Chapter 7: Minimum Spanning Trees

Algorithms

Spanning trees | Cut and cycle properties | Kruskal | Union-Find | Prim | Borůvka | Reverse-delete | Comparison

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/algorithms/slides/Chapter_07_Minimum_Spanning_Trees.html

Learning Outcomes

The concept

Define spanning trees and minimum spanning trees (MSTs); explain the V − 1 edges and the cut and cycle properties.

Kruskal and Union-Find

Sort the edges and keep those that close no cycle; check for cycles with disjoint sets (find / union).

Prim

Grow a tree from one vertex; trace the key and parent arrays with a priority queue step by step.

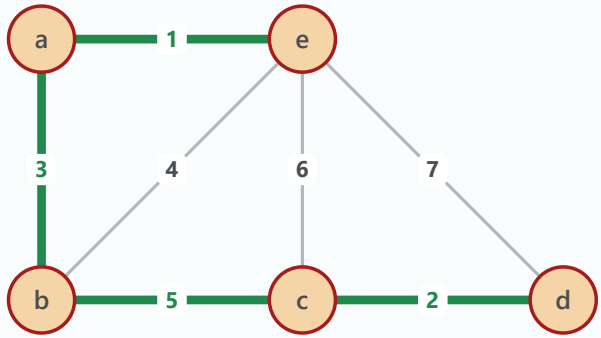
Borůvka, Reverse-Delete, Analysis

Apply the component-based and deletion-based approaches; compare the running times of all four algorithms and pick the right one.

THE MST CONCEPT

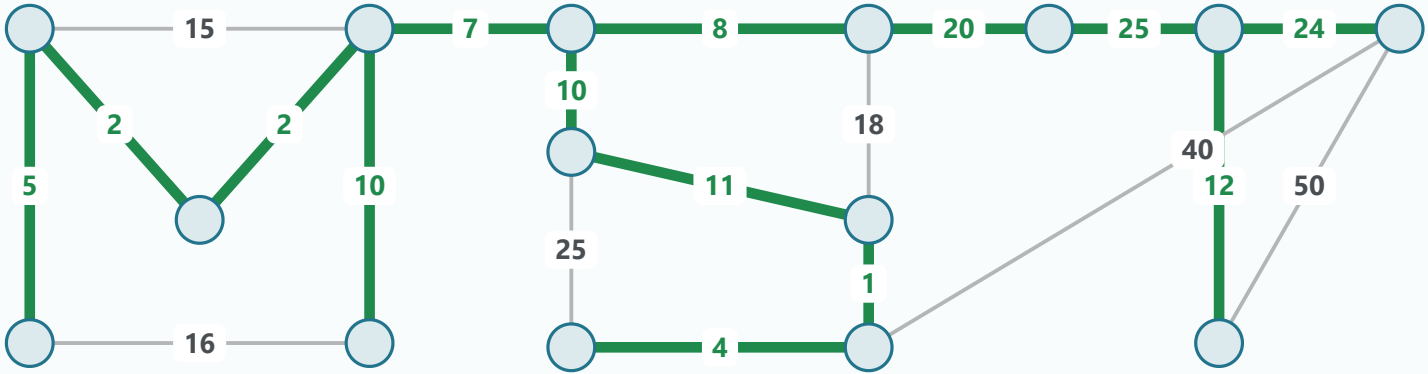
Spanning Trees and Minimum Spanning Trees

- G = (V, E): a **connected**, **undirected** graph with **weighted** edges.
- A **spanning tree** is a **connected**, **acyclic** subgraph that contains every vertex. It always has exactly **V − 1** edges.
- A **minimum spanning tree** (MST) is a spanning tree with the **smallest** total edge weight.
- Adding an edge to a tree creates a cycle; removing one splits the tree in two.



A graph can have very many spanning trees (n^{n-2} for a complete graph on n vertices, by Cayley's formula). Instead of trying them all, **greedy** algorithms find an MST directly. In the graph at right the MST weighs w(T) = 1 + 2 + 3 + 5 = **11**.

Example: The "MST" Graph

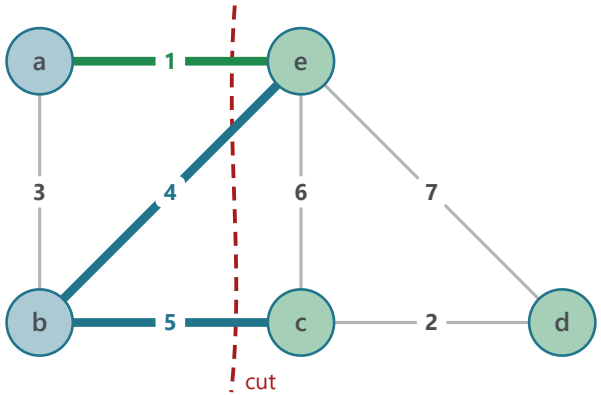


- 15 vertices, 20 edges. The 14 **bold** edges (= V − 1) form the minimum spanning tree: total weight **141**.
- The thin edges are left out: each closes a **cycle** with the bold edges and is the **heaviest** edge on that cycle (e.g. 15, 16, 18, 25, 40, 50).
- Heavy edges (20, 24, 25) can still be chosen: none of them is the **heaviest** on any cycle. For example, the cycle closed by 40 contains all three; the heaviest on that cycle is 40.

The Cut Property and the Cycle Property

Cut property

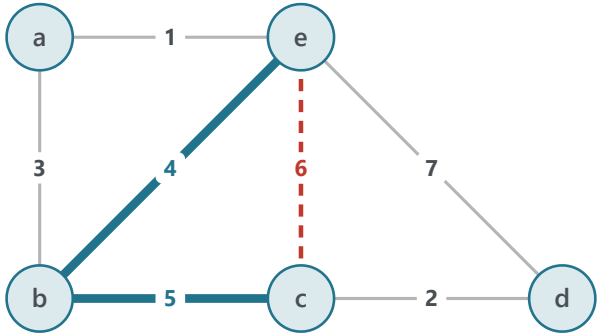
For any **cut** splitting the vertices in two, the **lightest** edge crossing the cut belongs to an MST.



{a, b} | {c, d, e}: crossing edges ae = 1, be = 4, bc = 5 → **ae** is safe.

Cycle property

The **heaviest edge on any cycle** (if unique) is in no MST.



cycle b–c–e: 4, 5, 6 → **ec = 6** is never chosen.

Kruskal and Prim rely on the cut property, reverse-delete on the cycle property. If all weights are distinct, the MST is **unique**.

Greedy MST Algorithms

All four algorithms are **greedy**: each step takes the locally best edge (or throws away the worst), and that choice is never undone.

Algorithm	Idea	Helper structure	Time
Kruskal (1956)	Sort edges from light to heavy, add those that close no cycle	union-find	O(E log E)
Prim (1957)	Start at one vertex and grow the tree by the cheapest adjacent edge	priority queue	O((V + E) log V)
Borůvka (1926)	Every component adds its own cheapest edge at the same time	union-find	O(E log V)
Reverse-delete	Try edges from heavy to light, delete those whose removal keeps the graph connected	BFS/DFS or union-find	O(E · (V + E))

All of them find an MST of the same weight; with distinct weights they find **the same tree**. They differ in the order they consider edges and in the data structures they need.

Where Are MSTs Used?

Network design

Connecting cities with the least cable: power, water, fiber and road networks. Borůvka's starting point was the power grid of Moravia.

Clustering

Deleting the $k - 1$ heaviest edges of the MST leaves k clusters (single-linkage clustering).

Approximation algorithms

For the metric traveling salesman problem (TSP), an MST-based tour is at most twice the optimum.

Image processing

Segmentation, with pixels as vertices and color differences as weights.

Circuit design

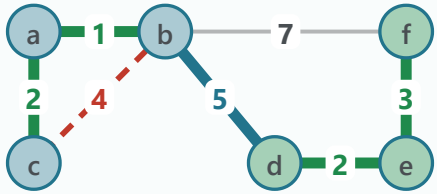
Connecting pins with the shortest total wire (a first step in VLSI routing).

Other graph problems

Every path in the MST is a bottleneck path (its largest edge is as small as possible); the basic approximation for Steiner trees.

Kruskal's Algorithm

- Builds a tree of minimum total weight that connects all vertices; a **greedy** approach.
- Developed by Joseph **Kruskal** (1956).
- At first every vertex is a **separate tree** (a forest); step by step these trees are **merged**.
- Edges are sorted **from lightest to heaviest**; an edge that closes no cycle is taken, one that closes a cycle is skipped.



green: chosen · red: cycle (skip)
blue: next, merges two trees

Strengths

+ Well suited to **sparse** graphs (E ≈ V).
+ Simple: an edge list plus union-find is enough.
+ On a disconnected graph it gives a minimum spanning **forest**.

Weaknesses

– The cost of **sorting edges**: O(E log E).
– On dense graphs (E ≈ V^2) Prim can be faster.
– Cycle checks need an extra structure (union-find).

Kruskal: Steps and Pseudocode

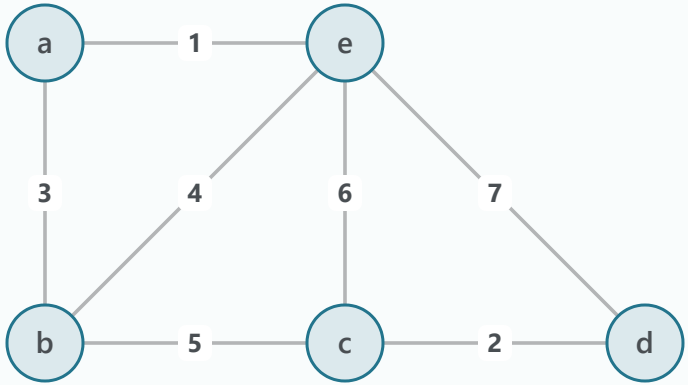
1. **Sort** all edges of the graph by weight.
2. Take the next edge with the **smallest** weight from the sorted list.
3. If it joins two vertices in **different trees** (closes no cycle), add it and merge the two trees.
4. If it joins two vertices of the same tree, it closes a cycle: **skip it**.
5. Repeat steps 2–4 until V − 1 edges are chosen (or the edges run out).

```
KRUSKAL(G):
  MST = [ ]
  dsu.makeSets(V)
  edges.sort(by increasing weight)
  for each (u, v, weight) in edges:
    if dsu.find(u) ≠ dsu.find(v):
      MST.add((u, v, weight))
      dsu.union(u, v)
  return MST
```

"Are they in the same tree?" is answered in nearly constant time by the **union-find** (disjoint-set) structure.

Kruskal Step by Step

ab	ae	bc	be	cd	ed	ec
3	1	5	4	2	7	6

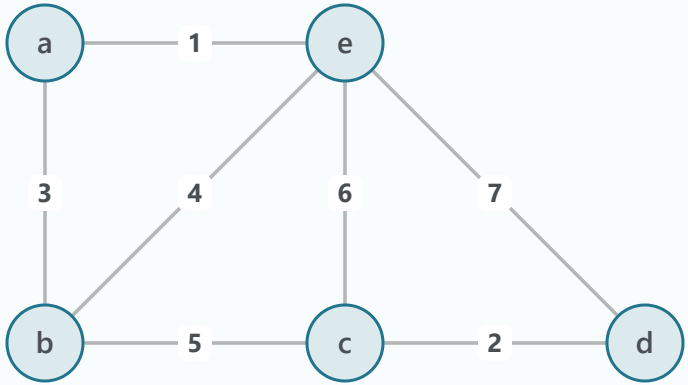


MST	—
w(T)	0
Trees	{a} {b} {c} {d} {e}

Start: each vertex is a **separate tree** (a forest of 5 trees). The edge table is still unsorted.

Kruskal Step by Step

ae	cd	ab	be	bc	ec	ed
1	2	3	4	5	6	7

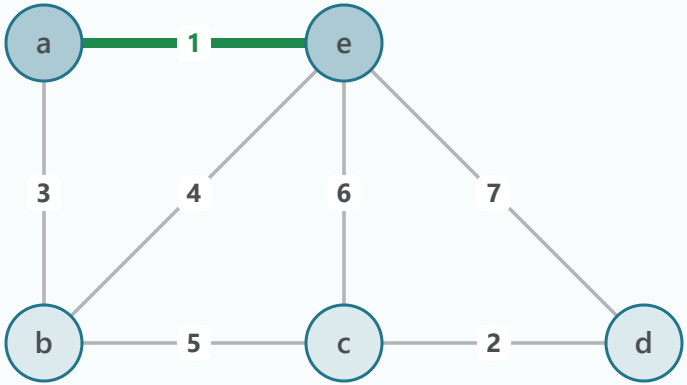


MST	—
w(T)	0
Trees	{a} {b} {c} {d} {e}

Step 1: sort the edges by **increasing weight**: ae, cd, ab, be, bc, ec, ed.

Kruskal Step by Step

ae	cd	ab	be	bc	ec	ed
1	2	3	4	5	6	7

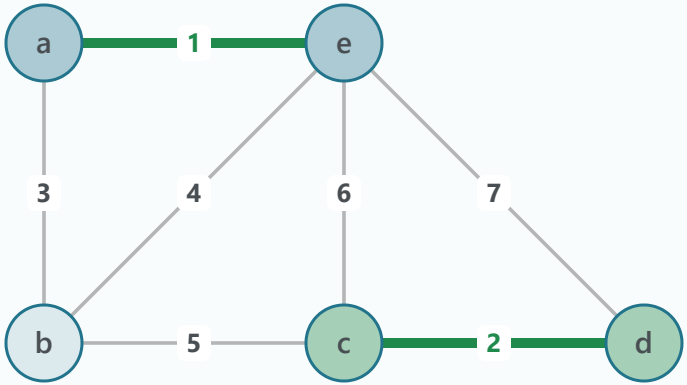


MST	ae
w(T)	1
Trees	{a, e} {b} {c} {d}

ae = 1: a and e are in different trees → add the edge; the two trees merge.

Kruskal Step by Step

ae	cd	ab	be	bc	ec	ed
1	2	3	4	5	6	7

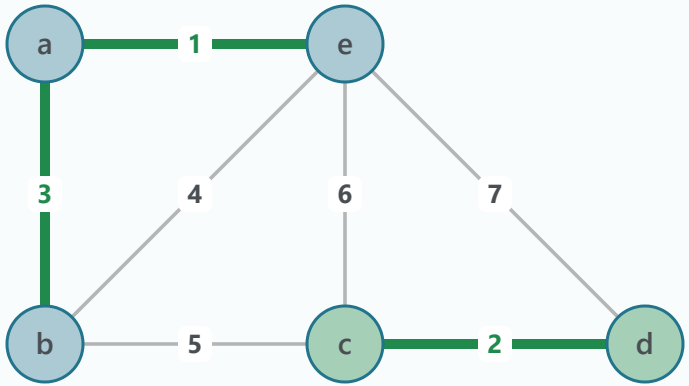


MST	ae, cd
w(T)	3
Trees	{a, e} {c, d} {b}

cd = 2: c and d are in different trees → add the edge; the two trees merge.

