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.

![Graph G](image)

---

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.

![Not connected](image)
**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.

---

**Network design**

MST of bicycle routes in North Seattle

![MST of bicycle routes in North Seattle](http://www.flickr.com/photos/owedistrict/21980840)

---

**Models of nature**

MST of random graph

![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

Medical image processing

MST dithering

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, cable, computer, road).

Minimum Spanning Trees

- Greedy algorithm
- Edge-weighted graph API
- Kruskal’s algorithm
- Prim’s algorithm
- Context
**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.

**Greedy MST algorithm**

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

**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. □
Greedy MST algorithm

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

MST edges
0-2

Greedy MST algorithm

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

MST edges
0-2

Greedy MST algorithm

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

MST edges
0-2  5-7

Greedy MST algorithm

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

MST edges
0-2  5-7

Greedy MST algorithm

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

crossing edges
(sorted by weight)

in MST
5-7  0.28
1-5  0.32
4-5  0.35

Greedy MST algorithm

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

crossing edges
(sorted by weight)

in MST
6-2  0.40
3-6  0.52
6-0  0.58
6-4  0.93

Greedy MST algorithm

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

MST edges
0-2  5-7

Greedy MST algorithm

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

MST edges
0-2  5-7

Greedy MST algorithm

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

MST edges
0-2  5-7

Greedy MST algorithm

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

crossing edges
(sorted by weight)

in MST
5-7  0.28
1-5  0.32
4-5  0.35

Greedy MST algorithm

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

MST edges
0-2  5-7

Greedy MST algorithm

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

MST edges
0-2  5-7

Greedy MST algorithm

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

MST edges
0-2  5-7
Greedy MST algorithm

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

```
MST edges
0-2  5-7  6-2
```

Greedy MST algorithm

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

```
MST edges
0-2  5-7  6-2
```

Greedy MST algorithm

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

```
MST edges
0-2  5-7  6-2
```

Greedy MST algorithm

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

```
MST edges
0-2  5-7  6-2  0-7
```

Greedy MST algorithm

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

```
MST edges
0-2  5-7  6-2
```
Greedy MST algorithm

• Start with all edges colored gray.
• Find a cut with no black crossing edges, and color its min-weight edge black.
• Repeat until \( V - 1 \) edges are colored black.

![Diagram of Greedy MST algorithm](image1)

MST edges

- 0-2
- 5-7
- 6-2
- 0-7
- 2-3

Greedy MST algorithm

• Start with all edges colored gray.
• Find a cut with no black crossing edges, and color its min-weight edge black.
• Repeat until \( V - 1 \) edges are colored black.

![Diagram of Greedy MST algorithm](image2)

MST edges

- 0-2
- 5-7
- 6-2
- 0-7
- 2-3

Greedy MST algorithm

• Start with all edges colored gray.
• Find a cut with no black crossing edges, and color its min-weight edge black.
• Repeat until \( V - 1 \) edges are colored black.

![Diagram of Greedy MST algorithm](image3)

MST edges

- 0-2
- 5-7
- 6-2
- 0-7
- 2-3

Greedy MST algorithm

• Start with all edges colored gray.
• Find a cut with no black crossing edges, and color its min-weight edge black.
• Repeat until \( V - 1 \) edges are colored black.

![Diagram of Greedy MST algorithm](image4)

MST edges

- 0-2
- 5-7
- 6-2
- 0-7
- 2-3

Crossing edges (sorted by weight)

- 1-7: 0.19
- 1-3: 0.29
- 1-5: 0.32
- 4-5: 0.35
- 1-2: 0.36
- 4-7: 0.37
- 0-4: 0.38
- 6-4: 0.93

Min-weight crossing edge

- 1-7
- 0.19
- 1-3
- 0.29
- 1-5
- 0.32
- 4-5
- 0.35
- 1-2
- 0.36
- 4-7
- 0.37
- 0-4
- 0.38
- 6-4
- 0.93
**Greedy MST algorithm**

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

![MST edges](image)

**Greedy MST algorithm: correctness proof**

**Proposition.** The greedy algorithm computes the MST.

**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)

**Greedy MST algorithm: efficient implementations**

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

**Efficient implementations.** Choose cut? 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.
MINIMUM SPANNING TREES

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

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 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;
    }
    
    public int V() { return v; }
    public int E() { return w; }
    public String toString() { return String.valueOf(weight); }
}
```

Idiom for processing an edge `e`:

```java
int v = e.either(), w = e.other(v);
```

Weighted edge: Java implementation

```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 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;
    }
    
    public int V() { return v; }
    public int E() { return w; }
    public String toString() { return String.valueOf(weight); }
}
```

Edge-weighted graph API

```java
public class EdgeWeightedGraph {
    private final int V;
    private final int E;
    
    public EdgeWeightedGraph(int V) {
        this.V = V;
        this.E = 0;
    }
    
    public EdgeWeightedGraph(In in) {
        int V = in.readInt();
        int E = in.readInt();
        this.V = V;
        this.E = E;
        
        for (int i = 0; i < E; i++) {
            int u = in.readInt();
            int w = in.readInt();
            double weight = in.readDouble();
            addEdge(new Edge(u, w, weight));
        }
    }
    
    public void addEdge(Edge e) {
        int u = e.one();
        int v = e.other(u);
        
        if (adj.contains(u)) {
            for (Edge e2 : adj(u)) {
                if (e2.equals(e)) return;
            }
        }
        
        adj.add(u);
        
        if (adj.contains(v)) {
            for (Edge e2 : adj(v)) {
                if (e2.equals(e)) return;
            }
        }
        
        adj.add(v);
        
        int w = e.weight();
        
        E = E + 1;
    }
    
    public Iterable<Edge> adj(int v) {
        return new List() {
            @Override
            public Iterator<Edge> iterator() {
                return new Iterator<Edge>() {
                    int i = 0;
                    int limit = adj.size();
                    
                    @Override
                    public boolean hasNext() {
                        return i < limit;
                    }
                    
                    @Override
                    public Edge next() {
                        Edge e = adj.get(i);
                        return e;
                    }
                    
                    @Override
                    public void remove() {
                        throw new UnsupportedOperationException();
                    }
                }
            }
        };
    }
    
    public Iterable<Edge> edges() {
        return new List() {
            @Override
            public Iterator<Edge> iterator() {
                return new Iterator<Edge>() {
                    int i = 0;
                    int limit = E;
                    
                    @Override
                    public boolean hasNext() {
                        return i < limit;
                    }
                    
                    @Override
                    public Edge next() {
                        Edge e = edges.get(i);
                        return e;
                    }
                    
                    @Override
                    public void remove() {
                        throw new UnsupportedOperationException();
                    }
                }
            }
        };
    }
    
    public int V() { return V; }
    public int E() { return E; }
    public String toString() { return String.valueOf(V) + " vertices, " + String.valueOf(E) + " edges;"; }
}
```

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

Maintain vertex-indexed array of edge lists.

```
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]; }
}
```

**Minimum spanning tree API**

Q. How to represent the MST?

```
public class MST {
    MST(EdgeWeightedGraph G) {
        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]; }
}
```

```
% 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
```
Kruskal’s algorithm

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

In the example graph, edges are sorted by weight:

- 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
- 2-7 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

In the minimum spanning tree (MST), the edges 0-7, 2-3, 1-7, 0-2, 5-7, 1-3, and 1-5 are considered.

- 0-7 does not create a cycle.
- 2-3 creates a cycle.

The edges 0-7, 2-3, 1-7, 0-2, 5-7, 1-3, and 1-5 are added to the MST.

- 0-7
- 2-3
- 1-7
- 0-2
- 5-7
- 1-3
- 1-5

In the MST, edge 2-3 creates a cycle.
Kruskal's algorithm

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

- Consider edges in ascending order of weight.
- Add next edge to tree $T$ unless doing so would create a cycle.
Kruskal’s algorithm

• Consider edges in ascending order of weight.
• Add next edge to tree $T$ unless doing so would create a cycle.
Kruskal’s algorithm

- 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
2-7 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

Kruskal’s algorithm: visualization

creates a cycle
not in MST

Kruskal’s algorithm

- 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
2-7 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

Kruskal’s algorithm: visualization
**Kruskal’s algorithm: correctness proof**

**Proposition.** [Kruskal 1956] Kruskal’s algorithm computes the MST.

**Pf.**

1. Suppose Kruskal’s algorithm colors the edge $e = v \rightarrow w$ black.
2. Cut = set of vertices connected to $v$ in tree $T$.
3. No crossing edge is black.
4. No crossing edge has lower weight. Why?

**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: implementation challenge**

**Challenge.**

Would adding edge $v \rightarrow w$ to tree $T$ create a cycle? If not, add it.

**How difficult?**

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

**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>();
        for (Edge e : G.edges())
            pq.insert(e);
        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);
            }
        }
    }
    public Iterable<Edge> edges()
    {  return mst;  }
}
```

**Build priority queue**
- greedily add edges to MST
- edge $v \rightarrow w$ does not create cycle
- merge sets
- add edge to MST

**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: running time**

**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 $E \log^* V$.

---

**Minimum Spanning Trees**

- Greedy algorithm
- Edge-weighted graph API
- Kruskal’s algorithm
- Prim’s algorithm
- Context

**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.

![Prim's algorithm example](image)

---

![Minimum Spanning Trees](image)
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.

