Minimum Spanning Tree

Introduction

Application

- 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).

Greedy Algorithm

Cut Property

Step

- start with all edges colored gray
- find cut with no black cossing edges; colors its min-weight edge black
- repeat until V-1 edges are colored black

Edge-weighted graph API

Weighted Edge

API

`public class Edge implements Comparable<Edge>`

	`Edge(int v, int w, double weight)`	*initializing constructor*
`double`	`weight()`	*weight of this edge*
`int`	`either()`	*either of this edge's vertices*
`int`	`other(int v)`	*the other vertex*
`int`	`compareTo(Edge that)`	*compare this edge to e*
`String`	`toString()`	*string representation*

Implementation

```java
import java.util.NoSuchElementException;

public class Edge implements Comparable<Edge> {

    private final int v;
    private final int w;
    private final double weight;

    public Edge(int v, int w, double weight) {
        if (v < 0) throw new IllegalArgumentException("vertex index must be a non-negative integer");
        if (w < 0) throw new IllegalArgumentException("vertex index must be a non-negative integer");
        if (Double.isNaN(weight)) throw new IllegalArgumentException("Weight is NaN");
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public double weight() {
        return weight;
    }

    public int either() {
        return v;
    }

    public int other(int vertex) {
        if      (vertex == v) return w;
        else if (vertex == w) return v;
        else throw new IllegalArgumentException("Illegal endpoint");
    }

    @Override
    public int compareTo(Edge that) {
        return Double.compare(this.weight, that.weight);
    }

    public String toString() {
        return String.format("%d-%d %.5f", v, w, weight);
    }

    public static void main(String[] args) {
        Edge e = new Edge(12, 34, 5.67);
        StdOut.println(e);
    }
}
```

Edge Weighted Graph

API

public class	EdgeWeightedGraph	
	EdgeWeightedGraph(int V)	*create an empty V-vertex graph*
	EdgeWeightedGraph(In in)	*read graph from input stream*
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(Edge e)	*add edge e to this graph*
Iterable<Edge>	adj(int v)	*edges incident to v*
Iterable<Edge>	edges()	*all of this graph's edges*
String	toString()	*string representation*

Adjacency-lists representation

```java
import java.util.NoSuchElementException;

public class EdgeWeightedGraph {
    private static final String NEWLINE = System.getProperty("line.separator");

    private final int V;
    private int E;
    private Bag<Edge>[] adj;

    public EdgeWeightedGraph(int V) {
        if (V < 0) throw new IllegalArgumentException("Number of vertices must be non-negative");
        this.V = V;
        this.E = 0;
        adj = (Bag<Edge>[]) new Bag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new Bag<Edge>();
        }
    }

    public EdgeWeightedGraph(int V, int E) {
        this(V);
        if (E < 0) throw new IllegalArgumentException("Number of edges must be non-negative");
        for (int i = 0; i < E; i++) {
            int v = StdRandom.uniformInt(V);
            int w = StdRandom.uniformInt(V);
            double weight = 0.01 * StdRandom.uniformInt(0, 100);
            Edge e = new Edge(v, w, weight);
            addEdge(e);
        }
    }
```

```java

    public EdgeWeightedGraph(In in) {
        if (in == null) throw new IllegalArgumentException("argument is null");

        try {
            V = in.readInt();
            adj = (Bag<Edge>[]) new Bag[V];
            for (int v = 0; v < V; v++) {
                adj[v] = new Bag<Edge>();
            }

            int E = in.readInt();
            if (E < 0) throw new IllegalArgumentException("Number of edges must be non-negative");
            for (int i = 0; i < E; i++) {
                int v = in.readInt();
                int w = in.readInt();
                validateVertex(v);
                validateVertex(w);
                double weight = in.readDouble();
                Edge e = new Edge(v, w, weight);
                addEdge(e);
            }
        }
        catch (NoSuchElementException e) {
            throw new IllegalArgumentException("invalid input format in EdgeWeightedGraph constructor", e);
        }

    }

    public EdgeWeightedGraph(EdgeWeightedGraph G) {
        this(G.V());
        this.E = G.E();
        for (int v = 0; v < G.V(); v++) {
            // reverse so that adjacency list is in same order as original
            Stack<Edge> reverse = new Stack<Edge>();
            for (Edge e : G.adj[v]) {
                reverse.push(e);
            }
            for (Edge e : reverse) {
                adj[v].add(e);
            }
        }
    }

    public int V() {
        return V;
    }

    public int E() {
```

```
        return E;
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
    }

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

    public Iterable<Edge> adj(int v) {
        validateVertex(v);
        return adj[v];
    }

    public int degree(int v) {
        validateVertex(v);
        return adj[v].size();
    }

    public Iterable<Edge> edges() {
        Bag<Edge> list = new Bag<Edge>();
        for (int v = 0; v < V; v++) {
            int selfLoops = 0;
            for (Edge e : adj(v)) {
                if (e.other(v) > v) {
                    list.add(e);
                }
                // add only one copy of each self loop (self loops will be consecutive)
                else if (e.other(v) == v) {
                    if (selfLoops % 2 == 0) list.add(e);
                    selfLoops++;
                }
            }
        }
        return list;
    }

    public String toString() {
        StringBuilder s = new StringBuilder();
        s.append(V + " " + E + NEWLINE);
```

```java
        for (int v = 0; v < V; v++) {
            s.append(v + ": ");
            for (Edge e : adj[v]) {
                s.append(e + "  ");
            }
            s.append(NEWLINE);
        }
        return s.toString();
    }

    public static void main(String[] args) {
        In in = new In(args[0]);
        EdgeWeightedGraph G = new EdgeWeightedGraph(in);
        StdOut.println(G);
    }
}
```

