {
    private int[] index;
    // ... other methods... |
    // Create an indexed priority queue with indices 0, 1, ..., N - 1
    void insert(int i, Key key) {
        // ... implementation... |
    }
    void decreaseKey(int i, Key key) {
        // ... implementation... |
        // ... associate key with index i |
    }
    boolean contains(int i) {
        // ... implementation... |
        // ... remove a minimal key and return associated key |
        // ... associated index |
    }
    int deleteMin() {
        // ... implementation... |
        // ... return smallest key |
        // ... return associated index |
    }
    boolean isEmpty() {
        // ... implementation... |
        // ... return true if the priority queue is empty |
    }
    int size() {
        // ... implementation... |
        // ... return 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 `minHeap`.
- 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 `swap(qp[i])` to implement `decreaseKey(i, key)`.

![Binary heap diagram]

**Prim's algorithm: which priority queue?**

- `unordered array`:
  - `insert`: `V`
  - `delete min`: `V`
  - `decreaseKey`: `V`
  - `total`: `V^2`

- `binary heap`:
  - `log V`
  - `log V`
  - `E log V`

- `d-way heap`:
  - `log V`
  - `log V`
  - `E log V`

- `Fibonacci heap`:
  - `1`
  - `log V`
  - `1`
  - `E + V log V`

**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**

*Algorithms*

**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 ~ `N^2/2` distances and run Prim's algorithm.

**Ingenuity.** Exploit geometry and do it in ~ `ε N log N`.

**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 10^6 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.

Distance between two clusters

Distance between two closest clusters

Tumors in similar tissues cluster together.

Dendrogram of cancers in human

Single-link clustering algorithm

"Well-known" algorithm in science literature for single-link clustering:

1. Form V clusters of one object each.
2. Find the closest pair of objects such that each object is in a different cluster, and merge the two clusters.
3. 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 weighted edges.