Kruskal Step by Step

ae	cd	ab	be	bc	ec	ed
1	2	3	4	5	6	7

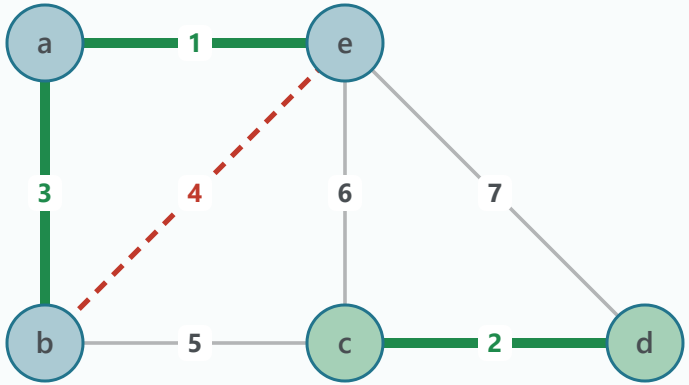


MST	ae, cd, ab
w(T)	6
Trees	{a, b, e} {c, d}

ab = 3: a and b are in different trees → add the edge; the two trees merge.

Kruskal Step by Step

ae	cd	ab	be	bc	ec	ed
1	2	3	4	5	6	7

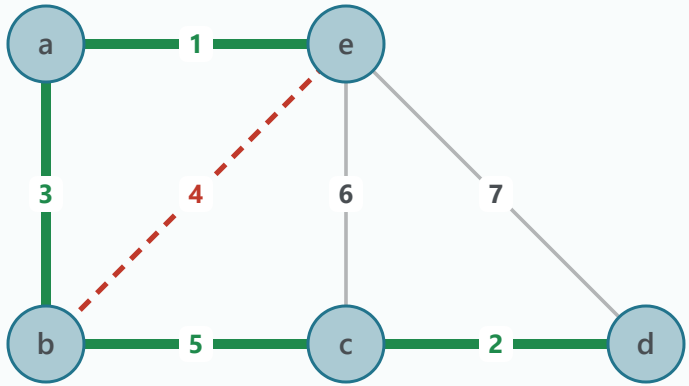


MST	ae, cd, ab
w(T)	6
Trees	{a, b, e} {c, d}

be = 4: b and e are already **in the same tree** → adding it would close a cycle, so skip it.

Kruskal Step by Step

ae	cd	ab	be	bc	ec	ed
1	2	3	4	5	6	7

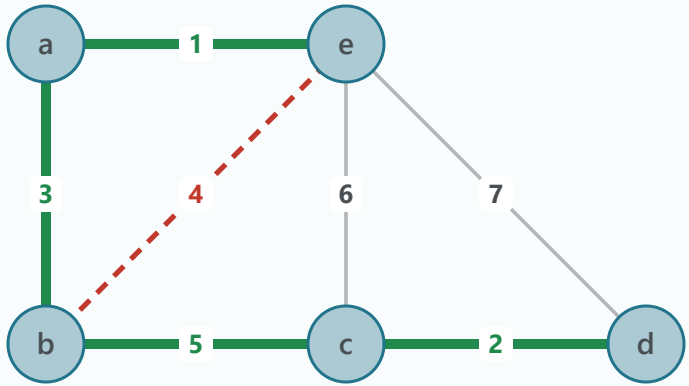


MST	ae, cd, ab, bc
w(T)	11
Trees	{a, b, c, d, e}

bc = 5: b and c are in different trees → add the edge; the two trees merge.

Kruskal Step by Step

ae	cd	ab	be	bc	ec	ed
1	2	3	4	5	6	7



MST	ae, cd, ab, bc
w(T)	11
Trees	one tree

V − 1 = 4 edges chosen: **the tree is complete**, w(T) = 11. There is no need to look at ec and ed.

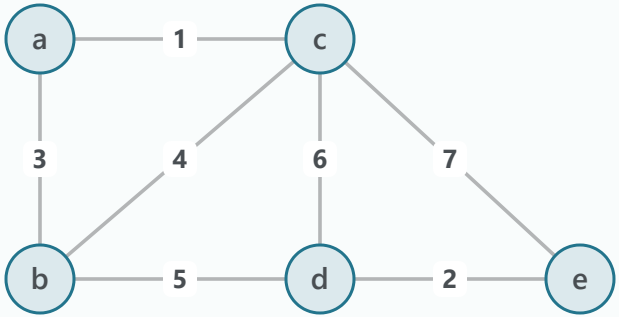
Union-Find (Disjoint Sets)

- Keeps a collection of disjoint sets. Each set is a **tree** whose root is the set's **representative**. The parent array `parent[]` : `parent[i]` is i's parent, and a root has `parent[r] = r`.
- **makeSets(V)**: each vertex is its own set → `parent[i] = i`.
- **find(x)**: climb to the root and return it.
- **union(x, y)**: hang one root under the other.
- In Kruskal: if `find(u) == find(v)`, u and v are in the same tree → the edge closes a **cycle**.

```
int find(int[] parent, int i) {
  if (parent[i] != i)
    parent[i] = find(parent, parent[i]); // compress
  return parent[i];
}
void union(int[] parent, int x, int y) {
  parent[find(parent, x)] = find(parent, y);
}
```

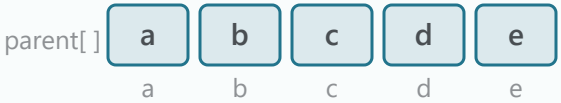
Two improvements: *path compression* and *union by rank*. Together they give **O(α(V))** amortized per operation; α (the inverse Ackermann function) is ≤ 4 in practice.

Cycle Checking with Union-Find



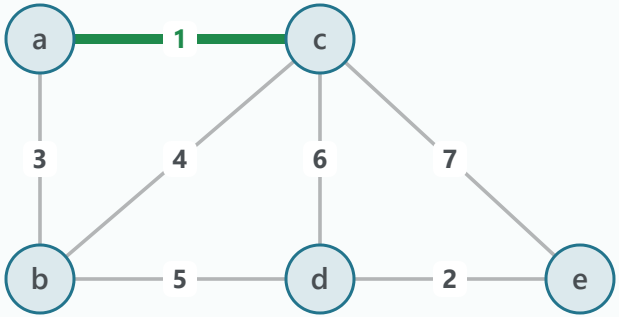
Disjoint-set forest (arrow: parent)



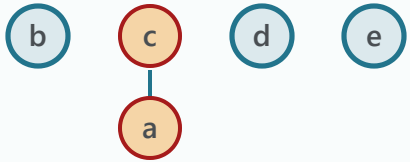


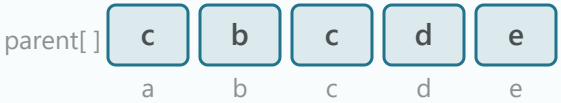
`makeSets(5)` : every vertex is the root of its own set, `parent[i] = i` . Edges: ac 1, de 2, ab 3, bc 4, bd 5, cd 6, ce 7.

Cycle Checking with Union-Find



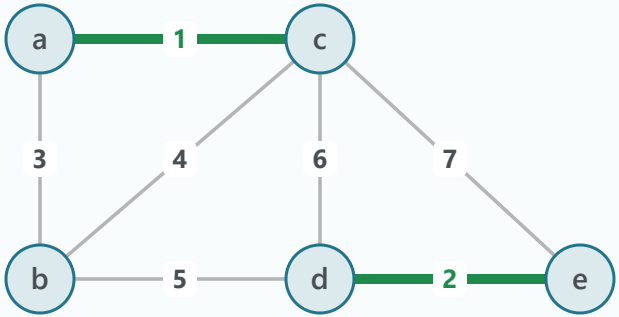
Disjoint-set forest (arrow: parent)



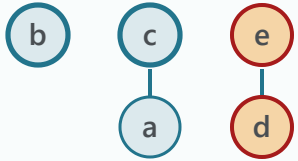


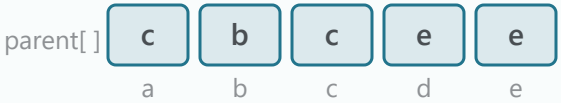
ac = 1: find(a) = a, find(c) = c → different → the edge joins the MST; `parent[a] = c`.

Cycle Checking with Union-Find



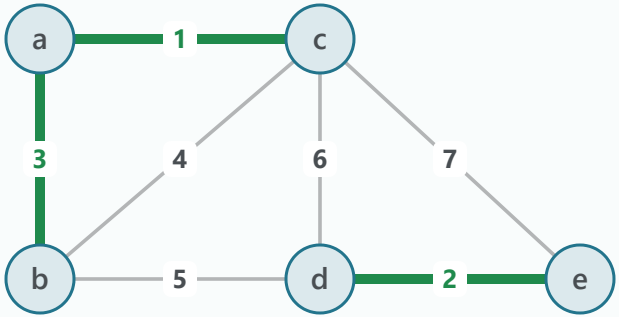
Disjoint-set forest (arrow: parent)



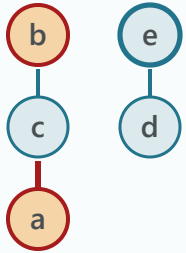


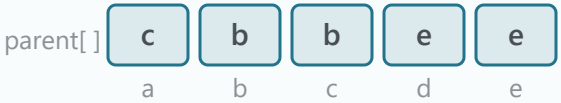
de = 2: find(d) = d, find(e) = e → different → the edge joins the MST; `parent[d] = e`.

Cycle Checking with Union-Find



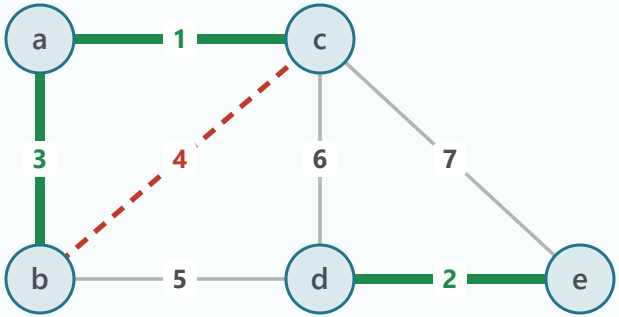
Disjoint-set forest (arrow: parent)



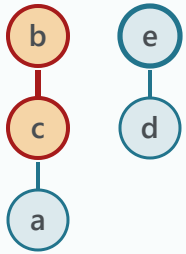


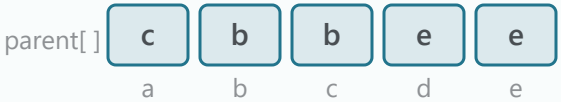
ab = 3: find(a) = c, find(b) = b → different → the edge joins the MST; `parent[c] = b`.

Cycle Checking with Union-Find



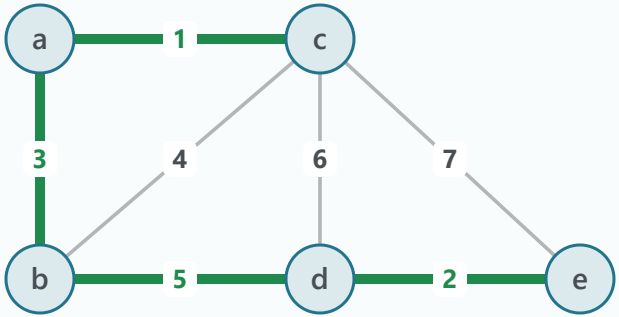
Disjoint-set forest (arrow: parent)



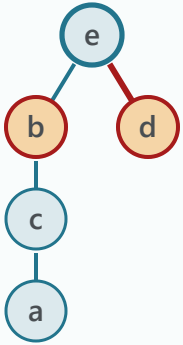


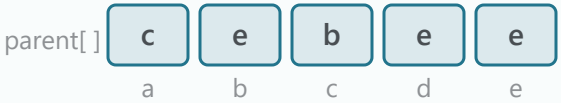
bc = 4: find(b) = b (path b), find(c) = b → **same root**: a cycle, so skip the edge.

Cycle Checking with Union-Find



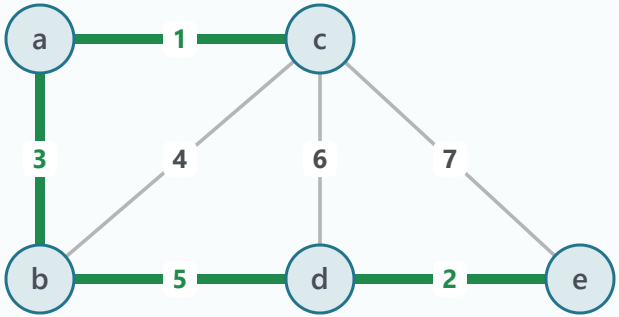
Disjoint-set forest (arrow: parent)



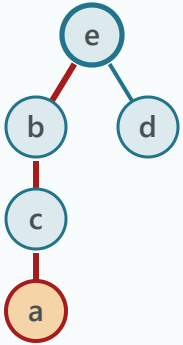


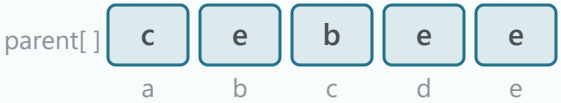
bd = 5: find(b) = b, find(d) = e → different → the edge joins the MST; `parent[b] = e`.

Cycle Checking with Union-Find



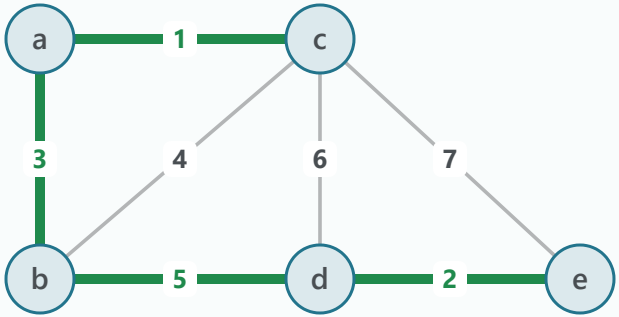
Disjoint-set forest (arrow: parent)



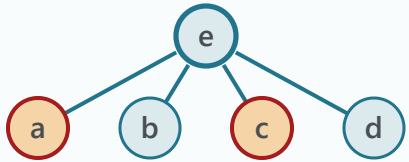


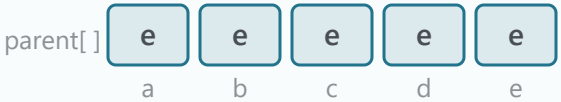
The MST is complete (4 edges, w = 11). But the forest has become **a long chain**: `find(a)` follows a→c→b→e (3 steps).

Cycle Checking with Union-Find



Disjoint-set forest (arrow: parent)



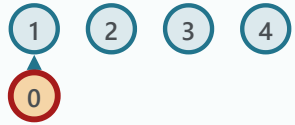


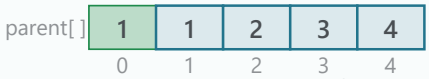
Path compression: as `find(a)` returns, it points every vertex on the path straight at the root (parent[a] = parent[c] = e). Later lookups take 1 step.

Union-Find: Tracing `find` and `union`

```
1 int find(int[] parent, int i) {
2   if (parent[i] != i)
3     parent[i] = find(parent, parent[i]); // compress
4   return parent[i];
5 }
6 void union(int[] parent, int x, int y) {
▶   parent[find(parent, x)] = find(parent, y);
8 }
```

disjoint-set forest (arrow: parent)





Initially each element is the root of its own set: parent[i] = i. `union(0, 1)` : find(0) = 0, find(1) = 1 → parent[0] = 1; 0 now hangs under 1.