**MST edges**

0-7

**min weight edge with exactly one endpoint in T**

edges with exactly one endpoint in T (sorted by weight)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td>0.16</td>
</tr>
<tr>
<td>0-2</td>
<td>0.26</td>
</tr>
<tr>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>6-0</td>
<td>0.58</td>
</tr>
</tbody>
</table>

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.

**MST edges**

0-7

**min weight edge with exactly one endpoint in T**

edges with exactly one endpoint in T (sorted by weight)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-7</td>
<td>0.19</td>
</tr>
<tr>
<td>0-2</td>
<td>0.26</td>
</tr>
<tr>
<td>5-7</td>
<td>0.28</td>
</tr>
<tr>
<td>2-7</td>
<td>0.34</td>
</tr>
<tr>
<td>4-7</td>
<td>0.37</td>
</tr>
<tr>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>6-0</td>
<td>0.58</td>
</tr>
</tbody>
</table>

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.

**MST edges**

0-7

**min weight edge with exactly one endpoint in T**

edges with exactly one endpoint in T (sorted by weight)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-7</td>
<td>0.19</td>
</tr>
<tr>
<td>0-2</td>
<td>0.26</td>
</tr>
<tr>
<td>5-7</td>
<td>0.28</td>
</tr>
<tr>
<td>2-7</td>
<td>0.34</td>
</tr>
<tr>
<td>4-7</td>
<td>0.37</td>
</tr>
<tr>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>6-0</td>
<td>0.58</td>
</tr>
</tbody>
</table>
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.

- 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.

- 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.

- 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**

- 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.

```
min weight edge with exactly one endpoint in T
```

```
\begin{array}{c}
\text{in MST} \\
5 - 7 & 0.28 \\
1 - 5 & 0.32 \\
4 - 7 & 0.37 \\
0 - 4 & 0.38 \\
6 - 2 & 0.40 \\
3 - 6 & 0.52 \\
6 - 0 & 0.58 \\
\end{array}
```

```
MST edges \\
0 - 7 \\
1 - 7 \\
0 - 2 \\
2 - 3 \\
```

**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.

```
min weight edge with exactly one endpoint in T
```

```
\begin{array}{c}
\text{in MST} \\
5 - 7 & 0.28 \\
1 - 5 & 0.32 \\
4 - 7 & 0.37 \\
0 - 4 & 0.38 \\
6 - 2 & 0.40 \\
3 - 6 & 0.52 \\
6 - 0 & 0.58 \\
\end{array}
```

```
MST edges \\
0 - 7 \\
1 - 7 \\
0 - 2 \\
2 - 3 \\
```

**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.

```
min weight edge with exactly one endpoint in T
```

```
\begin{array}{c}
\text{in MST} \\
4 - 5 & 0.35 \\
4 - 7 & 0.37 \\
0 - 4 & 0.38 \\
6 - 2 & 0.40 \\
3 - 6 & 0.52 \\
6 - 0 & 0.58 \\
\end{array}
```

```
MST edges \\
0 - 7 \\
1 - 7 \\
0 - 2 \\
2 - 3 \\
5 - 7 \\
4 - 5 \\
```

**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.

```
min weight edge with exactly one endpoint in T
```

```
\begin{array}{c}
\text{in MST} \\
4 - 5 & 0.35 \\
4 - 7 & 0.37 \\
0 - 4 & 0.38 \\
6 - 2 & 0.40 \\
3 - 6 & 0.52 \\
6 - 0 & 0.58 \\
\end{array}
```

```
MST edges \\
0 - 7 \\
1 - 7 \\
0 - 2 \\
2 - 3 \\
5 - 7 \\
4 - 5 \\
```

**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.

```
min weight edge with exactly one endpoint in T
```

```
\begin{array}{c}
\text{in MST} \\
4 - 5 & 0.35 \\
4 - 7 & 0.37 \\
0 - 4 & 0.38 \\
6 - 2 & 0.40 \\
3 - 6 & 0.52 \\
6 - 0 & 0.58 \\
\end{array}
```

```
MST edges \\
0 - 7 \\
1 - 7 \\
0 - 2 \\
2 - 3 \\
5 - 7 \\
4 - 5 \\
```
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.

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 $e = \text{min weight edge connecting a vertex on the tree to a vertex not on the tree}$.
- Cut $= \text{set of vertices connected on tree}$.
- No crossing edge is black.
- No crossing edge has lower weight.

Prim's algorithm: visualization

Prim's algorithm: proof of correctness
**Prim's algorithm: implementation challenge**

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

**How difficult?**
- $E$  
- $V$  
- $\log E$  
- $\log^* E$  
- $I$

**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 $e = v \rightarrow w$ 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**

- 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

- 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.

add to PQ all edges incident to 0

edges on PQ (sorted by weight)
* 0-7 0.16
* 0-2 0.26
* 0-4 0.38
* 6-0 0.58

Prim's algorithm - Lazy implementation

- 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.

delete 0–7 and add to MST

add to PQ all edges incident to 7

edges on PQ (sorted by weight)
* 1-7 0.19
* 0-2 0.26
* 5-7 0.28
* 2-7 0.34
* 4-7 0.37
* 0-4 0.38
* 6-0 0.58

MST edges
0–7
Prim's algorithm - Lazy implementation

- 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.

```
delete 1−7 and add to MST
```

```
MST edges
0−7
```

```
Prim's algorithm - Lazy implementation

- 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.

```
add to PQ all edges incident to 1
```

```
edges on PQ (sorted by weight)

1−7  0.19
0−2  0.26
5−7  0.28
2−7  0.34
4−7  0.37
0−4  0.38
6−0  0.58
```

```
MST edges
0−7  1−7
```

```
Prim's algorithm - Lazy implementation

- 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.

```
delete edge 0−2 and add to MST
```

```
edges on PQ (sorted by weight)

0−2  0.26
5−7  0.28
2−7  0.34
4−7  0.37
0−4  0.38
6−0  0.58
```

```
MST edges
0−7  1−7
```

```
Prim's algorithm - Lazy implementation

- 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

- 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.

```
delete edge 0−2 and add to MST
```

```
edges on PQ (sorted by weight)

0−2  0.26
5−7  0.28
2−7  0.34
4−7  0.37
0−4  0.38
6−0  0.58
```

```
MST edges
0−7  1−7
```
• 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**

1. Start with vertex 0 and greedily grow tree $T$.
2. Add to $T$ the min weight edge with exactly one endpoint in $T$.
3. Repeat until $V-1$ edges.

MST edges
0-7 1-7 0-2

Edges on PQ (sorted by weight)
5-7 0.28
1-3 0.29
1-5 0.32
2-7 0.34
1-2 0.36
4-7 0.37
0-4 0.38
6-0 0.58

MST edges
0-7 1-7 0-2

Edges on PQ (sorted by weight)
5-7 0.28
1-3 0.29
1-5 0.32
2-7 0.34
1-2 0.36
4-7 0.37
0-4 0.38
6-0 0.58

Edges on PQ (sorted by weight)
5-7 0.28
1-3 0.29
1-5 0.32
2-7 0.34
1-2 0.36
4-7 0.37
0-4 0.38
6-2 0.40
6-0 0.58
Prim's algorithm - Lazy implementation

• 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.

add to PQ all edges incident to 3

![Diagram](image1)

Prim's algorithm - Lazy implementation

• 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.

delete 5–7 and add to MST

![Diagram](image2)

Prim's algorithm - Lazy implementation

• 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.

add to PQ all edges incident to 5

![Diagram](image3)

Prim's algorithm - Lazy implementation

• 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.

![Diagram](image4)
**Prim's algorithm - Lazy implementation**

- 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.