Kruskal's Algorithm

> $ElogV$

Basic

- consider edges in ascending order of weight
- add next edge to tree T unless doing so would create a cycle

Implementation

```java
import java.util.Arrays;

public class KruskalMST {
    private static final double FLOATING_POINT_EPSILON = 1.0E-12;

    private double weight;                          // weight of MST
    private Queue<Edge> mst = new Queue<Edge>();  // edges in MST

    public KruskalMST(EdgeWeightedGraph G) {

        // create array of edges, sorted by weight
        Edge[] edges = new Edge[G.E()];
        int t = 0;
        for (Edge e: G.edges()) {
            edges[t++] = e;
        }
        Arrays.sort(edges);

        // run greedy algorithm
        UF uf = new UF(G.V());
        for (int i = 0; i < G.E() && mst.size() < G.V() - 1; i++) {
            Edge e = edges[i];
            int v = e.either();
```

```
            int w = e.other(v);

            // v-w does not create a cycle
            if (uf.find(v) != uf.find(w)) {
                uf.union(v, w);     // merge v and w components
                mst.enqueue(e);     // add edge e to mst
                weight += e.weight();
            }
        }

        // check optimality conditions
        assert check(G);
    }

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

    public double weight() {
        return weight;
    }

    // check optimality conditions (takes time proportional to E V lg* V)
    private boolean check(EdgeWeightedGraph G) {

        // check total weight
        double total = 0.0;
        for (Edge e : edges()) {
            total += e.weight();
        }
        if (Math.abs(total - weight()) > FLOATING_POINT_EPSILON) {
            System.err.printf("Weight of edges does not equal weight(): %f vs. %f\n", total, weight());
            return false;
        }

        // check that it is acyclic
        UF uf = new UF(G.V());
        for (Edge e : edges()) {
            int v = e.either(), w = e.other(v);
            if (uf.find(v) == uf.find(w)) {
                System.err.println("Not a forest");
                return false;
            }
            uf.union(v, w);
        }

        // check that it is a spanning forest
        for (Edge e : G.edges()) {
            int v = e.either(), w = e.other(v);
            if (uf.find(v) != uf.find(w)) {
                System.err.println("Not a spanning forest");
```

```
                return false;
            }
        }

        // check that it is a minimal spanning forest (cut optimality conditions)
        for (Edge e : edges()) {

            // all edges in MST except e
            uf = new UF(G.V());
            for (Edge f : mst) {
                int x = f.either(), y = f.other(x);
                if (f != e) uf.union(x, y);
            }

            // check that e is min weight edge in crossing cut
            for (Edge f : G.edges()) {
                int x = f.either(), y = f.other(x);
                if (uf.find(x) != uf.find(y)) {
                    if (f.weight() < e.weight()) {
                        System.err.println("Edge " + f + " violates cut optimality conditions");
                        return false;
                    }
                }
            }

        }

        return true;
    }

    public static void main(String[] args) {
        In in = new In(args[0]);
        EdgeWeightedGraph G = new EdgeWeightedGraph(in);
        KruskalMST mst = new KruskalMST(G);
        for (Edge e : mst.edges()) {
            StdOut.println(e);
        }
        StdOut.printf("%.5f\n", mst.weight());
    }

}
```