Step 1 / 12

VARIABLES

`i = –`

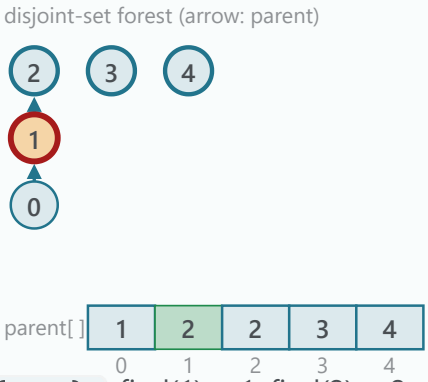
`parent = [1, 1, 2, 3, 4]`

CALL STACK

`union(0, 1)`

Union-Find: Tracing `find` and `union`

```
1 int find(int[] parent, int i) {
2   if (parent[i] != i)
3     parent[i] = find(parent, parent[i]); // compress
4   return parent[i];
5 }
6 void union(int[] parent, int x, int y) {
▶   parent[find(parent, x)] = find(parent, y);
8 }
```



`union(1, 2)` : find(1) = 1, find(2) = 2 → parent[1] = 2. A root was hung under a root, and the chain grew: 0 → 1 → 2.

Step 2 / 12

VARIABLES

`i = –`

`parent = [1, 2, 2, 3, 4]`

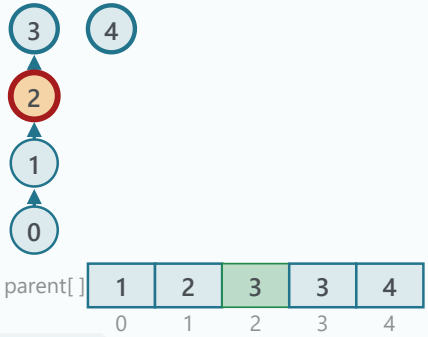
CALL STACK

`union(1, 2)`

Union-Find: Tracing `find` and `union`

```
1 int find(int[] parent, int i) {
2   if (parent[i] != i)
3     parent[i] = find(parent, parent[i]); // compress
4   return parent[i];
5 }
6 void union(int[] parent, int x, int y) {
▶   parent[find(parent, x)] = find(parent, y);
8 }
```

disjoint-set forest (arrow: parent)



`union(2, 3)` : parent[2] = 3. Chain 0 → 1 → 2 → 3: find(0) now takes 3 steps.

Step 3 / 12

VARIABLES

`i = –`

`parent = [1, 2, 3, 3, 4]`

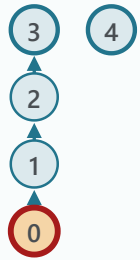
CALL STACK

`union(2, 3)`

Union-Find: Tracing `find` and `union`

```
1 int find(int[] parent, int i) {
▶   if (parent[i] != i)
▶     parent[i] = find(parent, parent[i]); //
  compress
4   return parent[i];
5 }
6 void union(int[] parent, int x, int y) {
7   parent[find(parent, x)] = find(parent, y);
8 }
```

disjoint-set forest (arrow: parent)



parent[ ]	1	2	3	3	4
	0	1	2	3	4

parent[0] = 1 ≠ 0: 0 is not a root → call `find(1)`.

Step 4 / 12

VARIABLES

`i = 0`

`parent = [1, 2, 3, 3, 4]`

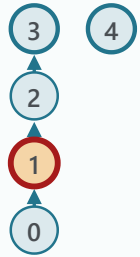
CALL STACK

`find(0)`

Union-Find: Tracing `find` and `union`

```
1 int find(int[] parent, int i) {
▶   if (parent[i] != i)
▶     parent[i] = find(parent, parent[i]); //
  compress
4   return parent[i];
5 }
6 void union(int[] parent, int x, int y) {
7   parent[find(parent, x)] = find(parent, y);
8 }
```

disjoint-set forest (arrow: parent)



parent[ ]	1	2	3	3	4
	0	1	2	3	4

parent[1] = 2 ≠ 1: 1 is not a root → call `find(2)`.

Step 5 / 12

VARIABLES

`i = 1`

`parent = [1, 2, 3, 3, 4]`

CALL STACK

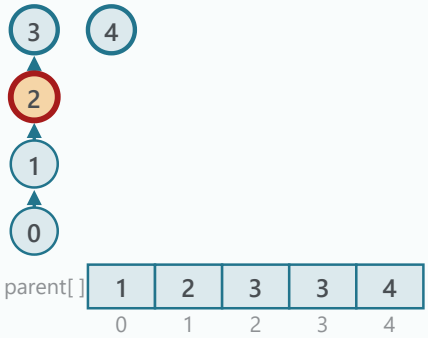
`find(1)`

`find(0)`

Union-Find: Tracing `find` and `union`

```
1 int find(int[] parent, int i) {
▶   if (parent[i] != i)
▶     parent[i] = find(parent, parent[i]); //
  compress
4   return parent[i];
5 }
6 void union(int[] parent, int x, int y) {
7   parent[find(parent, x)] = find(parent, y);
8 }
```

disjoint-set forest (arrow: parent)



parent[2] = 3 ≠ 2: 2 is not a root → call `find(3)`.

Step 6 / 12

VARIABLES

`i = 2`

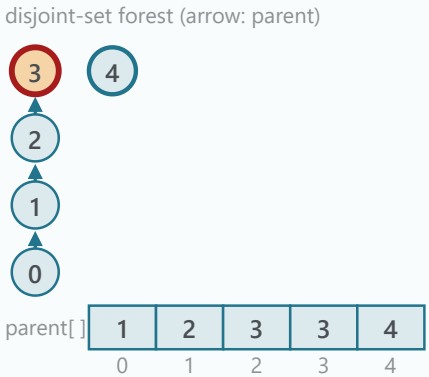
`parent = [1, 2, 3, 3, 4]`

CALL STACK

- `find(2)`
- `find(1)`
- `find(0)`

Union-Find: Tracing `find` and `union`

```
1 int find(int[] parent, int i) {
▶   if (parent[i] != i)
3     parent[i] = find(parent, parent[i]); //
  compress
▶   return parent[i];
5 }
6 void union(int[] parent, int x, int y) {
7   parent[find(parent, x)] = find(parent, y);
8 }
```



parent[3] = 3: 3 is a **root**. Return root 3; the recursion starts to unwind.

Step 7 / 12

VARIABLES

`i = 3`

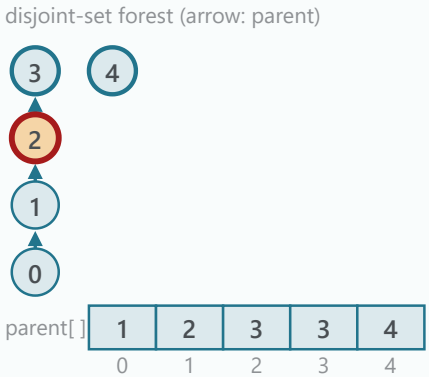
`parent = [1, 2, 3, 3, 4]`

CALL STACK

- `find(3)`
- `find(2)`
- `find(1)`
- `find(0)`

Union-Find: Tracing `find` and `union`

```
1 int find(int[] parent, int i) {
2   if (parent[i] != i)
      parent[i] = find(parent, parent[i]); // compress
    return parent[i];
5 }
6 void union(int[] parent, int x, int y) {
7   parent[find(parent, x)] = find(parent, y);
8 }
```



On return, parent[2] = 3 (it already pointed to the root; unchanged).

Step 8 / 12

VARIABLES

`i = 2`

`parent = [1, 2, 3, 3, 4]`

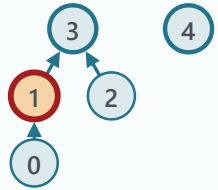
CALL STACK

- `find(2)`
- `find(1)`
- `find(0)`

Union-Find: Tracing `find` and `union`

```
1 int find(int[] parent, int i) {
2   if (parent[i] != i)
▶     parent[i] = find(parent, parent[i]); // compress
▶   return parent[i];
5 }
6 void union(int[] parent, int x, int y) {
7   parent[find(parent, x)] = find(parent, y);
8 }
```

disjoint-set forest (arrow: parent)



parent[ ]	1	3	3	3	4
	0	1	2	3	4

On return, parent[1] = 3: 1 now points straight to the root (2 → 3). **Path compression**.

Step 9 / 12

VARIABLES

`i = 1`

`parent = [1, 3, 3, 3, 4]`

CALL STACK

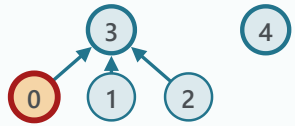
`find(1)`

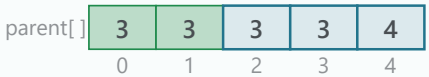
`find(0)`

Union-Find: Tracing `find` and `union`

```
1 int find(int[] parent, int i) {
2   if (parent[i] != i)
▶     parent[i] = find(parent, parent[i]); // compress
▶   return parent[i];
5 }
6 void union(int[] parent, int x, int y) {
7   parent[find(parent, x)] = find(parent, y);
8 }
```

disjoint-set forest (arrow: parent)





On return, parent[0] = 3: 0 now points straight to the root (1 → 3). **Path compression**.

Step 10 / 12

VARIABLES

`i = 0`

`parent = [3, 3, 3, 3, 4]`

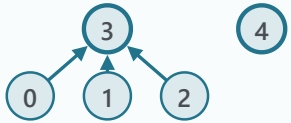
CALL STACK

`find(0)`

Union-Find: Tracing `find` and `union`

```
1 int find(int[] parent, int i) {
2   if (parent[i] != i)
3     parent[i] = find(parent, parent[i]); // compress
▶   return parent[i];
5 }
6 void union(int[] parent, int x, int y) {
7   parent[find(parent, x)] = find(parent, y);
8 }
```

disjoint-set forest (arrow: parent)



parent[ ]	3	3	3	3	4
	0	1	2	3	4

`find(0)` = 3. All of 0, 1, 2 on the path now point directly to 3: later find calls finish in one step.

Step 11 / 12

VARIABLES

`i = –`

`parent = [3, 3, 3, 3, 4]`

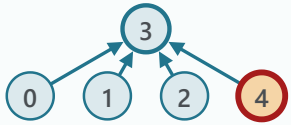
CALL STACK

empty

Union-Find: Tracing `find` and `union`

```
1 int find(int[] parent, int i) {
2   if (parent[i] != i)
3     parent[i] = find(parent, parent[i]); // compress
4   return parent[i];
5 }
6 void union(int[] parent, int x, int y) {
▶   parent[find(parent, x)] = find(parent, y);
8 }
```

disjoint-set forest (arrow: parent)



parent[ ]	3	3	3	3	3
	0	1	2	3	4

`union(4, 0)` : find(4) = 4, find(0) = 3 (one step!) → parent[4] = 3. All elements are in one set with root 3.

Step 12 / 12

VARIABLES

`i = –`

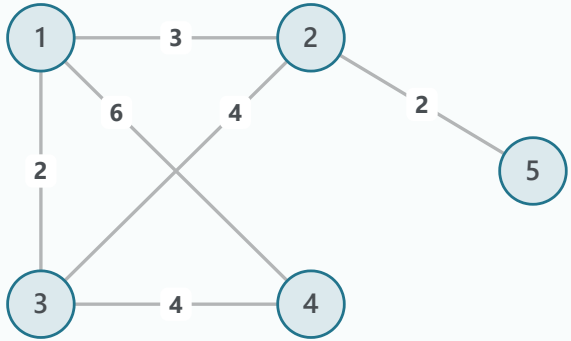
`parent = [3, 3, 3, 3, 3]`

CALL STACK

`union(4, 0)`

Kruskal: Step Through the Code

```
1  void kruskal(int V, List<Edge> es) {
▶    Collections.sort(es);          // ascending
3    int[] parent = new int[V + 1];
4    for (int i = 1; i <= V; i++) parent[i] = i;
5    List<Edge> tree = new ArrayList<>();
6    int k = 0;
7    while (tree.size() < V - 1) {
8      Edge e = es.get(k++);
9      int x = find(parent, e.u);
10     int y = find(parent, e.v);
11     if (x != y) {                 // different
   trees
12       tree.add(e);
13       parent[x] = y;              // union
14     }
15   }
16 }
```



Sort the edges: (1,3)=2, (2,5)=2, (1,2)=3, (2,3)=4, (3,4)=4, (1,4)=6.

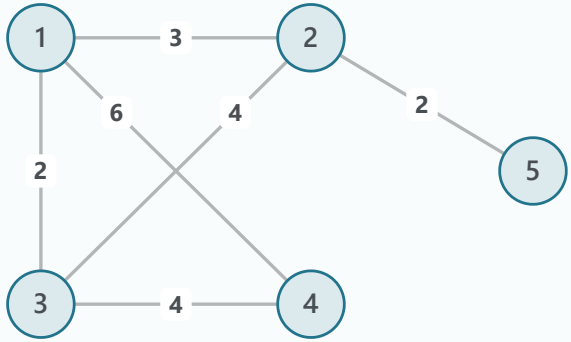
Step 1 / 13

VARIABLES

`V = 5` `k = 0` `tree = []` `w(T) = 0`

Kruskal: Step Through the Code

```
1  void kruskal(int V, List<Edge> es) {
2    Collections.sort(es);          // ascending
▶    int[] parent = new int[V + 1];
▶    for (int i = 1; i <= V; i++) parent[i] = i;
5    List<Edge> tree = new ArrayList<>();
6    int k = 0;
7    while (tree.size() < V - 1) {
8      Edge e = es.get(k++);
9      int x = find(parent, e.u);
10     int y = find(parent, e.v);
11     if (x != y) {                // different
   trees
12       tree.add(e);
13       parent[x] = y;             // union
14     }
15   }
16 }
```



`parent[i] = i` : each vertex is a separate tree (5 trees).

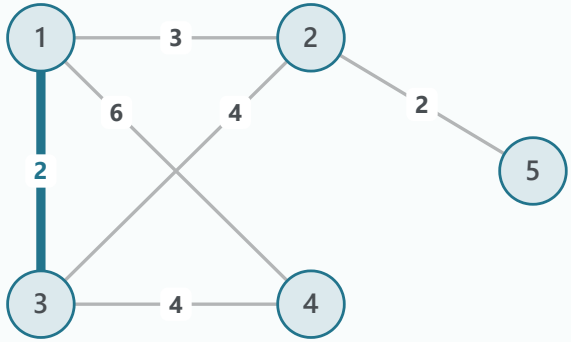
Step 2 / 13

VARIABLES

`V = 5` `parent = [1, 2, 3, 4, 5]` `k = 0` `tree = []` `w(T) = 0`

Kruskal: Step Through the Code

```
1  void kruskal(int V, List<Edge> es) {
2    Collections.sort(es);          // ascending
3    int[] parent = new int[V + 1];
4    for (int i = 1; i <= V; i++) parent[i] = i;
5    List<Edge> tree = new ArrayList<>();
6    int k = 0;
7    while (tree.size() < V - 1) {
▶      Edge e = es.get(k++);
▶      int x = find(parent, e.u);
▶      int y = find(parent, e.v);
11     if (x != y) {                  // different
   trees
12       tree.add(e);
13       parent[x] = y;               // union
14     }
15   }
16 }
```



e = (1,3), weight 2: `find(1)` = 1, `find(3)` = 3.