```
data MST edges
0-7 1-7 0-2 2-3 5-7
```

**Lazy implementation**

```
edges on PQ (sorted by weight)
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
```

Delete 1-3 and discard obsolete edge

```
MST edges
0-7 1-7 0-2 2-3 5-7
```

**Prim's algorithm - Lazy implementation**

- 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.

```
data MST edges
0-7 1-7 0-2 2-3 5-7
```

**Lazy implementation**

```
edges on PQ (sorted by weight)
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
```

Delete 1-5 and discard obsolete edge

```
MST edges
0-7 1-7 0-2 2-3 5-7
```

**Lazy implementation**

```
MST edges
0-7 1-7 0-2 2-3 5-7
```

Delete 4-5 and add to MST

```
edges on PQ (sorted by weight)
4-5 0.35
3-6 0.52
6-0 0.58
```

**Prim's algorithm - Lazy implementation**

- 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.

```
data MST edges
0-7 1-7 0-2 2-3 5-7
```

**Lazy implementation**

```
edges on PQ (sorted by weight)
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
```

Delete 2-7 and discard obsolete edge

```
MST edges
0-7 1-7 0-2 2-3 5-7
```

**Lazy implementation**

```
MST edges
0-7 1-7 0-2 2-3 5-7
```
Prim's algorithm - Lazy implementation

• 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.

edges on PQ (sorted by weight)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>0.36</td>
</tr>
<tr>
<td>4-7</td>
<td>0.37</td>
</tr>
<tr>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>6-2</td>
<td>0.40</td>
</tr>
<tr>
<td>3-6</td>
<td>0.52</td>
</tr>
<tr>
<td>6-0</td>
<td>0.58</td>
</tr>
</tbody>
</table>

MST edges

0-7  1-7  0-2  2-3  5-7  4-5

Prim's algorithm - Lazy implementation

• 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.

add to PQ all edges incident to 4

MST edges

0-7  1-7  0-2  2-3  5-7  4-5

Prim's algorithm - Lazy implementation

• 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.

delete 4-7 and discard obsolete edge

edges on PQ (sorted by weight)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-7</td>
<td>0.37</td>
</tr>
<tr>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>6-2</td>
<td>0.40</td>
</tr>
<tr>
<td>3-6</td>
<td>0.52</td>
</tr>
<tr>
<td>6-0</td>
<td>0.58</td>
</tr>
<tr>
<td>6-4</td>
<td>0.93</td>
</tr>
</tbody>
</table>

MST edges

0-7  1-7  0-2  2-3  5-7  4-5

Prim's algorithm - Lazy implementation

• 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.

delete 1-2 and discard obsolete edge

edges on PQ (sorted by weight)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>0.36</td>
</tr>
<tr>
<td>4-7</td>
<td>0.37</td>
</tr>
<tr>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>6-2</td>
<td>0.40</td>
</tr>
<tr>
<td>3-6</td>
<td>0.52</td>
</tr>
<tr>
<td>6-0</td>
<td>0.58</td>
</tr>
<tr>
<td>6-4</td>
<td>0.93</td>
</tr>
</tbody>
</table>

MST edges

0-7  1-7  0-2  2-3  5-7  4-5
**Prim's algorithm - Lazy implementation**

- 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.

Delete 0–4 and discard obsolete edge

**MST edges**
0–7  1–7  0–2  2–3  5–7  4–5

**edges on PQ (sorted by weight)**
0–4  0.38
6–2  0.40
3–6  0.52
6–0  0.58
6–4  0.93

**stop since $V-1$ edges**

**Prim's algorithm - Lazy implementation**

- 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.

Delete 6–2 and add to MST

**MST edges**
0–7  1–7  0–2  2–3  5–7  4–5

**edges on PQ (sorted by weight)**
6–2  0.40
3–6  0.52
6–0  0.58
6–4  0.93

**Prim's algorithm - Lazy implementation**

- 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.

Delete 0–4 and discard obsolete edge

**MST edges**
0–7  1–7  0–2  2–3  5–7  4–5

**edges on PQ (sorted by weight)**
0–4  0.38
6–2  0.40
3–6  0.52
6–0  0.58
6–4  0.93

**stop since $V-1$ edges**

**Prim's algorithm - Lazy implementation**

- 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.

Delete 0–4 and discard obsolete edge

**MST edges**
0–7  1–7  0–2  2–3  5–7  4–5

**edges on PQ (sorted by weight)**
0–4  0.38
6–2  0.40
3–6  0.52
6–0  0.58
6–4  0.93

**stop since $V-1$ edges**
Prim’s algorithm - Lazy implementation

- 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.

```
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

for each edge e = v – w, add to PQ if w not already in T
add v to T
add edge e to tree
add v or w to tree

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>

MST edges

0-7 1-7 0-2 2-3 5-7 4-5 6-2

Lazy Prim’s algorithm: running time
**Prim’s algorithm: 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 \( e = v \to w \) to \( T \).
- Update PQ by considering all edges \( e = v \to x \) incident to \( v \)
  - ignore if \( x \) is already in \( T \)
  - add \( x \) to PQ if not already on it
  - decrease priority of \( x \) if \( v \to x \) becomes shortest edge connecting \( x \) to \( T \)

**Prim’s algorithm - Eager implementation**

- 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: eager implementation**

- 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.

<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0-7</td>
<td>0.16</td>
</tr>
<tr>
<td>2</td>
<td>0-2</td>
<td>0.26</td>
</tr>
<tr>
<td>3</td>
<td>1-3</td>
<td>0.29</td>
</tr>
<tr>
<td>4</td>
<td>0-0</td>
<td>0.38</td>
</tr>
<tr>
<td>5</td>
<td>1-5</td>
<td>0.32</td>
</tr>
<tr>
<td>6</td>
<td>6-0</td>
<td>0.58</td>
</tr>
<tr>
<td>7</td>
<td>0-7</td>
<td>0.16</td>
</tr>
</tbody>
</table>

**Prim’s algorithm - Eager implementation**

- 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 - Eager implementation**

- 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.

<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0-7</td>
<td>0.16</td>
</tr>
<tr>
<td>2</td>
<td>0-2</td>
<td>0.26</td>
</tr>
<tr>
<td>3</td>
<td>1-5</td>
<td>0.32</td>
</tr>
<tr>
<td>4</td>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>5</td>
<td>5-7</td>
<td>0.28</td>
</tr>
<tr>
<td>6</td>
<td>6-0</td>
<td>0.58</td>
</tr>
<tr>
<td>7</td>
<td>0-7</td>
<td>0.16</td>
</tr>
</tbody>
</table>

vertices on PQ
(sorted by weight)

add vertices 7, 2, 4, and 6 to PQ
**Prim's algorithm - Eager implementation**

- 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.

<table>
<thead>
<tr>
<th>$v$</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>4</td>
<td>0–4</td>
<td>0.38</td>
</tr>
<tr>
<td>6</td>
<td>6–0</td>
<td>0.58</td>
</tr>
</tbody>
</table>

vertices on PQ (sorted by weight)

**MST edges**

0–7

---

**Prim's algorithm - Eager implementation**

- 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.

<table>
<thead>
<tr>
<th>$v$</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>5</td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>0–4</td>
<td>0.38</td>
</tr>
<tr>
<td>6</td>
<td>6–0</td>
<td>0.58</td>
</tr>
</tbody>
</table>

vertices on PQ (sorted by weight)

**MST edges**

0–7 1–7

---

**Prim's algorithm - Eager implementation**

- 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.

<table>
<thead>
<tr>
<th>$v$</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>1</td>
<td>1–7</td>
<td>0.19</td>
</tr>
<tr>
<td>5</td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>0–4</td>
<td>0.38</td>
</tr>
<tr>
<td>6</td>
<td>6–0</td>
<td>0.58</td>
</tr>
</tbody>
</table>

vertices on PQ (sorted by weight)

**MST edges**

0–7

---

**Prim's algorithm - Eager implementation**

- 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.

