# Minimum Spanning Trees

## Outline

# What are Minimum Spanning Trees?

## What are Minimum Spanning Trees?

A spanning tree of a graph is a connected subgraph with no cycles that includes all the vertices

## What are Minimum Spanning Trees?

A spanning tree of a graph is a connected subgraph with no cycles that includes all the vertices

A minimum spanning tree (MST) of an edge-weighted undirected graph is a spanning tree whose weight (the sum of the weights of its edges) is no larger than the weight of any other spanning tree

## What are Minimum Spanning Trees?

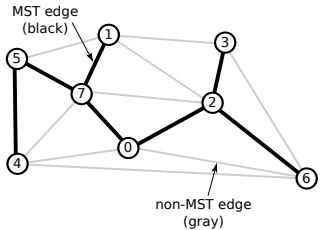
A spanning tree of a graph is a connected subgraph with no cycles that includes all the vertices

A minimum spanning tree (MST) of an edge-weighted undirected graph is a spanning tree whose weight (the sum of the weights of its edges) is no larger than the weight of any other spanning tree

An edge-weighted graph and its MST

```
>_ ~/workspace/dsa/programs

$ more ../data/tinyEWG.txt
8
16
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
```



# What are Minimum Spanning Trees?

## What are Minimum Spanning Trees?

If the graph is connected and the edge weights are unique, then MST exists and is unique

## What are Minimum Spanning Trees?

If the graph is connected and the edge weights are unique, then MST exists and is unique

Goal: given a connected undirected graph $G$ with arbitrary (but distinct) edge weights, find the MST of $G$

## What are Minimum Spanning Trees?

If the graph is connected and the edge weights are unique, then MST exists and is unique

Goal: given a connected undirected graph $G$ with arbitrary (but distinct) edge weights, find the MST of $G$

Typical MST applications

| Application | Vertex | Edge |
|---|---|---|
| circuit | component | wire |
| airline | airport | flight route |
| power distribution | power plant | transmission lines |
| image analysis | feature | proximity relationship |

# Edge-Weighted Graph API

# Edge-Weighted Graph API

| EdgeWeightedGraph | |
|---|---|
| `EdgeWeightedGraph(int V)` | create an empty graph with $V$ vertices |
| `EdgeWeightedGraph(In in)` | create a graph from input stream |
| `int V()` | number of vertices |
| `int E()` | number of edges |
| `void addEdge(Edge e)` | add weighted edge $e$ to this graph |
| `Iterable<Edge> adj(int v)` | edges incident to $v$ |
| `Iterable<Edge> edges()` | all edges in this graph |

# Edge-Weighted Graph API

| EdgeWeightedGraph | |
|---|---|
| `EdgeWeightedGraph(int V)` | create an empty graph with $V$ vertices |
| `EdgeWeightedGraph(In in)` | create a graph from input stream |
| `int V()` | number of vertices |
| `int E()` | number of edges |
| `void addEdge(Edge e)` | add weighted edge $e$ to this graph |
| `Iterable<Edge> adj(int v)` | edges incident to $v$ |
| `Iterable<Edge> edges()` | all edges in this graph |

| Edge | |
|---|---|
| `Edge(int v, int w, double weight)` | create a weighted edge $v$-$w$ |
| `double weight()` | edge weight |
| `int either()` | either endpoint |
| `int other(int v)` | the endpoint that's not $v$ |

# Edge-Weighted Graph API

# Edge-Weighted Graph API

EdgeWeightedGraph.java

```
package dsa;

import stdlib.In;
import stdlib.StdOut;

public class EdgeWeightedGraph {
    private LinkedBag<Edge>[] adj;
    private int V;
    private int E;

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

    public EdgeWeightedGraph(In in) {
        this(in.readInt());
        int E = in.readInt();
        for (int i = 0; i < E; i++) {
            int v = in.readInt();
            int w = in.readInt();
            double weight = in.readDouble();
            addEdge(new Edge(v, w, weight));
        }
    }

    public int V() {
        return V;
    }

    public int E() {
```