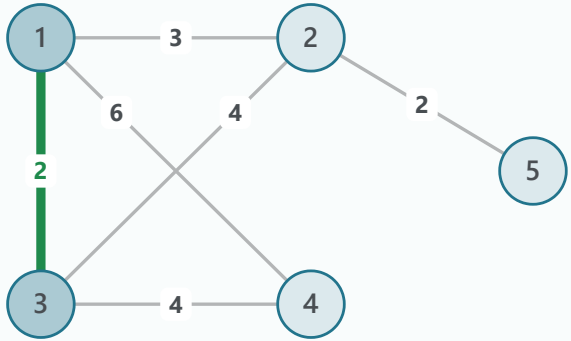
Step 3 / 13

VARIABLES

`V = 5` `parent = [1, 2, 3, 4, 5]` `k = 1`
`e = (1,3,2)` `x = 1` `y = 3` `tree = []`
`w(T) = 0`

Kruskal: Step Through the Code

```
1  void kruskal(int V, List<Edge> es) {
2    Collections.sort(es);          // ascending
3    int[] parent = new int[V + 1];
4    for (int i = 1; i <= V; i++) parent[i] = i;
5    List<Edge> tree = new ArrayList<>();
6    int k = 0;
7    while (tree.size() < V - 1) {
8      Edge e = es.get(k++);
9      int x = find(parent, e.u);
10     int y = find(parent, e.v);
11     if (x != y) {                // different trees
▶        tree.add(e);
▶        parent[x] = y;             // union
14     }
15   }
16 }
```



x ≠ y → (1,3) joins the MST, `parent[1] = 3`. v1 and v3 merge.

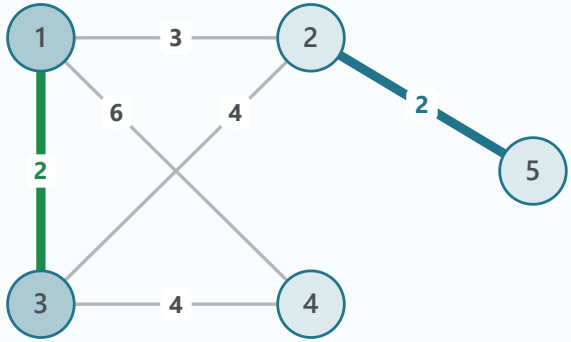
Step 4 / 13

VARIABLES

`V = 5` `parent = [3, 2, 3, 4, 5]` `k = 1` `e = (1,3,2)` `x = 1` `y = 3` `tree = [(1,3)]` `w(T) = 2`

Kruskal: Step Through the Code

```
1  void kruskal(int V, List<Edge> es) {
2    Collections.sort(es);          // ascending
3    int[] parent = new int[V + 1];
4    for (int i = 1; i <= V; i++) parent[i] = i;
5    List<Edge> tree = new ArrayList<>();
6    int k = 0;
7    while (tree.size() < V - 1) {
▶      Edge e = es.get(k++);
▶      int x = find(parent, e.u);
▶      int y = find(parent, e.v);
11     if (x != y) {                  // different trees
12       tree.add(e);
13       parent[x] = y;               // union
14     }
15   }
16 }
```



e = (2,5), weight 2: `find(2)` = 2, `find(5)` = 5.

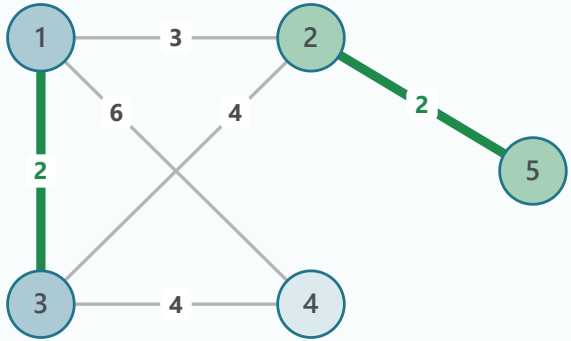
Step 5 / 13

VARIABLES

`V = 5` `parent = [3, 2, 3, 4, 5]` `k = 2` `e = (2,5,2)` `x = 2` `y = 5` `tree = [(1,3)]` `w(T) = 2`

Kruskal: Step Through the Code

```
1  void kruskal(int V, List<Edge> es) {
2    Collections.sort(es);          // ascending
3    int[] parent = new int[V + 1];
4    for (int i = 1; i <= V; i++) parent[i] = i;
5    List<Edge> tree = new ArrayList<>();
6    int k = 0;
7    while (tree.size() < V - 1) {
8      Edge e = es.get(k++);
9      int x = find(parent, e.u);
10     int y = find(parent, e.v);
11     if (x != y) {                // different trees
▶        tree.add(e);
▶        parent[x] = y;             // union
14     }
15   }
16 }
```



x ≠ y → (2,5) joins the MST, `parent[2] = 5`. v2 and v5 merge.

Step 6 / 13

VARIABLES

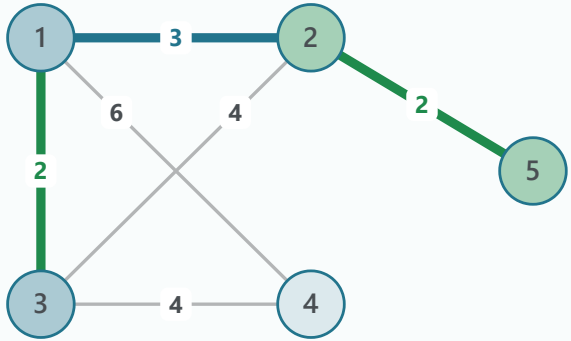
`V = 5` `parent = [3, 5, 3, 4, 5]` `k = 2`

`e = (2,5,2)` `x = 2` `y = 5`

`tree = [(1,3) (2,5)]` `w(T) = 4`

Kruskal: Step Through the Code

```
1  void kruskal(int V, List<Edge> es) {
2    Collections.sort(es);          // ascending
3    int[] parent = new int[V + 1];
4    for (int i = 1; i <= V; i++) parent[i] = i;
5    List<Edge> tree = new ArrayList<>();
6    int k = 0;
7    while (tree.size() < V - 1) {
▶      Edge e = es.get(k++);
▶      int x = find(parent, e.u);
▶      int y = find(parent, e.v);
11     if (x != y) {                  // different trees
12       tree.add(e);
13       parent[x] = y;               // union
14     }
15   }
16 }
```



e = (1,2), weight 3: `find(1)` = 3, `find(2)` = 5.

Step 7 / 13

VARIABLES

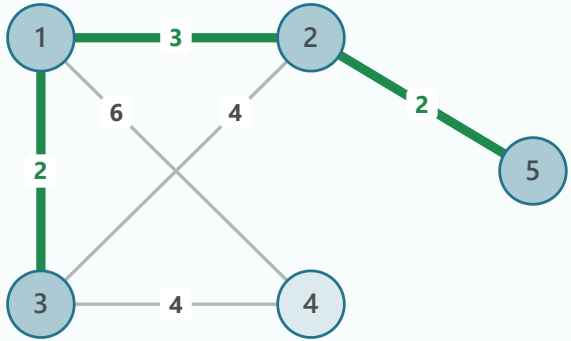
`V = 5` `parent = [3, 5, 3, 4, 5]` `k = 3`

`e = (1,2,3)` `x = 3` `y = 5`

`tree = [(1,3) (2,5)]` `w(T) = 4`

Kruskal: Step Through the Code

```
1  void kruskal(int V, List<Edge> es) {
2    Collections.sort(es);          // ascending
3    int[] parent = new int[V + 1];
4    for (int i = 1; i <= V; i++) parent[i] = i;
5    List<Edge> tree = new ArrayList<>();
6    int k = 0;
7    while (tree.size() < V - 1) {
8      Edge e = es.get(k++);
9      int x = find(parent, e.u);
10     int y = find(parent, e.v);
11     if (x != y) {                // different trees
▶        tree.add(e);
▶        parent[x] = y;             // union
14     }
15   }
16 }
```



x ≠ y → (1,2) joins the MST, `parent[3] = 5`. The two earlier trees merge.

Step 8 / 13

VARIABLES

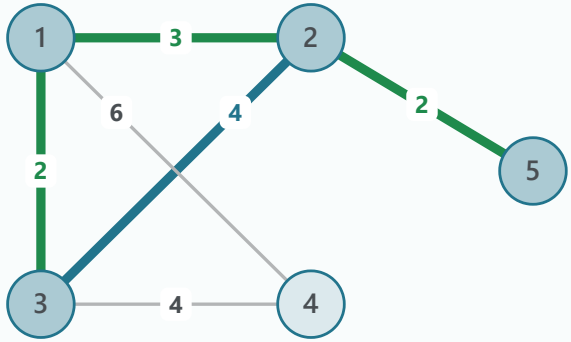
`V = 5` `parent = [3, 5, 5, 4, 5]` `k = 3`

`e = (1,2,3)` `x = 3` `y = 5`

`tree = [(1,3) (2,5) (1,2)]` `w(T) = 7`

Kruskal: Step Through the Code

```
1  void kruskal(int V, List<Edge> es) {
2    Collections.sort(es);          // ascending
3    int[] parent = new int[V + 1];
4    for (int i = 1; i <= V; i++) parent[i] = i;
5    List<Edge> tree = new ArrayList<>();
6    int k = 0;
7    while (tree.size() < V - 1) {
▶      Edge e = es.get(k++);
▶      int x = find(parent, e.u);
▶      int y = find(parent, e.v);
11     if (x != y) {                  // different trees
12       tree.add(e);
13       parent[x] = y;               // union
14     }
15   }
16 }
```



e = (2,3), weight 4: `find(2)` = 5, `find(3)` = 5.

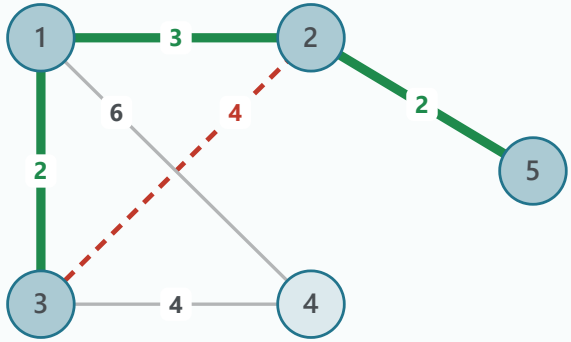
Step 9 / 13

VARIABLES

`V = 5` `parent = [3, 5, 5, 4, 5]` `k = 4` `e = (2,3,4)` `x = 5` `y = 5` `tree = [(1,3) (2,5) (1,2)]` `w(T) = 7`

Kruskal: Step Through the Code

```
1  void kruskal(int V, List<Edge> es) {
2    Collections.sort(es);          // ascending
3    int[] parent = new int[V + 1];
4    for (int i = 1; i <= V; i++) parent[i] = i;
5    List<Edge> tree = new ArrayList<>();
6    int k = 0;
7    while (tree.size() < V - 1) {
8      Edge e = es.get(k++);
9      int x = find(parent, e.u);
10     int y = find(parent, e.v);
▶      if (x != y) {                // different trees
12       tree.add(e);
13       parent[x] = y;             // union
14     }
15   }
16 }
```



x = y = 5: (2,3) does not join two different trees → **a cycle**, skip it.

Step 10 / 13

VARIABLES

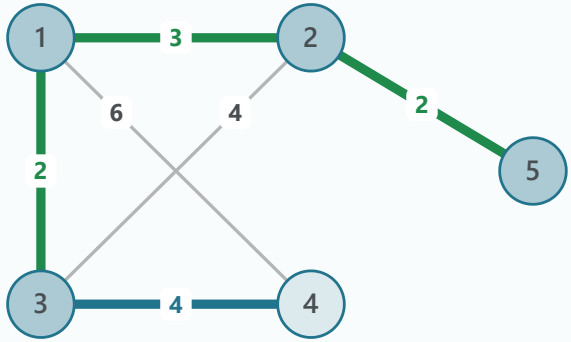
`V = 5` `parent = [3, 5, 5, 4, 5]` `k = 4`

`e = (2,3,4)` `x = 5` `y = 5`

`tree = [(1,3) (2,5) (1,2)]` `w(T) = 7`

Kruskal: Step Through the Code

```
1  void kruskal(int V, List<Edge> es) {
2    Collections.sort(es);          // ascending
3    int[] parent = new int[V + 1];
4    for (int i = 1; i <= V; i++) parent[i] = i;
5    List<Edge> tree = new ArrayList<>();
6    int k = 0;
7    while (tree.size() < V - 1) {
▶      Edge e = es.get(k++);
▶      int x = find(parent, e.u);
▶      int y = find(parent, e.v);
11     if (x != y) {                  // different trees
12       tree.add(e);
13       parent[x] = y;               // union
14     }
15   }
16 }
```



e = (3,4), weight 4: `find(3)` = 5, `find(4)` = 4.

Step 11 / 13

VARIABLES

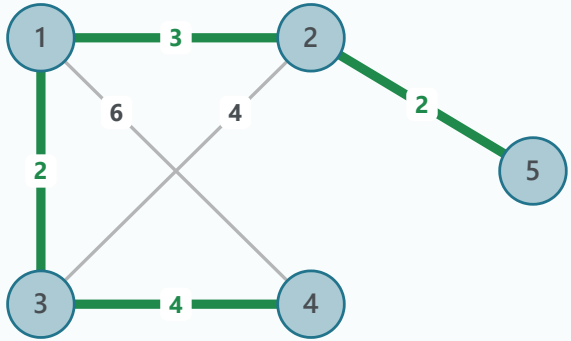
`V = 5` `parent = [3, 5, 5, 4, 5]` `k = 5`

`e = (3,4,4)` `x = 5` `y = 4`

`tree = [(1,3) (2,5) (1,2)]` `w(T) = 7`

Kruskal: Step Through the Code

```
1  void kruskal(int V, List<Edge> es) {
2    Collections.sort(es);          // ascending
3    int[] parent = new int[V + 1];
4    for (int i = 1; i <= V; i++) parent[i] = i;
5    List<Edge> tree = new ArrayList<>();
6    int k = 0;
7    while (tree.size() < V - 1) {
8      Edge e = es.get(k++);
9      int x = find(parent, e.u);
10     int y = find(parent, e.v);
11     if (x != y) {                // different trees
▶        tree.add(e);
▶        parent[x] = y;             // union
14     }
15   }
16 }
```



x ≠ y → (3,4) joins the MST, `parent[5] = 4`. v4 joins the tree; w(T) = 11.