<table>
<thead>
<tr>
<th>$v$</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>1</td>
<td>1–7</td>
<td>0.19</td>
</tr>
<tr>
<td>5</td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>0–4</td>
<td>0.38</td>
</tr>
<tr>
<td>6</td>
<td>6–0</td>
<td>0.58</td>
</tr>
</tbody>
</table>

vertices on PQ (sorted by weight)

**MST edges**

0–7 1–7

---
Prim’s algorithm - Eager implementation

• 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.

MST edges

<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1–7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>5</td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>0–4</td>
<td>0.38</td>
</tr>
<tr>
<td>6</td>
<td>6–0</td>
<td>0.58</td>
</tr>
</tbody>
</table>

vertices on PQ (sorted by weight)

add vertex 3 to PQ
already a better connection to 5 and 7 (discard)

MST edges

<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0–7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1–7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0–2</td>
<td>0.26</td>
</tr>
<tr>
<td>5</td>
<td>5–7</td>
<td>0.28</td>
</tr>
<tr>
<td>3</td>
<td>1–3</td>
<td>0.29</td>
</tr>
<tr>
<td>4</td>
<td>0–4</td>
<td>0.38</td>
</tr>
<tr>
<td>6</td>
<td>6–0</td>
<td>0.58</td>
</tr>
</tbody>
</table>

0-7  1-7  0-2
Prim's algorithm - Eager implementation

- 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.

<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0-7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1-7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0-2</td>
<td>0.26</td>
</tr>
<tr>
<td>3</td>
<td>2-3</td>
<td>0.17</td>
</tr>
<tr>
<td>5</td>
<td>5-7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>6</td>
<td>6-2</td>
<td>0.40</td>
</tr>
</tbody>
</table>

MST edges
0-7  1-7  0-2  2-3

- Decrease key of vertex 3 from 0.29 to 0.17
- Decrease key of vertex 6 from 0.58 to 0.40

Prim's algorithm - Eager implementation

- 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.

<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0-7</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>1-7</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0-2</td>
<td>0.26</td>
</tr>
<tr>
<td>3</td>
<td>2-3</td>
<td>0.17</td>
</tr>
<tr>
<td>5</td>
<td>5-7</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>6</td>
<td>6-2</td>
<td>0.40</td>
</tr>
</tbody>
</table>

MST edges
0-7  1-7  0-2  2-3

Already a better connection to 6 (discard)
**Prim's algorithm - Eager implementation**

- 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.

\[
\begin{array}{c|c|c}
\text{v} & \text{edgeTo[]} & \text{distTo[]} \\
\hline
7 & 0-7 & 0.16 \\
1 & 1-7 & 0.19 \\
2 & 0-2 & 0.26 \\
3 & 2-3 & 0.17 \\
5 & 5-7 & 0.28 \\
4 & 0-4 & 0.38 \\
6 & 6-2 & 0.40 \\
\end{array}
\]

MST edges
0-7 1-7 0-2 2-3 5-7

**Prim's algorithm - Eager implementation**

- 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.

\[
\begin{array}{c|c|c}
\text{v} & \text{edgeTo[]} & \text{distTo[]} \\
\hline
7 & 0-7 & 0.16 \\
1 & 1-7 & 0.19 \\
2 & 0-2 & 0.26 \\
3 & 2-3 & 0.17 \\
5 & 5-7 & 0.28 \\
4 & 0-4 & 0.38 \\
6 & 6-2 & 0.40 \\
\end{array}
\]

MST edges
0-7 1-7 0-2 2-3 5-7

**Prim's algorithm - Eager implementation**

- 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.

\[
\begin{array}{c|c|c}
\text{v} & \text{edgeTo[]} & \text{distTo[]} \\
\hline
7 & 0-7 & 0.16 \\
1 & 1-7 & 0.19 \\
2 & 0-2 & 0.26 \\
3 & 2-3 & 0.17 \\
5 & 5-7 & 0.28 \\
4 & 4-5 & 0.35 \\
6 & 6-2 & 0.40 \\
\end{array}
\]

MST edges
0-7 1-7 0-2 2-3 5-7

**Prim's algorithm - Eager implementation**

- 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.

\[
\begin{array}{c|c|c}
\text{v} & \text{edgeTo[]} & \text{distTo[]} \\
\hline
7 & 0-7 & 0.16 \\
1 & 1-7 & 0.19 \\
2 & 0-2 & 0.26 \\
3 & 2-3 & 0.17 \\
5 & 5-7 & 0.28 \\
4 & 4-5 & 0.35 \\
6 & 6-2 & 0.40 \\
\end{array}
\]

MST edges
0-7 1-7 0-2 2-3 5-7
Prim's algorithm - Eager implementation

- 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.

\[\begin{array}{c|c|c}
  v & \text{edgeTo[]} & \text{distTo[]} \\
  \hline
  0 & - & - \\
  7 & 0-7 & 0.16 \\
  1 & 1-7 & 0.19 \\
  2 & 0-2 & 0.26 \\
  3 & 2-3 & 0.17 \\
  5 & 5-7 & 0.28 \\
  4 & 4-5 & 0.35 \\
  6 & 6-2 & 0.40 \\
\end{array}\]

MST edges

0-7 1-7 0-2 2-3 5-7 4-5

Prim's algorithm - Eager implementation

- 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.

\[\begin{array}{c|c|c}
  v & \text{edgeTo[]} & \text{distTo[]} \\
  \hline
  0 & - & - \\
  7 & 0-7 & 0.16 \\
  1 & 1-7 & 0.19 \\
  2 & 0-2 & 0.26 \\
  3 & 2-3 & 0.17 \\
  5 & 5-7 & 0.28 \\
  4 & 4-5 & 0.35 \\
  6 & 6-2 & 0.40 \\
\end{array}\]

MST edges

0-7 1-7 0-2 2-3 5-7 4-5

already a better connection to 6 (discard)

Prim's algorithm - Eager implementation

- 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.

\[\begin{array}{c|c|c}
  v & \text{edgeTo[]} & \text{distTo[]} \\
  \hline
  0 & - & - \\
  7 & 0-7 & 0.16 \\
  1 & 1-7 & 0.19 \\
  2 & 0-2 & 0.26 \\
  3 & 2-3 & 0.17 \\
  5 & 5-7 & 0.28 \\
  4 & 4-5 & 0.35 \\
  6 & 6-2 & 0.40 \\
\end{array}\]

MST edges

0-7 1-7 0-2 2-3 5-7 4-5
**Prim’s algorithm - Eager implementation**

- 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.

**Index 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.

```
public class IndexMinPQ<Key extends Comparable<Key>> {

    // Create indexed priority queue
    public IndexMinPQ(int N) {
        // Code...
    }

    // Insert a key with index k
    public void insert(int k, Key key) {
        // Code...
    }

    // Decrease the key associated with index k
    public void decreaseKey(int k, Key key) {
        // Code...
    }

    // Return the index of the minimal key
    public int delMin() {
        // Code...
    }

    // Return true if priority queue is empty
    public boolean isEmpty() {
        // Code...
    }

    // Return number of entries in priority queue
    public int size() {
        // Code...
    }
}
```

**Index 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)`.

**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>1</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>$d \log_d V$</td>
<td>$d \log_d V$</td>
<td>$d \log_d V$</td>
<td>$E \log_d V$</td>
</tr>
<tr>
<td>Fibonacci heap (Fredman-Tarjan 1984)</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.
**Minimum Spanning Trees**

- Greedy algorithm
- Edge-weighted graph API
- Kruskal’s algorithm
- Prim’s algorithm
- Context

---

**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 \( \sim cN \log N \).

---

**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 \( 10^9 \) 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 of cancers in human

Tumors in similar tissues cluster together.

Reference: Botstein & Brown group

Gene 1
Gene n
gene expressed
gene not expressed

Graphs

› Minimum Spanning Trees
› Shortest Paths
› Edge-weighted digraph API
› Shortest-paths properties
› Dijkstra’s algorithm
› Edge-weighted DAGs
› Negative weights

Shortest Paths

› Edge-weighted digraph API
› Shortest-paths properties
› Dijkstra’s algorithm
› Edge-weighted DAGs
› Negative weights
Shortest paths in a weighted digraph

Given an edge-weighted digraph, find the shortest (directed) path from \( s \) to \( t \).