Prim's Algorithm

Basic

- 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

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 $e = v - w$ to add to T
  - Disregard if both endpoints v and w are in T
  - Otherwise, let w be the vertex not in T
    - add to PQ any edge incident to w (assuming other endpoint not in T)
    - add w to T
- complexity

operation	frequency	binary heap
delete min	E	log E
insert	E	log E

```
public class LazyPrimMST {
    private static final double FLOATING_POINT_EPSILON = 1.0E-12;

    private double weight;       // total weight of MST
    private Queue<Edge> mst;     // edges in the MST
    private boolean[] marked;    // marked[v] = true iff v on tree
    private MinPQ<Edge> pq;      // edges with one endpoint in tree

    public LazyPrimMST(EdgeWeightedGraph G) {
        mst = new Queue<Edge>();
        pq = new MinPQ<Edge>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)     // run Prim from all vertices to
            if (!marked[v]) prim(G, v);     // get a minimum spanning forest

        // check optimality conditions
        assert check(G);
    }

    // run Prim's algorithm
    private void prim(EdgeWeightedGraph G, int s) {
        scan(G, s);
```

```
        while (!pq.isEmpty()) {                        // better to stop when mst has V-1 edges
            Edge e = pq.delMin();                      // smallest edge on pq
            int v = e.either(), w = e.other(v);        // two endpoints
            assert marked[v] || marked[w];
            if (marked[v] && marked[w]) continue;      // lazy, both v and w already scanned
            mst.enqueue(e);                            // add e to MST
            weight += e.weight();
            if (!marked[v]) scan(G, v);               // v becomes part of tree
            if (!marked[w]) scan(G, w);               // w becomes part of tree
        }
    }

    // add all edges e incident to v onto pq if the other endpoint has not yet been scanned
    private void scan(EdgeWeightedGraph G, int v) {
        assert !marked[v];
        marked[v] = true;
        for (Edge e : G.adj(v))
            if (!marked[e.other(v)]) pq.insert(e);
    }

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

    public double weight() {
        return weight;
    }

    // check optimality conditions (takes time proportional to E V lg* V)
    private boolean check(EdgeWeightedGraph G) {

        // check weight
        double totalWeight = 0.0;
        for (Edge e : edges()) {
            totalWeight += e.weight();
        }
        if (Math.abs(totalWeight - weight()) > FLOATING_POINT_EPSILON) {
            System.err.printf("Weight of edges does not equal weight(): %f vs. %f\n", totalWeight, weight());
            return false;
        }

        // check that it is acyclic
        UF uf = new UF(G.V());
        for (Edge e : edges()) {
            int v = e.either(), w = e.other(v);
            if (uf.find(v) == uf.find(w)) {
```

```
                System.err.println("Not a forest");
                return false;
            }
            uf.union(v, w);
        }

        // check that it is a spanning forest
        for (Edge e : G.edges()) {
            int v = e.either(), w = e.other(v);
            if (uf.find(v) != uf.find(w)) {
                System.err.println("Not a spanning forest");
                return false;
            }
        }

        // check that it is a minimal spanning forest (cut optimality conditions)
        for (Edge e : edges()) {

            // all edges in MST except e
            uf = new UF(G.V());
            for (Edge f : mst) {
                int x = f.either(), y = f.other(x);
                if (f != e) uf.union(x, y);
            }

            // check that e is min weight edge in crossing cut
            for (Edge f : G.edges()) {
                int x = f.either(), y = f.other(x);
                if (uf.find(x) != uf.find(y)) {
                    if (f.weight() < e.weight()) {
                        System.err.println("Edge " + f + " violates cut optimality conditions");
                        return false;
                    }
                }
            }

        }

        return true;
    }

    public static void main(String[] args) {
        In in = new In(args[0]);
        EdgeWeightedGraph G = new EdgeWeightedGraph(in);
        LazyPrimMST mst = new LazyPrimMST(G);
        for (Edge e : mst.edges()) {
            StdOut.println(e);
        }
        StdOut.printf("%.5f\n", mst.weight());
    }
```

```
}
```