Step 12 / 13

VARIABLES

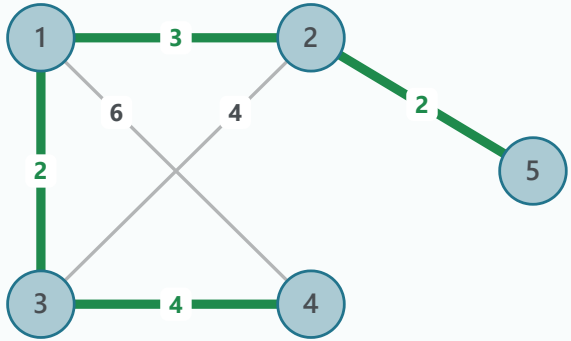
`V = 5` `parent = [3, 5, 5, 4, 4]` `k = 5`

`e = (3,4,4)` `x = 5` `y = 4`

`tree = [(1,3) (2,5) (1,2) (3,4)]` `w(T) = 11`

Kruskal: Step Through the Code

```
1  void kruskal(int V, List<Edge> es) {
2    Collections.sort(es);          // ascending
3    int[] parent = new int[V + 1];
4    for (int i = 1; i <= V; i++) parent[i] = i;
5    List<Edge> tree = new ArrayList<>();
6    int k = 0;
▶    while (tree.size() < V - 1) {
8      Edge e = es.get(k++);
9      int x = find(parent, e.u);
10     int y = find(parent, e.v);
11     if (x != y) {                // different
   trees
12       tree.add(e);
13       parent[x] = y;             // union
14     }
15   }
16 }
```



`tree.size() = 4 = V − 1` : the loop ends. (1,4)=6 is never examined. **w(T) = 11**.

Step 13 / 13

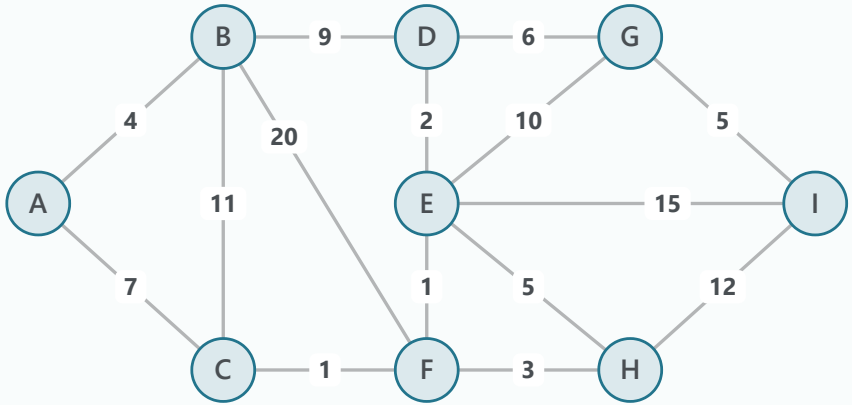
VARIABLES

`V = 5` `parent = [3, 5, 5, 4, 4]` `k = 5`

`tree = [(1,3) (2,5) (1,2) (3,4)]` `w(T) = 11`

Kruskal Example: Nine Vertices

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

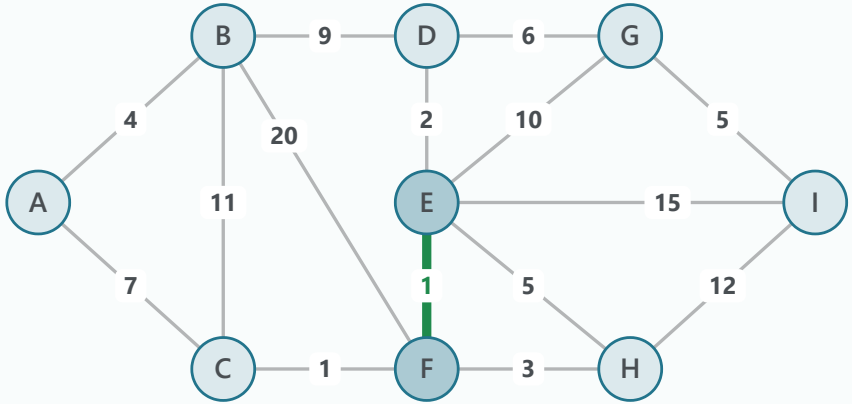


Sets: singletons · Weight: 0 · Connected: NO

Edges sorted: 1, 1, 2, 3, 4, 5, 5, 6, 7, 9, 10, 11, 12, 15, 20. No edge chosen yet.

Kruskal Example: Nine Vertices

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

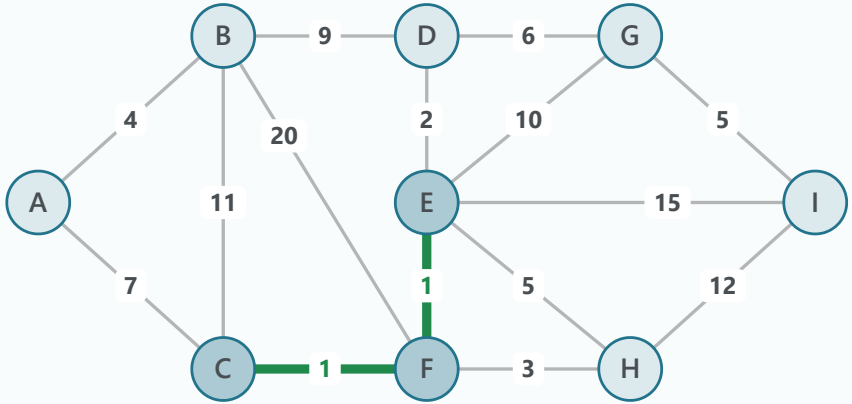


Sets: {E, F} · Weight: 1 · Connected: NO

EF = 1 added (two different sets). Total weight 1.

Kruskal Example: Nine Vertices

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

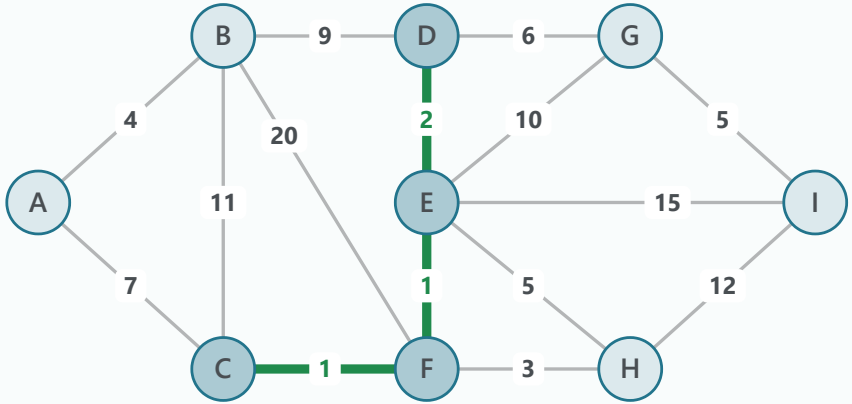


Sets: {C, E, F} · Weight: 2 · Connected: NO

CF = 1 added (two different sets). Total weight 2.

Kruskal Example: Nine Vertices

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

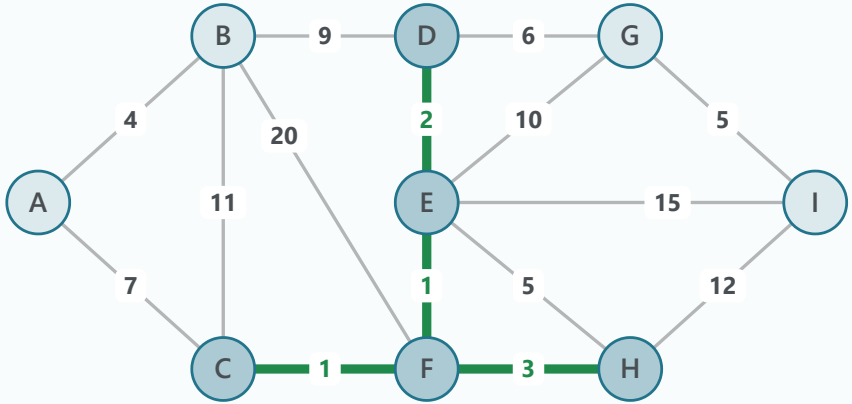


Sets: {C, D, E, F} · Weight: 4 · Connected: NO

DE = 2 added (two different sets). Total weight 4.

Kruskal Example: Nine Vertices

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

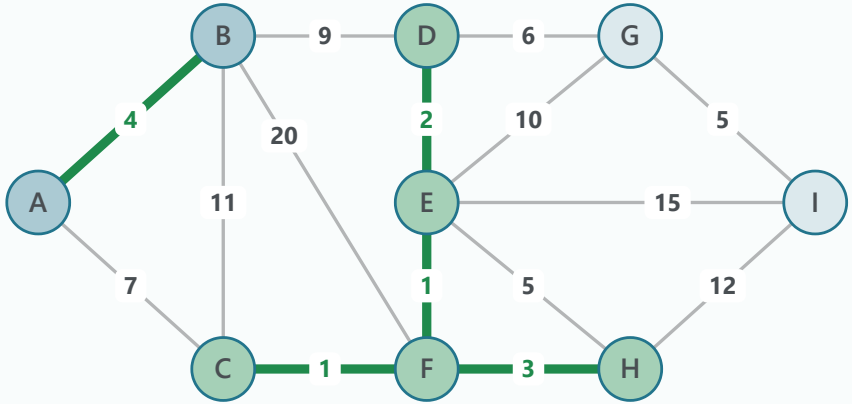


Sets: {C, D, E, F, H} · Weight: 7 · Connected: NO

FH = 3 added (two different sets). Total weight 7.

Kruskal Example: Nine Vertices

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

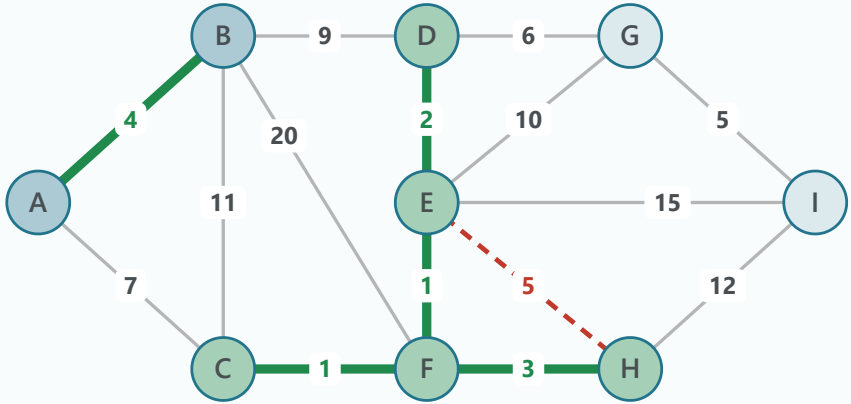


Sets: {C, D, E, F, H} {A, B} · Weight: 11 · Connected: NO

AB = 4 added (two different sets). Total weight 11.

Kruskal Example: Nine Vertices

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

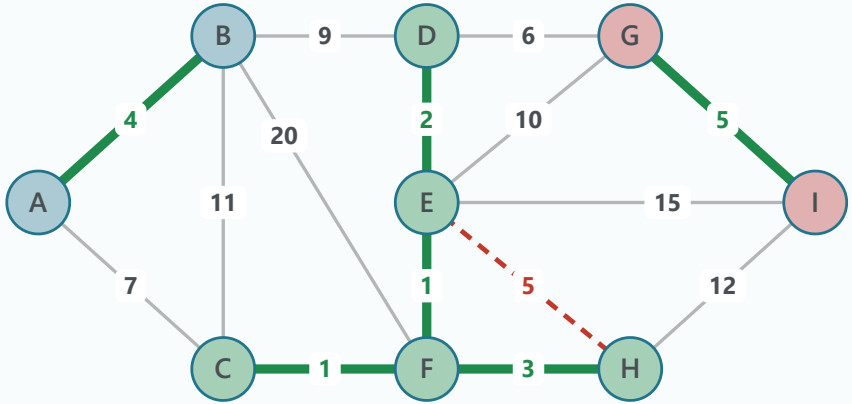


Sets: {C, D, E, F, H} {A, B} · Weight: 11 · Connected: NO

EH = 5: E and H are already in the same set (path E–F–H) → a cycle, not added. The total stays 11.

Kruskal Example: Nine Vertices

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

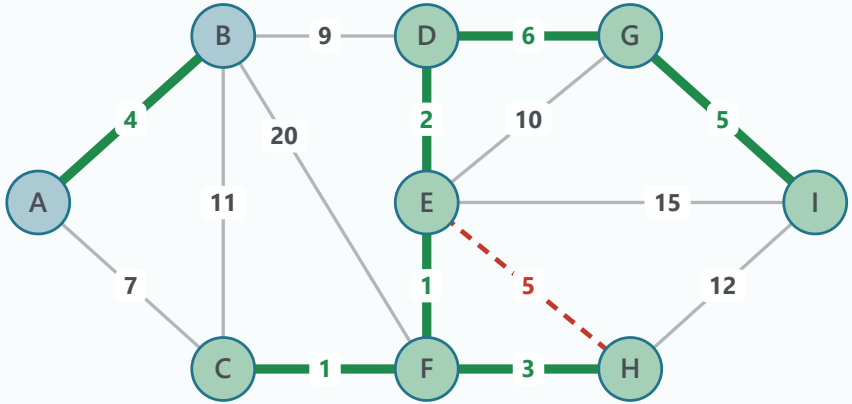


Sets: {C, D, E, F, H} {A, B} {G, I} · Weight: 16 · Connected: NO

GI = 5 added (two different sets). Total weight 16.

Kruskal Example: Nine Vertices

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

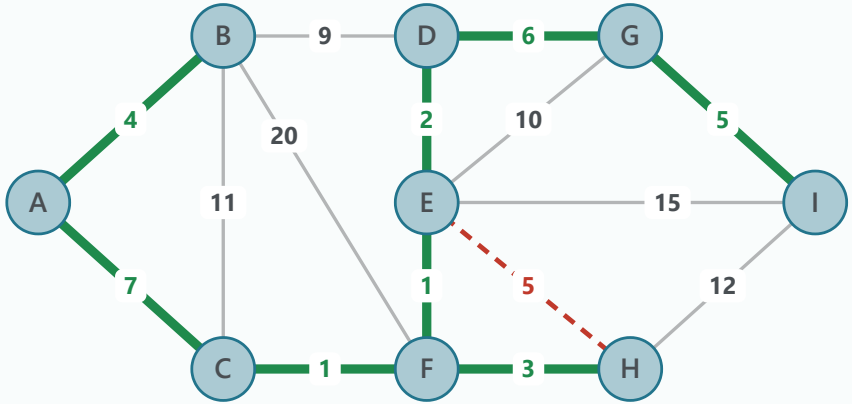


Sets: {C, D, E, F, G, H, I} {A, B} · Weight: 22 · Connected: NO

DG = 6 added (two different sets). Total weight 22.

Kruskal Example: Nine Vertices

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20



Sets: {A, B, C, D, E, F, G, H, I} · Weight: 29 · Connected: YES

AC = 7 added (two different sets). Total weight 29. 8 = V − 1 edges: **MST = 29**, and the graph is connected.