**edge-weighted digraph**

- 4->5: 0.35
- 5->4: 0.35
- 4->7: 0.37
- 5->7: 0.28
- 7->5: 0.28
- 5->1: 0.32
- 0->4: 0.38
- 0->2: 0.26
- 7->3: 0.39
- 1->3: 0.29
- 2->7: 0.34
- 6->2: 0.40
- 3->6: 0.52
- 6->0: 0.58
- 6->4: 0.93

**shortest path from 0 to 6**

- 0->2: 0.26
- 2->7: 0.34
- 7->3: 0.39
- 3->6: 0.52

Google maps

Car navigation

Shortest path applications

- PERT/CPM.
- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

Shortest path variants

Which vertices?
• Source-sink: from one vertex to another.
• Single source: from one vertex to every other.
• All pairs: between all pairs of vertices.

Restrictions on edge weights?
• Nonnegative weights.
• Arbitrary weights.
• Euclidean weights.

Cycles?
• No directed cycles.
• No "negative cycles."

Simplifying assumption. Shortest paths from $s$ to each vertex $v$ exist.

Weighted directed edge API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DirectedEdge(int v, int w, double weight)</td>
<td>weighted edge $v \rightarrow w$</td>
</tr>
<tr>
<td>int from()</td>
<td>vertex $v$</td>
</tr>
<tr>
<td>int to()</td>
<td>vertex $w$</td>
</tr>
<tr>
<td>double weight()</td>
<td>weight of this edge</td>
</tr>
<tr>
<td>String toString()</td>
<td>string representation</td>
</tr>
</tbody>
</table>

Idiom for processing an edge $e$: int $v = e$.from(), $w = e$.to();

Shortest Paths

 Française: 

Edge-weighted digraph API
• Shortest-paths properties
• Dijkstra’s algorithm
• Edge-weighted DAGs
• Negative weights

Weighted directed edge: implementation in Java

Similar to Edge for undirected graphs, but a bit simpler.

```java
public class DirectedEdge {
    private final int v, w;
    private final double weight;
    public DirectedEdge(int v, int w, double weight) {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }
    public int from() {
        return v;
    }
    public int to() {
        return w;
    }
    public int weight() {
        return weight;
    }
}
```
Edge-weighted digraph API

```java
public class EdgeWeightedDigraph
{
    private final int V;
    private final Bag<Edge>[] adj;

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

    public void addEdge(DirectedEdge e)
    {
        int v = e.from();
        adj[v].add(e);
    }

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

    public String toString()
    {  return "toString()";  }
}
```

Conventions. Allow self-loops and parallel edges.

Edge-weighted digraph: adjacency-lists implementation in Java

Same as EdgeWeightedGraph except replace Graph with Digraph.

```java
public class EdgeWeightedDigraph
{
    private final int V;
    private final Bag<Edge>[] adj;

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

    public void addEdge(DirectedEdge e)
    {
        int v = e.from();
        adj[v].add(e);
    }

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

    public String toString()
    {  return "toString()";  }
}
```

Edge-weighted digraph: adjacency-lists representation

Single-source shortest paths API

Goal. Find the shortest path from $s$ to every other vertex.

```java
public class SP
{
    SP(EdgeWeightedDigraph G, int s)
    { shortest paths from $s$ in graph $G$
        double distTo(int v)
        { length of shortest path from $s$ to $v$
            Iterable<DirectedEdge> pathTo(int v)
            { shortest path from $s$ to $v$
                boolean hasPathTo(int v)
                { is there a path from $s$ to $v$?

        SP sp = new SP(G, s);
        for (int v = 0; v < G.V(); v++)
        {  StdOut.printf("%d to %d (%.2f): ", s, v, sp.distTo(v));
            for (DirectedEdge e : sp.pathTo(v))
                StdOut.print(e + "  ");
            StdOut.println();
        }
    }
```
**Single-source shortest paths API**

**Goal.** Find the shortest path from \( s \) to every other vertex.

```java
public class SP

SP(EdgeWeightedDigraph G, int s)  // shortest paths from \( s \) in graph \( G \)
  double distTo(int v)  // length of shortest path from \( s \) to \( v \)
  Iterable<DirectedEdge> pathTo(int v)  // shortest path from \( s \) to \( v \)
  boolean hasPathTo(int v)  // is there a path from \( s \) to \( v \)?
```

% java SP tinyEWD.txt 0
0 to 0 (0.00): 0 -> 0 0.00
0 to 1 (1.05): 0 -> 4 0.38 4 -> 5 0.35 5 -> 1 0.32
0 to 2 (0.26): 0 -> 2 0.26
0 to 3 (0.99): 0 -> 2 0.26 2 -> 7 0.34 7 -> 3 0.39
0 to 4 (0.38): 0 -> 4 0.38
0 to 5 (0.73): 0 -> 4 0.38 4 -> 5 0.35
0 to 6 (1.51): 0 -> 2 0.26 2 -> 7 0.34 7 -> 3 0.39 3 -> 6 0.52
0 to 7 (0.60): 0 -> 2 0.26 2 -> 7 0.34
```

**Data structures for single-source shortest paths**

**Goal.** Find the shortest path from \( s \) to every other vertex.

**Observation.** A *shortest-paths tree* (SPT) solution exists. Why?

**Consequence.** Can represent the SPT with two vertex-indexed arrays:
- \( \text{distTo}[v] \) is length of shortest path from \( s \) to \( v \).
- \( \text{edgeTo}[v] \) is last edge on shortest path from \( s \) to \( v \).