# Edge-Weighted Graph API

EdgeWeightedGraph.java

```
        return E;
    }

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

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

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

    public Iterable<Edge> edges() {
        LinkedBag<Edge> edges = new LinkedBag<Edge>();
        for (int v = 0; v < V; v++) {
            int selfLoops = 0;
            for (Edge e : adj(v)) {
                if (e.other(v) > v) {
                    edges.add(e);
                } else if (e.other(v) == v) {
                    if (selfLoops % 2 == 0) {
                        edges.add(e);
                    }
                    selfLoops++;
                }
            }
        }
        return edges;
```

# Edge-Weighted Graph API

EdgeWeightedGraph.java

```
    }

    public String toString() {
        StringBuilder s = new StringBuilder();
        s.append(V + " " + E + "\n");
        for (int v = 0; v < V; v++) {
            s.append(v + ": ");
            for (Edge e : adj[v]) {
                s.append(e + "  ");
            }
            s.append("\n");
        }
        return s.toString().strip();
    }

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

class Edge implements Comparable<Edge> {
    private int v;
    private int w;
    private double weight;

    public Edge(int v, int w, double weight) {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public int either() {
        return v;
```

# Edge-Weighted Graph API

EdgeWeightedGraph.java

```
    }

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

    public double weight() {
        return weight;
    }

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

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

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

# Minimum Spanning Tree API

# Minimum Spanning Tree API

| Kruskal | |
|---|---|
| `Kruskal(EdgeWeightedGraph G)` | constructor |
| `Iterable<Edge> edges()` | all of the MST edges |
| `double weight()` | weight of MST |

# Minimum Spanning Tree API

**Kruskal**

| | |
|---|---|
| `Kruskal(EdgeWeightedGraph G)` | constructor |
| `Iterable<Edge> edges()` | all of the MST edges |
| `double weight()` | weight of MST |

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

# Minimum Spanning Tree API

Kruskal

| | |
|---|---|
| `Kruskal(EdgeWeightedGraph G)` | constructor |
| `Iterable<Edge> edges()` | all of the MST edges |
| `double weight()` | weight of MST |

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

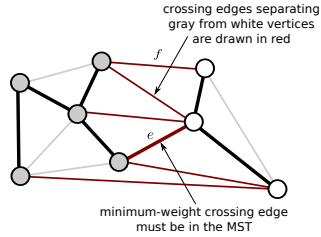
>_ ~/workspace/dsa/programs