Kruskal: Complexity Analysis

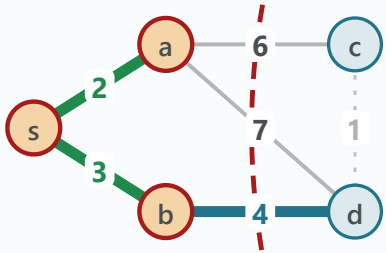
Operation	How often	Cost
Sorting the edges by weight	1	**O(E log E)**
`makeSets(V)`	1	O(V)
`find` (2 per edge)	≤ 2E	O(E · α(V)) — path compression + rank O(E log V) — rank only
`union`	V − 1	

- Total: **O(E log E)**; sorting dominates. Since $E \le V^2$, $\log E \le 2 \log V$, so $O(E \log E) = $ **O(E log V)**.
- If the edges are already sorted (or the weights are small integers, so counting sort applies), the time is nearly linear: $O(E \cdot \alpha(V))$.
- Memory: edge list + parent array (`parent`) → O(V + E).

With union by rank alone, union-find operations take O(log V) in the worst case; adding path compression brings the amortized cost per operation to **O(α(V))**. Either way, sorting determines the total.

Prim's Algorithm

- Builds a minimum-weight tree that connects all vertices; a **greedy** algorithm.
- Robert C. **Prim** (1957); V. Jarník found it earlier (1930), and E. W. Dijkstra rediscovered it (1959).
- Pick a **start vertex**; at each step, add the non-tree vertex that connects to the tree by the **cheapest edge**.
- At any moment there is **a single tree** (unlike Kruskal's forest).



cheapest crossing edge: 4

Strengths

+ Good for **dense** graphs: $O(V^2)$ with a matrix.
+ No edge sorting needed.
+ Works naturally with an adjacency list or matrix.

Weaknesses

– Performance **depends on the data structure** (array, binary heap, Fibonacci heap).
– On a disconnected graph it spans only the start vertex's component.

Prim: Steps, Keys and Parents

1. Choose a start vertex and add all its edges to a **priority queue**.
2. From the queue, take the **lightest** edge leading to a vertex not yet in the tree.
3. **Add** the new vertex on that edge **to the tree**.
4. Add the new vertex's edges to the queue; repeat steps 2–4 until every vertex is in the tree.

key[v]

The weight of the **cheapest edge** found so far that connects v to the tree. ∞ at first, 0 at the source.

parent[v] (π)

The tree end of that cheapest edge. At the end, the edges (parent[v], v) form the MST.

For the vertex taken from the queue, for each non-tree neighbor v: if `weight(u, v) < key[v]`, update the key and the parent.

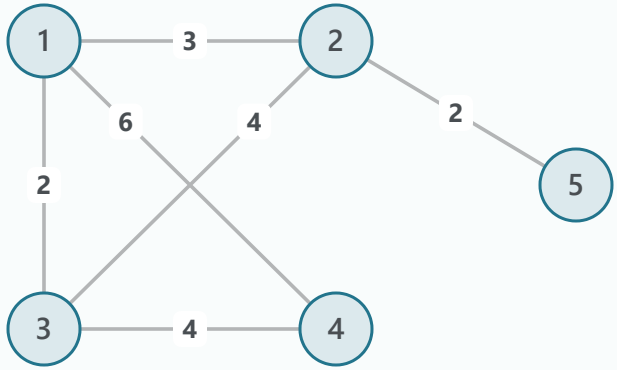
Prim: Pseudocode

```
PRIM(G, source):
  MST = [ ]
  dist = [ ]
  parent = [ ]
  visited = [ ]
  for each v in V:
    dist[v] = ∞
    visited[v] = 0
  dist[source] = 0
  pq.add(source, 0)
```

```
while pq is not empty do:
  u = pq.extractMin()
  if visited[u] == 0:
    visited[u] = 1
    for each (u, v, weight) in G[u]:
      if visited[v] == 0 and
           weight < dist[v]:
        dist[v] = weight
        parent[v] = u
        pq.add(v, weight)
for each v in V
    (except source, with dist[v] ≠ ∞):
  MST.add((parent[v], v, dist[v]))
return MST
```

This "lazy" version may add a vertex to the queue several times; the `visited` check throws away stale entries. The queue holds at most E entries → O(E log E).

Prim Step by Step: The Priority Queue



key	u	π
0	1	/
∞	2	/
∞	3	/
∞	4	/
∞	5	/

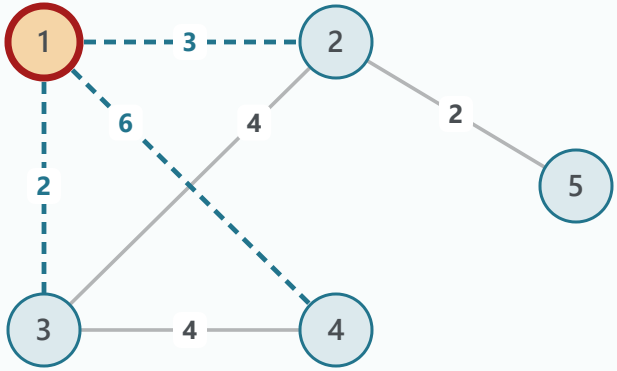
w(T) = 0

Initialization: the queue Q starts with every vertex; `key(1) = 0`, the others ∞, π = /.

Step 1 / 7

Dashed line: the cheapest connection recorded in π, not yet final. The Q table is sorted by key.

Prim Step by Step: The Priority Queue



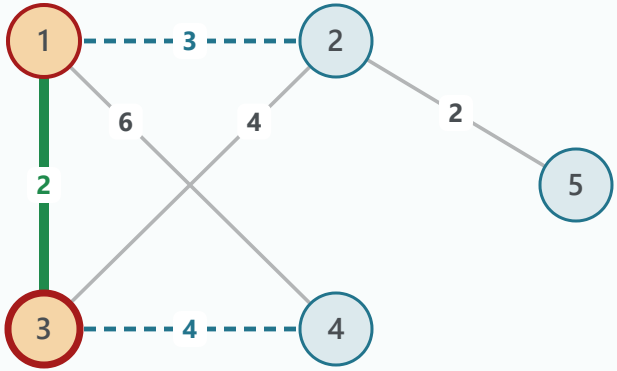
key	u	π
2	3	1
3	2	1
6	4	1
∞	5	/

w(T) = 0

Vertex 1 leaves the queue. Neighbors are updated: key(3) = 2 (1,3), key(2) = 3 (1,2), key(4) = 6 (1,4).

Dashed line: the cheapest connection recorded in π, not yet final. The Q table is sorted by key.

Prim Step by Step: The Priority Queue



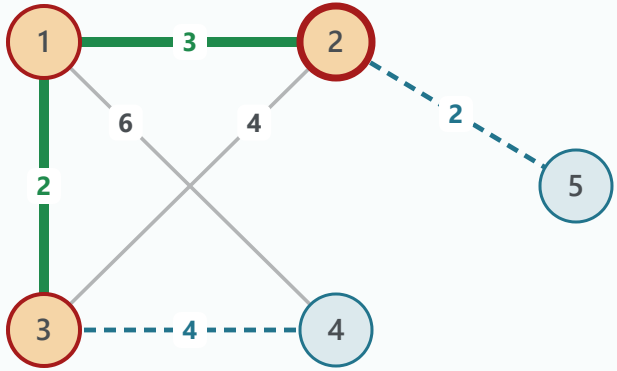
key	u	π
3	2	1
4	4	3
∞	5	/

w(T) = 2

Vertex 3 leaves; **(1,3)** is added to T. key(4) becomes 4 (3,4); (2,3) = 4 > key(2) = 3 → unchanged.

Dashed line: the cheapest connection recorded in π, not yet final. The Q table is sorted by key.

Prim Step by Step: The Priority Queue



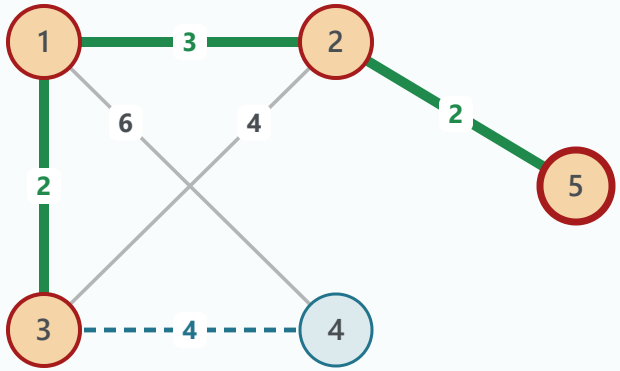
key	u	π
2	5	2
4	4	3

w(T) = 5

Vertex 2 leaves; **(1,2)** is added to T. key(5) = 2 (2,5).

Dashed line: the cheapest connection recorded in π, not yet final. The Q table is sorted by key.

Prim Step by Step: The Priority Queue



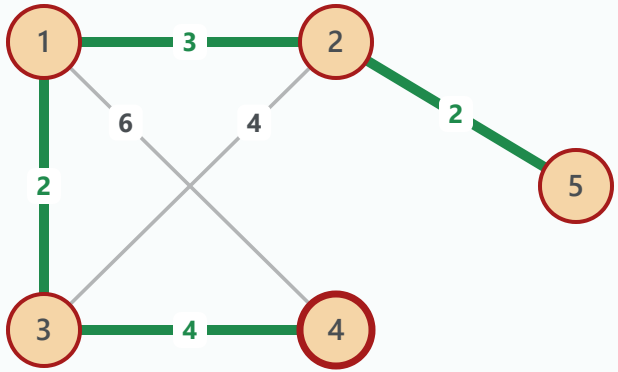
key	u	π
4	4	3

w(T) = 7

Vertex 5 leaves; **(2,5)** is added to T. It has no neighbor outside the tree: no update in Q.

Dashed line: the cheapest connection recorded in π, not yet final. The Q table is sorted by key.

Prim Step by Step: The Priority Queue



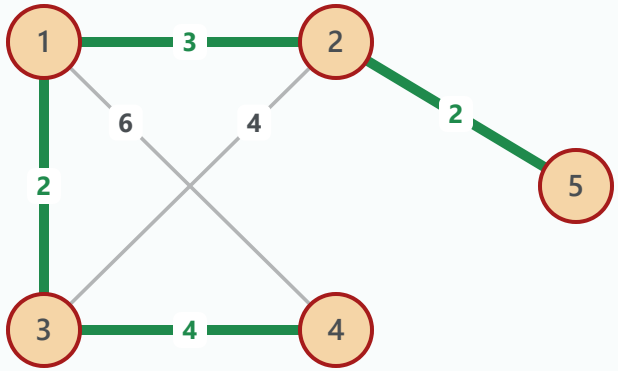
key	u	π
Q = ∅		

w(T) = 11

Vertex 4 leaves; **(3,4)** is added to T. No update in Q.

Dashed line: the cheapest connection recorded in π, not yet final. The Q table is sorted by key.

Prim Step by Step: The Priority Queue



key	u	π
Q = ∅		

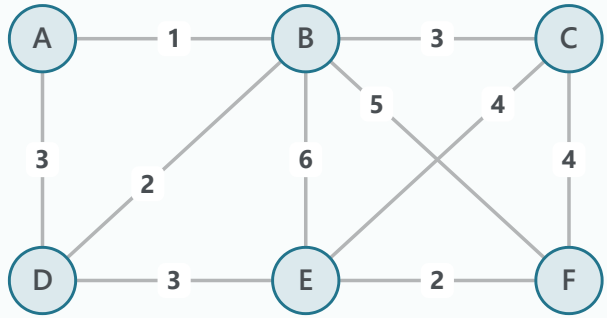
w(T) = 11

Final state: Q = ∅. MST = (1,3), (1,2), (2,5), (3,4), **w(T) = 11**: the same tree as Kruskal.

Dashed line: the cheapest connection recorded in π, not yet final. The Q table is sorted by key.

Prim: Step Through the Code (Adjacency Matrix)

```
1  void prim(int[][] G, int src) {
2    int V = G.length;
3    int[] key = new int[V];
4    int[] parent = new int[V];
5    boolean[] inTree = new boolean[V];
▶    Arrays.fill(key, Integer.MAX_VALUE);
▶    key[src] = 0;
▶    parent[src] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = minKey(key, inTree);
11     inTree[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !inTree[v]
14           && G[u][v] < key[v]) {
15         key[v] = G[u][v];
16         parent[v] = u;
17       }
18   }
19 }
```



Graph A–F, source **B** (index 1). All keys are ∞; `key[B] = 0`, `parent[B] = −1`.

Step 1 / 15

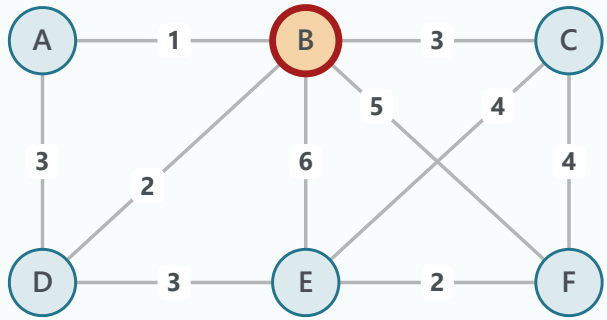
VARIABLES

`src = B` `key = [∞, 0, ∞, ∞, ∞, ∞]`

`parent = [−, −, −, −, −, −]` `inTree = {}`

Prim: Step Through the Code (Adjacency Matrix)

```
1  void prim(int[][] G, int src) {
2    int V = G.length;
3    int[] key = new int[V];
4    int[] parent = new int[V];
5    boolean[] inTree = new boolean[V];
6    Arrays.fill(key, Integer.MAX_VALUE);
7    key[src] = 0;
8    parent[src] = -1;
9    for (int s = 0; s < V - 1; s++) {
▶      int u = minKey(key, inTree);
▶      inTree[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !inTree[v]
14           && G[u][v] < key[v]) {
15         key[v] = G[u][v];
16         parent[v] = u;
17       }
18   }
19 }
```



`minKey` → u = **B** (key 0): the source enters the tree.

Step 2 / 15

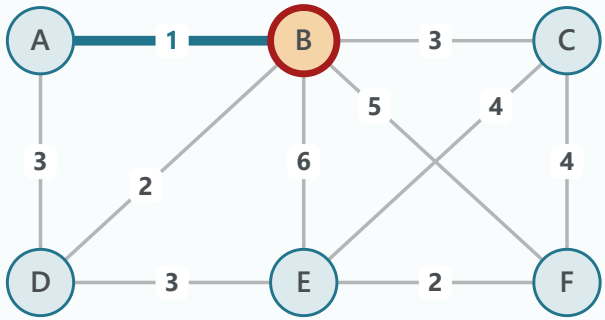
VARIABLES

`s = 0` `u = B` `key = [∞, 0, ∞, ∞, ∞, ∞]`

`parent = [-, -, -, -, -, -]` `inTree = {B}`

Prim: Step Through the Code (Adjacency Matrix)

```
1  void prim(int[][] G, int src) {
2    int V = G.length;
3    int[] key = new int[V];
4    int[] parent = new int[V];
5    boolean[] inTree = new boolean[V];
6    Arrays.fill(key, Integer.MAX_VALUE);
7    key[src] = 0;
8    parent[src] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = minKey(key, inTree);
11     inTree[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !inTree[v]
14           && G[u][v] < key[v]) {
▶          key[v] = G[u][v];
▶          parent[v] = u;
17       }
18   }
19 }
```



v = A: G[B][A] = 1 < key[A] = ∞ → `key[A] = 1`, `parent[A] = B`.