```
shortest-paths tree from 0
```

```
public double distTo(int v)  {  return distTo[v];  }
public Iterable<DirectedEdge> pathTo(int v)  {
  Stack<DirectedEdge> path = new Stack<DirectedEdge>();
  for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
    path.push(e);
  return path;
}
```
**Shortest-paths optimality conditions**

**Proposition.** Let $G$ be an edge-weighted digraph. Then $\text{distTo}[]$ are the shortest path distances from $s$ iff:

- For each vertex $v$, $\text{distTo}[v]$ is the length of some path from $s$ to $v$.
- For each edge $e = v \rightarrow w$, $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.

**Pf.** \(\Rightarrow [\text{ sufficient }]\)

- Suppose that $\text{distTo}[w] > \text{distTo}[v] + e.\text{weight}()$ for some edge $e = v \rightarrow w$.
- Then, $e$ gives a path from $s$ to $w$ (through $v$) of length less than $\text{distTo}[w]$.

- Add inequalities; simplify; and substitute $\text{distTo}[v_i] = 0$:
  \[
  \text{distTo}[w] = \text{distTo}[v_i] + e_i.\text{weight}() + \ldots + e_s.\text{weight}() \\
  \text{weight of shortest path from } s \text{ to } w \\
  \text{weight of some path from } s \text{ to } w \\
  \]
- Thus, $\text{distTo}[w]$ is the weight of shortest path to $w$. $\blacksquare$

**Shortest-paths optimality conditions**

**Proposition.** Let $G$ be an edge-weighted digraph. Then $\text{distTo}[]$ are the shortest path distances from $s$ iff:

- For each vertex $v$, $\text{distTo}[v]$ is the length of some path from $s$ to $v$.
- For each edge $e = v \rightarrow w$, $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.

**Pf.** \(\Leftarrow [\text{ necessary }]\)

- Suppose that $\text{distTo}[w] > \text{distTo}[v] + e.\text{weight}()$ for some edge $e = v \rightarrow w$.
- Then, $e$ gives a path from $s$ to $w$ (through $v$) of length less than $\text{distTo}[w]$.

- Add inequalities; simplify; and substitute $\text{distTo}[v_i] = 0$:
  \[
  \text{distTo}[w] = \text{distTo}[v_i] + e_i.\text{weight}() + \ldots + e_s.\text{weight}() \\
  \text{weight of shortest path from } s \text{ to } w \\
  \text{weight of some path from } s \text{ to } w \\
  \]
**Proposition.** Generic algorithm computes SPT (if it exists) from \( s \).

**Pf sketch.**

- Throughout algorithm, \( \text{distTo}[v] \) is the length of a simple path from \( s \) to \( v \) (and \( \text{edgeTo}[v] \) is last edge on path).
- Each successful relaxation decreases \( \text{distTo}[v] \) for some \( v \).
- The entry \( \text{distTo}[v] \) can decrease at most a finite number of times. 

**Generic algorithm (to compute SPT from \( s \))**

- Initialize \( \text{distTo}[s] = 0 \) and \( \text{distTo}[v] = \infty \) for all other vertices.
- Repeat until optimality conditions are satisfied:
  - Relax any edge.

**Efficient implementations.** How to choose which edge to relax?

- **Ex 1.** Dijkstra’s algorithm (nonnegative weights).
- **Ex 2.** Topological sort algorithm (no directed cycles).
- **Ex 3.** Bellman-Ford algorithm (no negative cycles).

---

**Edsger W. Dijkstra: select quotes**

- “Do only what only you can do.”
- “In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.”
- “The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.”
- “It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.”
- “APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.”
Edsger W. Dijkstra: select quotes

"Object-oriented programming is an exceptionally bad idea which could only have originated in California." — Edsger Dijkstra

Dijkstra's algorithm

- Consider vertices in increasing order of distance from $s$ (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges incident from that vertex.

Dijkstra's algorithm

- Consider vertices in increasing order of distance from $s$ (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges incident from that vertex.
Dijkstra's algorithm

• Consider vertices in increasing order of distance from $s$
  (non-tree vertex with the lowest $\text{distTo}[]$ value).
• Add vertex to tree and relax all edges incident from that vertex.

relax all edges incident from 0

choose vertex 1

relax all edges incident from 1
Dijkstra's algorithm

- Consider vertices in increasing order of distance from \( s \) (non-tree vertex with the lowest \( \text{distTo}[] \) value).
- Add vertex to tree and relax all edges incident from that vertex.

Choose vertex 7

Choose vertex 7

Choose vertex 7

Choose vertex 7

Relax all edges incident from 7

Relax all edges incident from 7

Relax all edges incident from 1

Relax all edges incident from 1
Dijkstra's algorithm

- Consider vertices in increasing order of distance from \( s \) (non-tree vertex with the lowest \( \text{distTo}[\cdot] \) value).
- Add vertex to tree and relax all edges incident from that vertex.

\[
\begin{array}{c|c|c}
\text{v} & \text{distTo[\cdot]} & \text{edgeTo[\cdot]} \\
\hline
0 & 0.0 & - \\
1 & 5.0 & 0\rightarrow1 \\
2 & 15.0 & 7\rightarrow2 \\
3 & 20.0 & 1\rightarrow3 \\
4 & 9.0 & 0\rightarrow4 \\
5 & 14.0 & 7\rightarrow5 \\
6 & & \\
7 & 8.0 & 0\rightarrow7 \\
\end{array}
\]

relax all edges incident from 7

select vertex 4

\[
\begin{array}{c|c|c}
\text{v} & \text{distTo[\cdot]} & \text{edgeTo[\cdot]} \\
\hline
0 & 0.0 & - \\
1 & 5.0 & 0\rightarrow1 \\
2 & 15.0 & 7\rightarrow2 \\
3 & 20.0 & 1\rightarrow3 \\
4 & 9.0 & 0\rightarrow4 \\
5 & 14.0 & 7\rightarrow5 \\
6 & & \\
7 & 8.0 & 0\rightarrow7 \\
\end{array}
\]

relax all edges incident from 4
Dijkstra's algorithm

- Consider vertices in increasing order of distance from $s$ (non-tree vertex with the lowest $distTo[]$ value).
- Add vertex to tree and relax all edges incident from that vertex.

```
<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>15.0</td>
<td>7→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>29.0</td>
<td>4→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>
```

1. Consider vertices in increasing order of distance from $s$ (non-tree vertex with the lowest $distTo[]$ value).
2. Add vertex to tree and relax all edges incident from that vertex.

relax all edges incident from 4

select vertex 5

relax all edges incident from 5
Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges incident from that vertex.

Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges incident from that vertex.

Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges incident from that vertex.

Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges incident from that vertex.
Dijkstra’s algorithm

- Consider vertices in increasing order of distance from $s$
  (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges incident from that vertex.

\[
\begin{array}{c|c|c}
\text{v} & \text{distTo}[] & \text{edgeTo}[] \\
0 & 0.0 & - \\
1 & 5.0 & 0\rightarrow1 \\
2 & 14.0 & 5\rightarrow2 \\
3 & 17.0 & 2\rightarrow3 \\
4 & 9.0 & 0\rightarrow4 \\
5 & 13.0 & 4\rightarrow5 \\
6 & 25.0 & 2\rightarrow6 \\
7 & 8.0 & 0\rightarrow7 \\
\end{array}
\]

relax all edges incident from 2

select vertex 3

\[
\begin{array}{c|c|c}
\text{v} & \text{distTo}[] & \text{edgeTo}[] \\
0 & 0.0 & - \\
1 & 5.0 & 0\rightarrow1 \\
2 & 14.0 & 5\rightarrow2 \\
3 & 17.0 & 2\rightarrow3 \\
4 & 9.0 & 0\rightarrow4 \\
5 & 13.0 & 4\rightarrow5 \\
6 & 25.0 & 2\rightarrow6 \\
7 & 8.0 & 0\rightarrow7 \\
\end{array}
\]

relax all edges incident from 3

Dijkstra’s algorithm

- Consider vertices in increasing order of distance from $s$
  (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges incident from that vertex.
Consider vertices in increasing order of distance from $s$ (non-tree vertex with the lowest $distTo[]$ value).

Add vertex to tree and relax all edges incident from that vertex.

Dijkstra's algorithm

- Consider vertices in increasing order of distance from $s$ (non-tree vertex with the lowest $distTo[]$ value).
- Add vertex to tree and relax all edges incident from that vertex.

Dijkstra's algorithm

- Consider vertices in increasing order of distance from $s$ (non-tree vertex with the lowest $distTo[]$ value).
- Add vertex to tree and relax all edges incident from that vertex.

Dijkstra's algorithm
Dijkstra's algorithm

- Consider vertices in increasing order of distance from $s$
  (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges incident from that vertex.

<table>
<thead>
<tr>
<th>$v$</th>
<th>$\text{distTo}[]$</th>
<th>$\text{edgeTo}[]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>17.0</td>
<td>2→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>25.0</td>
<td>2→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

Dijkstra's algorithm visualization

shortest-paths tree from vertex $s$
**Dijkstra’s algorithm: correctness proof**

**Proposition.** Dijkstra’s algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

**Pf.**

- Each edge \( e = \overrightarrow{v \rightarrow w} \) is relaxed exactly once (when \( v \) is relaxed), leaving \( \text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight()} \).
- Inequality holds until algorithm terminates because:
  - \( \text{distTo}[w] \) cannot increase
  - \( \text{distTo}[v] \) will not change
- Thus, upon termination, shortest-paths optimality conditions hold.

**Dijkstra’s algorithm: Java implementation**