```
$ java dsa.Kruskal ../data/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
```

# Greedy Algorithm

# Greedy Algorithm

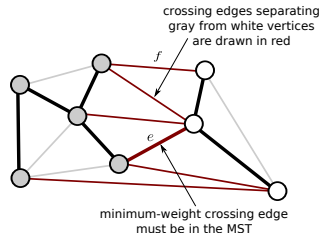
A cut of a graph is a partition of its vertices into two nonempty disjoint sets



# Greedy Algorithm

A cut of a graph is a partition of its vertices into two nonempty disjoint sets

A crossing edge of a cut is an edge that connects a vertex in one set with a vertex in the other

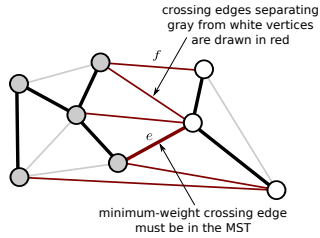


# Greedy Algorithm

A cut of a graph is a partition of its vertices into two nonempty disjoint sets

A crossing edge of a cut is an edge that connects a vertex in one set with a vertex in the other

Cut property: given any cut, a crossing edge of minimum weight is in the MST

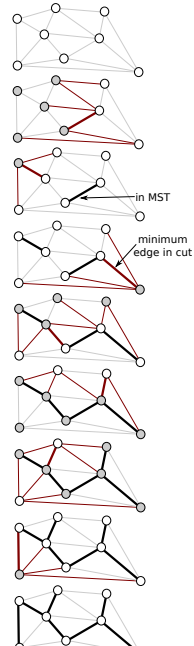


# Greedy Algorithm

## Greedy Algorithm

Greedy MST algorithm

- Start with all edges colored gray
- Find cut with no black edges and color its minimum-weight edge black
- Repeat until $V - 1$ edges are colored black

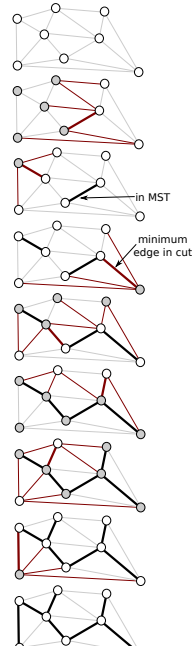


## Greedy Algorithm

Greedy MST algorithm

- Start with all edges colored gray
- Find cut with no black edges and color its minimum-weight edge black
- Repeat until $V - 1$ edges are colored black

The greedy algorithm computes the MST



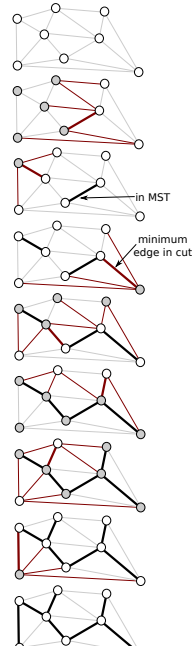
## Greedy Algorithm

Greedy MST algorithm

- Start with all edges colored gray
- Find cut with no black edges and color its minimum-weight edge black
- Repeat until $V - 1$ edges are colored black

The greedy algorithm computes the MST

Kruskal's algorithm that we consider next chooses cuts and minimum-weight edges efficiently



## Greedy Algorithm

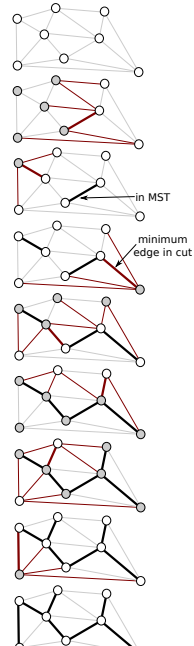
Greedy MST algorithm

- Start with all edges colored gray
- Find cut with no black edges and color its minimum-weight edge black
- Repeat until $V - 1$ edges are colored black

The greedy algorithm computes the MST

Kruskal's algorithm that we consider next chooses cuts and minimum-weight edges efficiently

Add edges to tree $T$ in ascending order of weight unless doing so would create a cycle



## Greedy Algorithm

Greedy MST algorithm

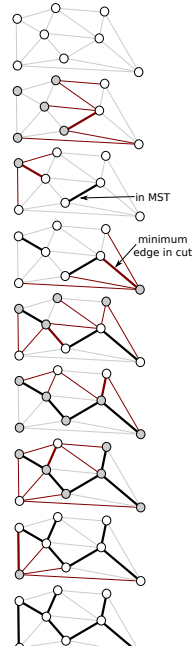
- Start with all edges colored gray
- Find cut with no black edges and color its minimum-weight edge black
- Repeat until $V - 1$ edges are colored black

The greedy algorithm computes the MST

Kruskal's algorithm that we consider next chooses cuts and minimum-weight edges efficiently

Add edges to tree $T$ in ascending order of weight unless doing so would create a cycle

Kruskal's algorithm computes MST in time proportional to $E \log E$ in the worst case



# Kruskal's Algorithm

# Kruskal's Algorithm

Kruskal.java

```
package dsa;

import stdlib.In;
import stdlib.StdOut;

public class Kruskal {
    private LinkedQueue<Edge> mst = new LinkedQueue<Edge>();
    private double weight;

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

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

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

# Kruskal's Algorithm

Kruskal.java

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

# Kruskal's Algorithm

# Kruskal's Algorithm

## Trace