Step 3 / 15

VARIABLES

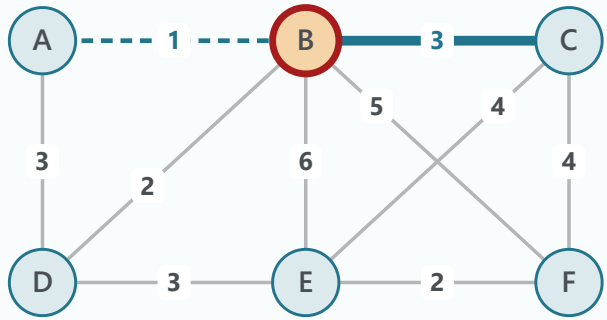
`s = 0` `u = B` `v = A`

`key = [1, 0, ∞, ∞, ∞, ∞]`

`parent = [B, -, -, -, -, -]` `inTree = {B}`

Prim: Step Through the Code (Adjacency Matrix)

```
1  void prim(int[][] G, int src) {
2    int V = G.length;
3    int[] key = new int[V];
4    int[] parent = new int[V];
5    boolean[] inTree = new boolean[V];
6    Arrays.fill(key, Integer.MAX_VALUE);
7    key[src] = 0;
8    parent[src] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = minKey(key, inTree);
11     inTree[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !inTree[v]
14           && G[u][v] < key[v]) {
▶          key[v] = G[u][v];
▶          parent[v] = u;
17       }
18   }
19 }
```



v = C: G[B][C] = 3 < key[C] = ∞ → `key[C] = 3`, `parent[C] = B`.

Step 4 / 15

VARIABLES

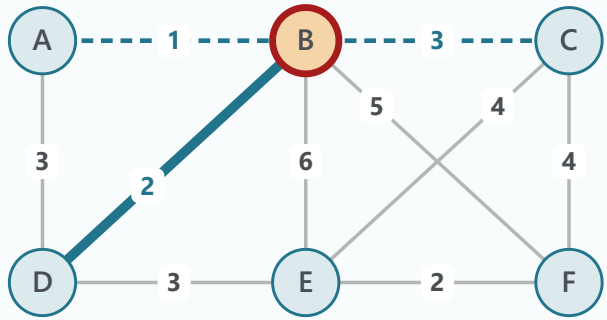
`s = 0` `u = B` `v = C`

`key = [1, 0, 3, ∞, ∞, ∞]`

`parent = [B, –, B, –, –, –]` `inTree = {B}`

Prim: Step Through the Code (Adjacency Matrix)

```
1  void prim(int[][] G, int src) {
2    int V = G.length;
3    int[] key = new int[V];
4    int[] parent = new int[V];
5    boolean[] inTree = new boolean[V];
6    Arrays.fill(key, Integer.MAX_VALUE);
7    key[src] = 0;
8    parent[src] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = minKey(key, inTree);
11     inTree[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !inTree[v]
14           && G[u][v] < key[v]) {
▶          key[v] = G[u][v];
▶          parent[v] = u;
17       }
18   }
19 }
```



v = D: G[B][D] = 2 < key[D] = ∞ → `key[D] = 2`, `parent[D] = B`.

Step 5 / 15

VARIABLES

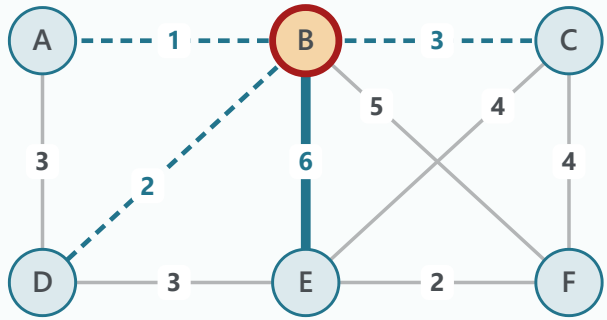
`s = 0` `u = B` `v = D`

`key = [1, 0, 3, 2, ∞, ∞]`

`parent = [B, -, B, B, -, -]` `inTree = {B}`

Prim: Step Through the Code (Adjacency Matrix)

```
1  void prim(int[][] G, int src) {
2    int V = G.length;
3    int[] key = new int[V];
4    int[] parent = new int[V];
5    boolean[] inTree = new boolean[V];
6    Arrays.fill(key, Integer.MAX_VALUE);
7    key[src] = 0;
8    parent[src] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = minKey(key, inTree);
11     inTree[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !inTree[v]
14           && G[u][v] < key[v]) {
▶          key[v] = G[u][v];
▶          parent[v] = u;
17       }
18   }
19 }
```



v = E: G[B][E] = 6 < key[E] = ∞ → `key[E] = 6`, `parent[E] = B`.

Step 6 / 15

VARIABLES

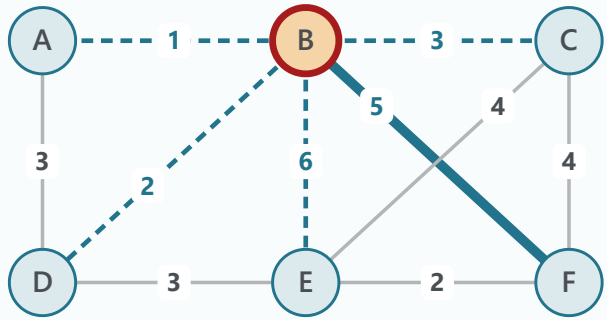
`s = 0` `u = B` `v = E`

`key = [1, 0, 3, 2, 6, ∞]`

`parent = [B, -, B, B, B, -]` `inTree = {B}`

Prim: Step Through the Code (Adjacency Matrix)

```
1  void prim(int[][] G, int src) {
2    int V = G.length;
3    int[] key = new int[V];
4    int[] parent = new int[V];
5    boolean[] inTree = new boolean[V];
6    Arrays.fill(key, Integer.MAX_VALUE);
7    key[src] = 0;
8    parent[src] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = minKey(key, inTree);
11     inTree[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !inTree[v]
14           && G[u][v] < key[v]) {
▶          key[v] = G[u][v];
▶          parent[v] = u;
17       }
18   }
19 }
```



v = F: G[B][F] = 5 < key[F] = ∞ → `key[F] = 5`, `parent[F] = B`.

Step 7 / 15

VARIABLES

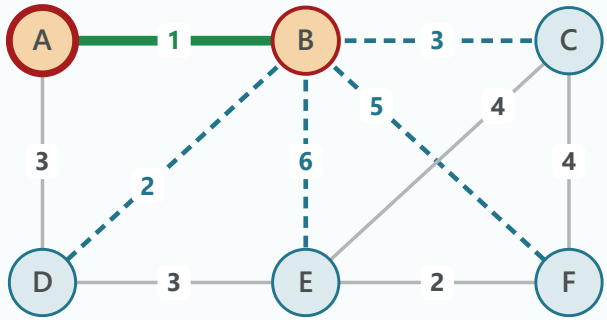
`s = 0` `u = B` `v = F`

`key = [1, 0, 3, 2, 6, 5]`

`parent = [B, -, B, B, B, B]` `inTree = {B}`

Prim: Step Through the Code (Adjacency Matrix)

```
1  void prim(int[][] G, int src) {
2    int V = G.length;
3    int[] key = new int[V];
4    int[] parent = new int[V];
5    boolean[] inTree = new boolean[V];
6    Arrays.fill(key, Integer.MAX_VALUE);
7    key[src] = 0;
8    parent[src] = -1;
9    for (int s = 0; s < V - 1; s++) {
▶      int u = minKey(key, inTree);
▶      inTree[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !inTree[v]
14           && G[u][v] < key[v]) {
15         key[v] = G[u][v];
16         parent[v] = u;
17       }
18   }
19 }
```



`minKey` → u = **A** (key 1); (B,A) enters the tree. w(T) = 1. AD = 3 > key[D] = 2: no update.

Step 8 / 15

VARIABLES

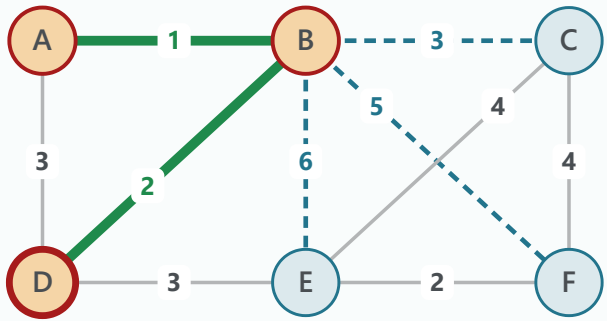
`s = 1` `u = A` `key = [1, 0, 3, 2, 6, 5]`

`parent = [B, –, B, B, B, B]`

`inTree = {B,A}`

Prim: Step Through the Code (Adjacency Matrix)

```
1  void prim(int[][] G, int src) {
2    int V = G.length;
3    int[] key = new int[V];
4    int[] parent = new int[V];
5    boolean[] inTree = new boolean[V];
6    Arrays.fill(key, Integer.MAX_VALUE);
7    key[src] = 0;
8    parent[src] = -1;
9    for (int s = 0; s < V - 1; s++) {
▶      int u = minKey(key, inTree);
▶      inTree[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !inTree[v]
14           && G[u][v] < key[v]) {
15         key[v] = G[u][v];
16         parent[v] = u;
17       }
18   }
19 }
```



`minKey` → u = **D** (key 2); (B,D) enters the tree. w(T) = 3.

Step 9 / 15

VARIABLES

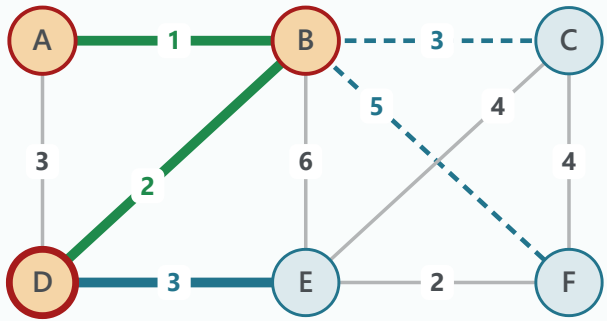
`s = 2` `u = D` `key = [1, 0, 3, 2, 6, 5]`

`parent = [B, –, B, B, B, B]`

`inTree = {B,A,D}`

Prim: Step Through the Code (Adjacency Matrix)

```
1  void prim(int[][] G, int src) {
2    int V = G.length;
3    int[] key = new int[V];
4    int[] parent = new int[V];
5    boolean[] inTree = new boolean[V];
6    Arrays.fill(key, Integer.MAX_VALUE);
7    key[src] = 0;
8    parent[src] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = minKey(key, inTree);
11     inTree[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !inTree[v]
14           && G[u][v] < key[v]) {
▶        key[v] = G[u][v];
▶        parent[v] = u;
17       }
18   }
19 }
```



v = E: G[D][E] = 3 < key[E] = 6 → `key[E] = 3`, `parent[E] = D`.

Step 10 / 15

VARIABLES

`s = 2` `u = D` `v = E`

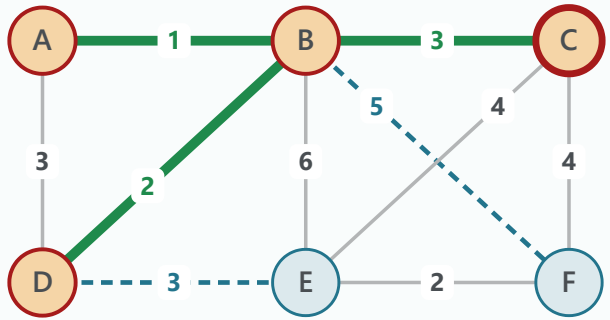
`key = [1, 0, 3, 2, 3, 5]`

`parent = [B, –, B, B, D, B]`

`inTree = {B,A,D}`

Prim: Step Through the Code (Adjacency Matrix)

```
1  void prim(int[][] G, int src) {
2    int V = G.length;
3    int[] key = new int[V];
4    int[] parent = new int[V];
5    boolean[] inTree = new boolean[V];
6    Arrays.fill(key, Integer.MAX_VALUE);
7    key[src] = 0;
8    parent[src] = -1;
9    for (int s = 0; s < V - 1; s++) {
▶      int u = minKey(key, inTree);
▶      inTree[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !inTree[v]
14           && G[u][v] < key[v]) {
15         key[v] = G[u][v];
16         parent[v] = u;
17       }
18   }
19 }
```



`minKey` → u = **C** (key 3); (B,C) enters the tree. w(T) = 6. C and E tie at 3: the lower index, C, wins.

Step 11 / 15

VARIABLES

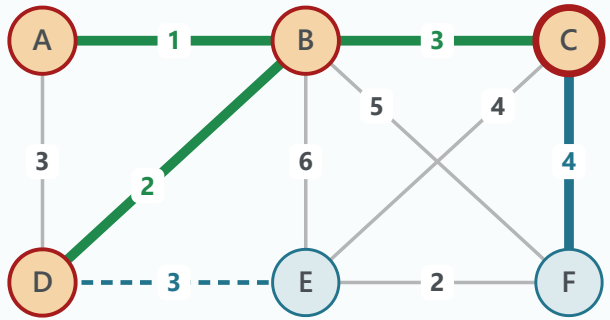
`s = 3` `u = C` `key = [1, 0, 3, 2, 3, 5]`

`parent = [B, –, B, B, D, B]`

`inTree = {B,A,D,C}`

Prim: Step Through the Code (Adjacency Matrix)

```
1  void prim(int[][] G, int src) {
2    int V = G.length;
3    int[] key = new int[V];
4    int[] parent = new int[V];
5    boolean[] inTree = new boolean[V];
6    Arrays.fill(key, Integer.MAX_VALUE);
7    key[src] = 0;
8    parent[src] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = minKey(key, inTree);
11     inTree[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !inTree[v]
14           && G[u][v] < key[v]) {
▶        key[v] = G[u][v];
▶        parent[v] = u;
17       }
18   }
19 }
```



v = F: G[C][F] = 4 < key[F] = 5 → `key[F] = 4`, `parent[F] = C`.

Step 12 / 15

VARIABLES

`s = 3` `u = C` `v = F`

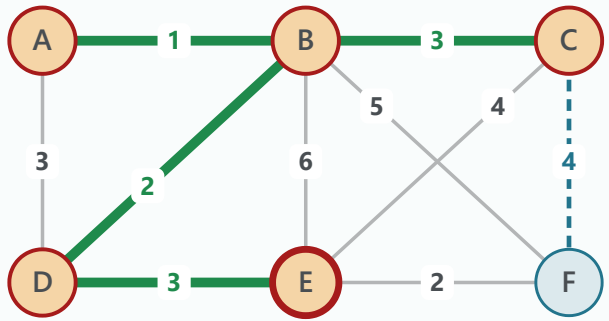
`key = [1, 0, 3, 2, 3, 4]`

`parent = [B, –, B, B, D, C]`

`inTree = {B,A,D,C}`

Prim: Step Through the Code (Adjacency Matrix)

```
1  void prim(int[][] G, int src) {
2    int V = G.length;
3    int[] key = new int[V];
4    int[] parent = new int[V];
5    boolean[] inTree = new boolean[V];
6    Arrays.fill(key, Integer.MAX_VALUE);
7    key[src] = 0;
8    parent[src] = -1;
9    for (int s = 0; s < V - 1; s++) {
▶      int u = minKey(key, inTree);
▶      inTree[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !inTree[v]
14           && G[u][v] < key[v]) {
15         key[v] = G[u][v];
16         parent[v] = u;
17       }
18   }
19 }
```



`minKey` → u = **E** (key 3); (D,E) enters the tree. w(T) = 9.