```java
public class DijkstraSP {
    private DirectedEdge[] edgeTo;
    private double[] distTo;
    private IndexMinPQ<Double> pq;
    public DijkstraSP(EdgeWeightedDigraph G, int s) {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        pq = new IndexMinPQ<Double>(G.V());
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;
        pq.insert(s, 0.0);
        while (!pq.isEmpty()) {
            int v = pq.delMin();
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }
}
```

**Dijkstra’s algorithm: which priority queue?**

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_d V )</td>
<td>( d \log_d V )</td>
<td>( d \log_d V )</td>
<td>( E \log_{d/V} V )</td>
</tr>
<tr>
<td>Fibonacci heap (Fredman-Tarjan 1984)</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.
- d-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.
**Priority-first search**

**Insight.** Four of our graph-search methods are the same algorithm!
- Maintain a set of explored vertices \( S \).
- Grow \( S \) by exploring edges with exactly one endpoint leaving \( S \).

**DFS.** Take edge from vertex which was discovered most recently.

**BFS.** Take edge from vertex which was discovered least recently.

**Prim.** Take edge of minimum weight.

**Dijkstra.** Take edge to vertex that is closest to \( S \).

**Challenge.** Express this insight in reusable Java code.

---

**Acyclic edge-weighted digraphs**

**Q.** Suppose that an edge-weighted digraph has no directed cycles. Is it easier to find shortest paths than in a general digraph?

**A.** Yes!

---

**Shortest Paths**

- Edge-weighted digraph API
- Shortest-paths properties
- Dijkstra’s algorithm
- Edge-weighted DAGs
- Negative weights

---

**Topological sort algorithm**

- Consider vertices in topological order.
- Relax all edges incident from that vertex.
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

Topological order: 0 1 4 7 5 2 3 6

choose vertex 0

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

relax all edges incident from 0

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

relax all edges incident from 0
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

choose vertex 1

relax all edges incident from 1
Consider vertices in topological order.
Relax all edges incident from that vertex.

Topological sort algorithm

<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>17.0</td>
<td>1→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

select vertex 4
(Dijkstra would have selected vertex 7)

Topological sort algorithm

<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>17.0</td>
<td>1→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

relax all edges incident from 4

Topological sort algorithm

<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>17.0</td>
<td>1→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

relax all edges incident from 4
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

0 1 4 7 5 2 3 6

v   distTo[]  edgeTo[]
0     0.0        -
1     5.0       0→1
2     17.0      1→2
3     20.0      1→3
4     9.0       0→4
5     13.0      4→5
6     29.0      4→6
7     8.0       0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

choose vertex 7

0 1 4 7 5 2 3 6

v   distTo[]  edgeTo[]
0     0.0       -
1     5.0       0→1
2     17.0      1→2
3     20.0      1→3
4     9.0       0→4
5     13.0      4→5
6     29.0      4→6
7     8.0       0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

relax all edges incident from 7

0 1 4 7 5 2 3 6

v   distTo[]  edgeTo[]
0     0.0       -
1     5.0       0→1
2     17.0      1→2
3     20.0      1→3
4     9.0       0→4
5     13.0      4→5
6     29.0      4→6
7     8.0       0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

relax all edges incident from 7

0 1 4 7 5 2 3 6

v   distTo[]  edgeTo[]
0     0.0       -
1     5.0       0→1
2     15.0      7→2
3     20.0      1→3
4     9.0       0→4
5     13.0      4→5
6     29.0      4→6
7     8.0       0→7
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

Consider vertices in topological order.
Relax all edges incident from that vertex.

Topological sort algorithm

select vertex 5

relax all edges incident from 5
Consider vertices in topological order.
Relax all edges incident from that vertex.

Topological sort algorithm

Topological sort algorithm

Topological sort algorithm

Topological sort algorithm
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

Relax all edges incident from 3

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

Relax all edges incident from 3

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

Relax all edges incident from 3
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

\[ v \text{ distTo[]} \text{ edgeTo[]} \]
\[
\begin{array}{c|c|c}
0 & 0.0 & - \\
1 & 5.0 & 0\rightarrow\text{1} \\
2 & 14.0 & 5\rightarrow\text{2} \\
3 & 17.0 & 2\rightarrow\text{3} \\
4 & 9.0 & 0\rightarrow\text{4} \\
5 & 13.0 & 4\rightarrow\text{5} \\
6 & 25.0 & 2\rightarrow\text{6} \\
7 & 8.0 & 0\rightarrow\text{7} \\
\end{array}
\]

select vertex 6

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

\[ v \text{ distTo[]} \text{ edgeTo[]} \]
\[
\begin{array}{c|c|c}
0 & 0.0 & - \\
1 & 5.0 & 0\rightarrow\text{1} \\
2 & 14.0 & 5\rightarrow\text{2} \\
3 & 17.0 & 2\rightarrow\text{3} \\
4 & 9.0 & 0\rightarrow\text{4} \\
5 & 13.0 & 4\rightarrow\text{5} \\
6 & 25.0 & 2\rightarrow\text{6} \\
7 & 8.0 & 0\rightarrow\text{7} \\
\end{array}
\]

relax all edges incident from 6
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

```
 0 1 4 7 5 2 3 6

  v  distTo[]  edgeTo[]
  0    0.0       -
  1    5.0       0→1
  2    14.0       5→2
  3    17.0       2→3
  4    9.0        0→4
  5    13.0       4→5
  6    25.0       2→6
  7    8.0        0→7
```

shortest-paths tree from vertex s

**Proposition.** Topological sort algorithm computes SPT in any edge-weighted DAG in time proportional to $E + V$.

**Pf.**
- Each edge $e = v \rightarrow w$ is relaxed exactly once (when $v$ is relaxed), leaving $\text{distTo}[w] \leq \text{distTo}[v] + e\text{.weight()}$.  
- Inequality holds until algorithm terminates because:
  - $\text{distTo}[w]$ cannot increase  
  - $\text{distTo}[v]$ will not change  
- Thus, upon termination, shortest-paths optimality conditions hold.

Shortest paths in edge-weighted DAGs

```java
public class AcyclicSP {
    private DirectedEdge[] edgeTo;
    private double[] distTo;
    public AcyclicSP(EdgeWeightedDigraph G, int s) {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;
        Topological topological = new Topological(G);
        for (int v : topological.order())
            for (DirectedEdge e : G.adj(v))
                relax(e);
    }
}
```

Content-aware resizing

**Seam carving.** [Avidan and Shamir] Resize an image without distortion for display on cell phones and web browsers.
**Content-aware resizing**

*Seam carving.* [Avidan and Shamir] Resize an image without distortion for display on cell phones and web browsers.

*In the wild.* Photoshop CS 5, Imagemagick, GIMP, ...

---

**Content-aware resizing**

To find vertical seam:
- Grid DAG: vertex = pixel; edge = from pixel to 3 downward neighbors.
- Weight of pixel = energy function of 8 neighboring pixels.
- Seam = shortest path from top to bottom.

---

**Content-aware resizing**

To remove vertical seam:
- Delete pixels on seam (one in each row).
Content-aware resizing

To remove vertical seam:
- Delete pixels on seam (one in each row).

Longest paths in edge-weighted DAGs

Formulate as a shortest paths problem in edge-weighted DAGs.
- Negate all weights.
- Find shortest paths.
- Negate weights in result.

Key point: Topological sort algorithm works even with negative edge weights.

Longest paths in edge-weighted DAGs: application

Parallel job scheduling. Given a set of jobs with durations and precedence constraints, schedule the jobs (by finding a start time for each) so as to achieve the minimum completion time, while respecting the constraints.

Critical path method

CPM. To solve a parallel job-scheduling problem, create edge-weighted DAG:
- Source and sink vertices.
- Two vertices (begin and end) for each job.
- Three edges for each job.
  - begin to end (weighted by duration)
  - source to begin (0 weight)
  - end to sink (0 weight)
- One edge for each precedence constraint (0 weight).
Critical path method

**CPM.** Use longest path from the source to schedule each job.

**Parallel job scheduling solution**

**Shortest paths with negative weights: failed attempts**

**Dijkstra.** Doesn’t work with negative edge weights.

**Re-weighting.** Add a constant to every edge weight doesn’t work.

**Bad news.** Need a different algorithm.

Shortest Paths

- Edge-weighted digraph API
- Shortest-paths properties
- Dijkstra’s algorithm
- Edge-weighted DAGs
- Negative weights

Negative cycles

**Def.** A **negative cycle** is a directed cycle whose sum of edge weights is negative.

**Proposition.** A SPT exists iff no negative cycles.

assuming all vertices reachable from s
Bellman-Ford algorithm

Bellman–Ford algorithm

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.
Repeat V times:
  - Relax each edge.

for (int i = 0; i < G.V(); i++)
  for (int v = 0; v < G.V(); v++)
    for (DirectedEdge e : G.adj(v))
      relax(e);

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.
Repeat $V$ times: relax all $E$ edges.
Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.
Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

**Pass 1:**
- $v$  distTo[]  edgeTo[]
  - 0  0.0  -
  - 1  5.0  0→1
  - 2  17.0  1→2
  - 3  20.0  0→3
  - 4  9.0  0→4
  - 5
  - 6
  - 7  8.0  0→7

**Pass 2:**
- $v$  distTo[]  edgeTo[]
  - 0  0.0  -
  - 1  5.0  0→1
  - 2  17.0  1→2
  - 3  20.0  1→3
  - 4  9.0  0→4
  - 5
  - 6
  - 7  8.0  0→7
Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.
Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.
Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

<table>
<thead>
<tr>
<th>$v$</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>28.0</td>
<td>2→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

<table>
<thead>
<tr>
<th>$v$</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>26.0</td>
<td>5→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

<table>
<thead>
<tr>
<th>$v$</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>26.0</td>
<td>5→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

<table>
<thead>
<tr>
<th>$v$</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>26.0</td>
<td>5→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

\[ \begin{array}{lcl}
\text{pass 1} \\
0 \rightarrow 1 & 0 \rightarrow 4 & 0 \rightarrow 7 \\
1 \rightarrow 2 & 1 \rightarrow 3 & 1 \rightarrow 7 \\
2 \rightarrow 3 & 2 \rightarrow 6 & 3 \rightarrow 6 \\
3 \rightarrow 5 & 4 \rightarrow 6 & 4 \rightarrow 7 \\
5 \rightarrow 2 & 5 \rightarrow 5 & 6 \rightarrow 7 \\
6 \rightarrow 7 & 7 \rightarrow 2 \\
\end{array} \]

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

\[ \begin{array}{lcl}
\text{pass 1} \\
0 \rightarrow 1 & 0 \rightarrow 4 & 0 \rightarrow 7 \\
1 \rightarrow 2 & 1 \rightarrow 3 & 1 \rightarrow 7 \\
2 \rightarrow 3 & 2 \rightarrow 6 & 3 \rightarrow 6 \\
3 \rightarrow 5 & 4 \rightarrow 6 & 4 \rightarrow 7 \\
5 \rightarrow 2 & 5 \rightarrow 5 & 6 \rightarrow 7 \\
6 \rightarrow 7 & 7 \rightarrow 2 \\
\end{array} \]
Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.
Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>17.0</td>
<td>2→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>26.0</td>
<td>5→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>17.0</td>
<td>2→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>25.0</td>
<td>2→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 6→7 7→5 7→2

Bellman-Ford algorithm demo

<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>17.0</td>
<td>2→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>25.0</td>
<td>2→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 6→7 7→5 7→2

Bellman-Ford algorithm demo

<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>17.0</td>
<td>2→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>25.0</td>
<td>2→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 6→7 7→5 7→2

Bellman-Ford algorithm demo

<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>17.0</td>
<td>2→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>25.0</td>
<td>2→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 6→7 7→5 7→2

Bellman-Ford algorithm demo

<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>17.0</td>
<td>2→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>25.0</td>
<td>2→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 6→7 7→5 7→2

Bellman-Ford algorithm demo

<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>17.0</td>
<td>2→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>25.0</td>
<td>2→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 6→7 7→5 7→2

Bellman-Ford algorithm demo

<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>17.0</td>
<td>2→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>25.0</td>
<td>2→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 6→7 7→5 7→2
Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.
Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.
Bellman-Ford algorithm visualization

Bellman-Ford algorithm: analysis

**Proposition.** Dynamic programming algorithm computes SPT in any edge-weighted digraph with no negative cycles in time proportional to $E \times V$.

**Pf idea.** After pass $i$, found shortest path containing at most $i$ edges.

Bellman-Ford algorithm: practical improvement

**Observation.** If $\text{distTo}[v]$ does not change during pass $i$, no need to relax any edge pointing from $v$ in pass $i + 1$.

**FIFO implementation.** Maintain queue of vertices whose $\text{distTo}[\cdot]$ changed. be careful to keep at most one copy of each vertex on queue (why?)

**Overall effect.**
- The running time is still proportional to $E \times V$ in worst case.
- But much faster than that in practice.

Bellman-Ford algorithm: Java implementation