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 =weight of shortest edge connecting v to T
    - Delete-min vertex v and add its associated edge $e=v-w$ to T
    - Update PQ by considering all edges $e=v-x$ incident to v
        - ignore if x is already in T
        - add x to PQ if not already on it
        - decrease priority of x if $v-x$ becomes shortest edge connecting to T

```java
public class PrimMST {
    private static final double FLOATING_POINT_EPSILON = 1.0E-12;

    private Edge[] edgeTo;        // edgeTo[v] = shortest edge from tree vertex to non-tree vertex
    private double[] distTo;      // distTo[v] = weight of shortest such edge
    private boolean[] marked;     // marked[v] = true if v on tree, false otherwise
    private IndexMinPQ<Double> pq;

    public PrimMST(EdgeWeightedGraph G) {
        edgeTo = new Edge[G.V()];
        distTo = new double[G.V()];
        marked = new boolean[G.V()];
        pq = new IndexMinPQ<Double>(G.V());
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;

        for (int v = 0; v < G.V(); v++)      // run from each vertex to find
            if (!marked[v]) prim(G, v);      // minimum spanning forest

        // check optimality conditions
        assert check(G);
    }

    // run Prim's algorithm in graph G, starting from vertex s
    private void prim(EdgeWeightedGraph G, int s) {
        distTo[s] = 0.0;
        pq.insert(s, distTo[s]);
        while (!pq.isEmpty()) {
            int v = pq.delMin();
            scan(G, v);
        }
    }

```

```java

    // scan vertex v
    private void scan(EdgeWeightedGraph G, int v) {
        marked[v] = true;
        for (Edge e : G.adj(v)) {
            int w = e.other(v);
            if (marked[w]) continue;         // v-w is obsolete edge
            if (e.weight() < distTo[w]) {
                distTo[w] = e.weight();
                edgeTo[w] = e;
                if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
                else                pq.insert(w, distTo[w]);
            }
        }
    }

    public Iterable<Edge> edges() {
        Queue<Edge> mst = new Queue<Edge>();
        for (int v = 0; v < edgeTo.length; v++) {
            Edge e = edgeTo[v];
            if (e != null) {
                mst.enqueue(e);
            }
        }
        return mst;
    }
}
```

MST Context

- Does a linear-time MST exists?

year	worst case	discovered by
1975	$E \log \log V$	Yao
1976	$E \log \log V$	Cheriton-Tarjan
1984	$E \log^* V$, $E + V \log V$	Fredman-Tarjan
1986	$E \log (\log^* V)$	Gabow-Galil-Spencer-Tarjan
1997	$E\, \alpha(V) \log \alpha(V)$	Chazelle
2000	$E\, \alpha(V)$	Chazelle
2002	optimal	Pettie-Ramachandran
20xx	E	???

- Single-link clustering algorithm