```java
public class BellmanFordSP
{
    private double[] distTo;
    private DirectedEdge[] edgeTo;
    private boolean[] onQ;
    private Queue<Integer> queue;

    public BellmanFordSP(EdgeWeightedDigraph G, int s)
    {
        distTo = new double[G.V()];
        edgeTo = new DirectedEdge[G.V()];
        onQ = new boolean[G.V()];
        queue = new Queue<Integer>();

        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;

        distTo[s] = 0.0;
        queue.enqueue(s);
        while (!queue.isEmpty())
        {
            int v = queue.dequeue();
            onQ[v] = false;
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }
}
```

private void relax(DirectedEdge e)
{
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
        if (!onQ[w])
        {
            queue.enqueue(w);
            onQ[w] = true;
        }
    }
}
Single source shortest-paths implementation: cost summary

<table>
<thead>
<tr>
<th>algorithm</th>
<th>restriction</th>
<th>typical case</th>
<th>worst case</th>
<th>extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>topological sort</td>
<td>no directed cycles</td>
<td>E + V</td>
<td>E + V</td>
<td>V</td>
</tr>
<tr>
<td>Dijkstra (binary heap)</td>
<td>no negative weights</td>
<td>E log V</td>
<td>E log V</td>
<td>V</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>no negative cycles</td>
<td>E</td>
<td>E</td>
<td>V</td>
</tr>
<tr>
<td>Bellman-Ford (queue-based)</td>
<td>no negative cycles</td>
<td>E + V</td>
<td>E</td>
<td>V</td>
</tr>
</tbody>
</table>

Remark 1. Directed cycles make the problem harder.
Remark 2. Negative weights make the problem harder.
Remark 3. Negative cycles makes the problem intractable.

Finding a negative cycle

Observation. If there is a negative cycle, Bellman-Ford gets stuck in loop, updating distTo[] and edgeTo[] entries of vertices in the cycle.

![Diagram](image)

Proposition. If any vertex v is updated in phase V, there exists a negative cycle (and can trace back edgeTo[v] entries to find it).

In practice. Check for negative cycles more frequently.

Finding a negative cycle

Negative cycle. Add two method to the API for sp.

```java
boolean hasNegativeCycle() is there a negative cycle?
Iterable<DirectedEdge> negativeCycle() negative cycle reachable from s
```

Problem. Given table of exchange rates, is there an arbitrage opportunity?

<table>
<thead>
<tr>
<th></th>
<th>USD</th>
<th>EUR</th>
<th>GBP</th>
<th>CHF</th>
<th>CAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>USD</td>
<td>1</td>
<td>0.741</td>
<td>0.657</td>
<td>1.061</td>
<td>1.011</td>
</tr>
<tr>
<td>EUR</td>
<td>1.350</td>
<td>1</td>
<td>0.888</td>
<td>1.433</td>
<td>1.366</td>
</tr>
<tr>
<td>GBP</td>
<td>1.521</td>
<td>1.126</td>
<td>1</td>
<td>1.614</td>
<td>1.538</td>
</tr>
<tr>
<td>CHF</td>
<td>0.943</td>
<td>0.698</td>
<td>0.620</td>
<td>1</td>
<td>0.953</td>
</tr>
<tr>
<td>CAD</td>
<td>0.995</td>
<td>0.732</td>
<td>0.650</td>
<td>1.049</td>
<td>1</td>
</tr>
</tbody>
</table>

Ex. $1,000 ⇒ 741 Euros ⇒ 1,012.206 Canadian dollars ⇒ $1,007.14497.

$$1000 \times 0.741 \times 1.366 \times 0.995 = 1007.14497$$
**Negative cycle application: arbitrage detection**

Currency exchange graph.
- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find a directed cycle whose product of edge weights is > 1.

**Challenge.** Express as a negative cycle detection problem.

**Negative cycle application: arbitrage detection**

Model as a negative cycle detection problem by taking logs.
- Let weight of edge \( v \rightarrow w \) be \( -\ln \) (exchange rate from currency \( v \) to \( w \)).
- Multiplication turns to addition; \( > 1 \) turns to \( < 0 \).
- Find a directed cycle whose sum of edge weights is \( < 0 \) (negative cycle).

**Remark.** Fastest algorithm is extraordinarily valuable!

**Shortest paths summary**

Dijkstra’s algorithm.
- Nearly linear-time when weights are nonnegative.
- Generalization encompasses DFS, BFS, and Prim.

Acyclic edge-weighted digraphs.
- Arise in applications.
- Faster than Dijkstra’s algorithm.
- Negative weights are no problem.

Negative weights and negative cycles.
- Arise in applications.
- If no negative cycles, can find shortest paths via Bellman-Ford.
- If negative cycles, can find one via Bellman-Ford.

Shortest-paths is a broadly useful problem-solving model.