Step 13 / 15

VARIABLES

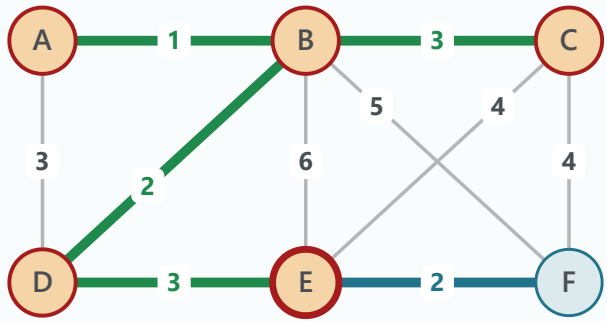
`s = 4` `u = E` `key = [1, 0, 3, 2, 3, 4]`

`parent = [B, -, B, B, D, C]`

`inTree = {B,A,D,C,E}`

Prim: Step Through the Code (Adjacency Matrix)

```
1  void prim(int[][] G, int src) {
2    int V = G.length;
3    int[] key = new int[V];
4    int[] parent = new int[V];
5    boolean[] inTree = new boolean[V];
6    Arrays.fill(key, Integer.MAX_VALUE);
7    key[src] = 0;
8    parent[src] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = minKey(key, inTree);
11     inTree[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !inTree[v]
14           && G[u][v] < key[v]) {
▶        key[v] = G[u][v];
▶        parent[v] = u;
17       }
18   }
19 }
```



v = F: G[E][F] = 2 < key[F] = 4 → `key[F] = 2`, `parent[F] = E`.

Step 14 / 15

VARIABLES

`s = 4` `u = E` `v = F`

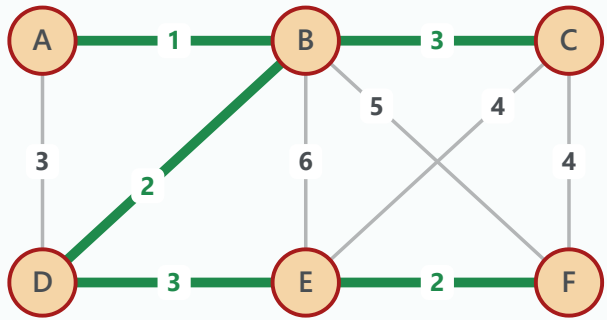
`key = [1, 0, 3, 2, 3, 2]`

`parent = [B, –, B, B, D, E]`

`inTree = {B,A,D,C,E}`

Prim: Step Through the Code (Adjacency Matrix)

```
1  void prim(int[][] G, int src) {
2    int V = G.length;
3    int[] key = new int[V];
4    int[] parent = new int[V];
5    boolean[] inTree = new boolean[V];
6    Arrays.fill(key, Integer.MAX_VALUE);
7    key[src] = 0;
8    parent[src] = -1;
▶    for (int s = 0; s < V - 1; s++) {
10     int u = minKey(key, inTree);
11     inTree[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !inTree[v]
14           && G[u][v] < key[v]) {
15         key[v] = G[u][v];
16         parent[v] = u;
17       }
18   }
19 }
```



The loop ran V − 1 = 5 times; the last vertex, F, has parent E. MST = (parent[v], v): AB, BC, BD, DE, EF → **w(T) = 11**.

Step 15 / 15

VARIABLES

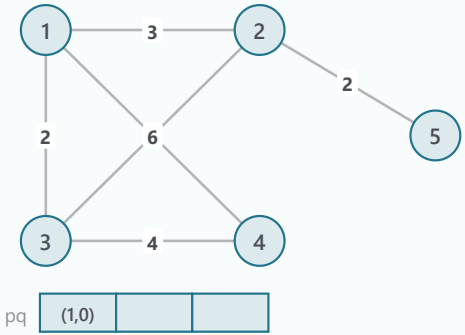
`key = [1, 0, 3, 2, 3, 2]`

`parent = [B, −, B, B, D, E]`

`w(T) = 11`

Prim: Tracing the Lazy Priority-Queue Version

```
1  void primPQ(int src) {
▶    Arrays.fill(key, INF);
▶    key[src] = 0;
▶    PriorityQueue<int[]> pq =
▶        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
▶    pq.add(new int[] { src, 0 });
7    while (!pq.isEmpty()) {
8      int u = pq.poll()[0];
9      if (inTree[u]) continue;  // stale
10     inTree[u] = true;
11     for (Edge e : adj.get(u))
12       if (!inTree[e.v] && e.w < key[e.v]) {
13         key[e.v] = e.w; parent[e.v] = u;
14         pq.add(new int[] { e.v, e.w });
15       }
16   }
17 }
```



Undirected graph with five vertices, source 1. key[1] = 0, the others ∞; the queue holds a single (vertex, key) entry, (1,0). In the parent array (parent[]), − means not yet assigned.

Step 1 / 8

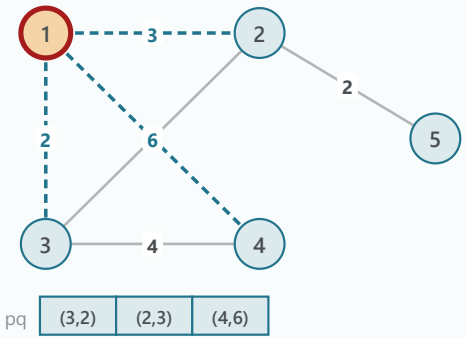
VARIABLES

`u = −` `key = [0, ∞, ∞, ∞, ∞]`

`parent = [−, −, −, −, −]`

Prim: Tracing the Lazy Priority-Queue Version

```
1  void primPQ(int src) {
2    Arrays.fill(key, INF);
3    key[src] = 0;
4    PriorityQueue<int[]> pq =
5        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
6    pq.add(new int[] { src, 0 });
7    while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
▶      if (inTree[u]) continue;  // stale
▶      inTree[u] = true;
▶      for (Edge e : adj.get(u))
▶        if (!inTree[e.v] && e.w < key[e.v]) {
▶          key[e.v] = e.w; parent[e.v] = u;
▶          pq.add(new int[] { e.v, e.w });
15       }
16   }
17 }
```



(1,0) is the smallest entry: 1 enters the tree. key[2]: ∞ → 3, key[3]: ∞ → 2, key[4]: ∞ → 6; the new entries are added to the queue.

Step 2 / 8

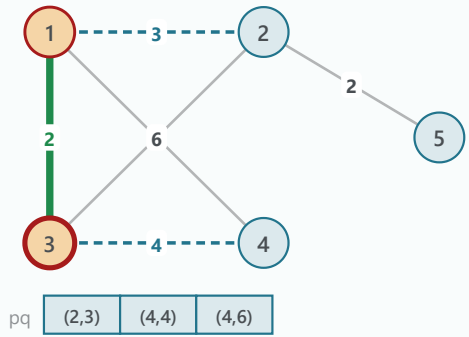
VARIABLES

`u = 1` `key = [0, 3, 2, 6, ∞]`

`parent = [-, 1, 1, 1, -]`

Prim: Tracing the Lazy Priority-Queue Version

```
1  void primPQ(int src) {
2    Arrays.fill(key, INF);
3    key[src] = 0;
4    PriorityQueue<int[]> pq =
5        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
6    pq.add(new int[] { src, 0 });
7    while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
▶      if (inTree[u]) continue;  // stale
▶      inTree[u] = true;
▶      for (Edge e : adj.get(u))
▶        if (!inTree[e.v] && e.w < key[e.v]) {
▶          key[e.v] = e.w; parent[e.v] = u;
▶          pq.add(new int[] { e.v, e.w });
15       }
16   }
17 }
```



(3,2) is the smallest entry: 3 enters the tree (edge 1–3, weight 2). key[4]: 6 → 4; the new entries are added to the queue. The edge of weight 4 to 2 costs more than the current 3: no update. The entry (4,6) stays in the queue.

Step 3 / 8

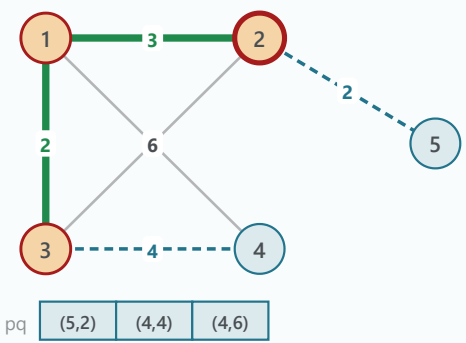
VARIABLES

`u = 3` `key = [0, 3, 2, 4, ∞]`

`parent = [–, 1, 1, 3, –]`

Prim: Tracing the Lazy Priority-Queue Version

```
1  void primPQ(int src) {
2    Arrays.fill(key, INF);
3    key[src] = 0;
4    PriorityQueue<int[]> pq =
5        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
6    pq.add(new int[] { src, 0 });
7    while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
▶      if (inTree[u]) continue;  // stale
▶      inTree[u] = true;
▶      for (Edge e : adj.get(u))
▶        if (!inTree[e.v] && e.w < key[e.v]) {
▶          key[e.v] = e.w; parent[e.v] = u;
▶          pq.add(new int[] { e.v, e.w });
15       }
16   }
17 }
```



(2,3) is the smallest entry: 2 enters the tree (edge 1–2, weight 3). key[5]: ∞ → 2; the new entries are added to the queue.

Step 4 / 8

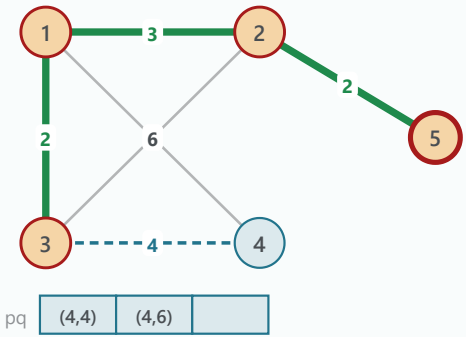
VARIABLES

`u = 2` `key = [0, 3, 2, 4, 2]`

`parent = [-, 1, 1, 3, 2]`

Prim: Tracing the Lazy Priority-Queue Version

```
1  void primPQ(int src) {
2    Arrays.fill(key, INF);
3    key[src] = 0;
4    PriorityQueue<int[]> pq =
5        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
6    pq.add(new int[] { src, 0 });
7    while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
▶      if (inTree[u]) continue;  // stale
▶      inTree[u] = true;
▶      for (Edge e : adj.get(u))
▶        if (!inTree[e.v] && e.w < key[e.v]) {
▶          key[e.v] = e.w; parent[e.v] = u;
▶          pq.add(new int[] { e.v, e.w });
15       }
16   }
17 }
```



(5,2) is the smallest entry: 5 enters the tree (edge 2–5, weight 2). No key of a non-tree neighbor decreased.

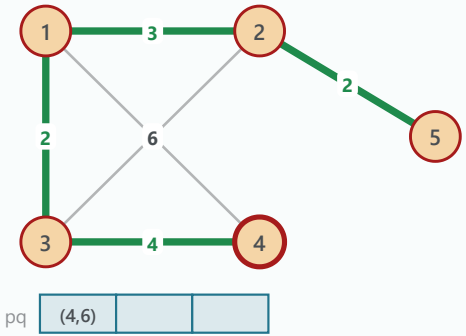
Step 5 / 8

VARIABLES

`u = 5` `key = [0, 3, 2, 4, 2]` `parent = [-, 1, 1, 3, 2]`

Prim: Tracing the Lazy Priority-Queue Version

```
1  void primPQ(int src) {
2    Arrays.fill(key, INF);
3    key[src] = 0;
4    PriorityQueue<int[]> pq =
5        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
6    pq.add(new int[] { src, 0 });
7    while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
▶      if (inTree[u]) continue;  // stale
▶      inTree[u] = true;
▶      for (Edge e : adj.get(u))
▶        if (!inTree[e.v] && e.w < key[e.v]) {
▶          key[e.v] = e.w; parent[e.v] = u;
▶          pq.add(new int[] { e.v, e.w });
15       }
16   }
17 }
```



(4,4) is the smallest entry: 4 enters the tree (edge 3–4, weight 4). No key of a non-tree neighbor decreased.

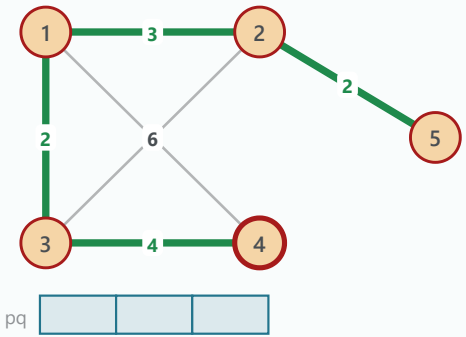
Step 6 / 8

VARIABLES

`u = 4` `key = [0, 3, 2, 4, 2]` `parent = [-, 1, 1, 3, 2]`

Prim: Tracing the Lazy Priority-Queue Version

```
1  void primPQ(int src) {
2    Arrays.fill(key, INF);
3    key[src] = 0;
4    PriorityQueue<int[]> pq =
5        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
6    pq.add(new int[] { src, 0 });
7    while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
▶      if (inTree[u]) continue;  // stale
10     inTree[u] = true;
11     for (Edge e : adj.get(u))
12       if (!inTree[e.v] && e.w < key[e.v]) {
13         key[e.v] = e.w; parent[e.v] = u;
14         pq.add(new int[] { e.v, e.w });
15       }
16   }
17 }
```



(4,6) comes out, but 4 is already in the tree: it is a **stale entry**, skipped with `continue` (lazy deletion).

Step 7 / 8

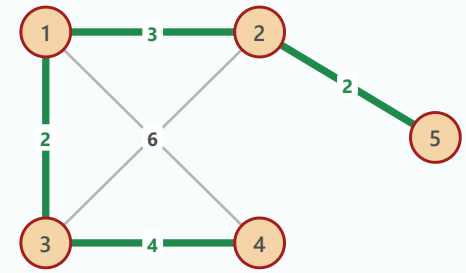
VARIABLES

`u = 4` `key = [0, 3, 2, 4, 2]`

`parent = [-, 1, 1, 3, 2]`

Prim: Tracing the Lazy Priority-Queue Version

```
1  void primPQ(int src) {
2    Arrays.fill(key, INF);
3    key[src] = 0;
4    PriorityQueue<int[]> pq =
5        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
6    pq.add(new int[] { src, 0 });
▶    while (!pq.isEmpty()) {
8      int u = pq.poll()[0];
9      if (inTree[u]) continue;  // stale
10     inTree[u] = true;
11     for (Edge e : adj.get(u))
12       if (!inTree[e.v] && e.w < key[e.v]) {
13         key[e.v] = e.w; parent[e.v] = u;
14         pq.add(new int[] { e.v, e.w });
15       }
16   }
17 }
```



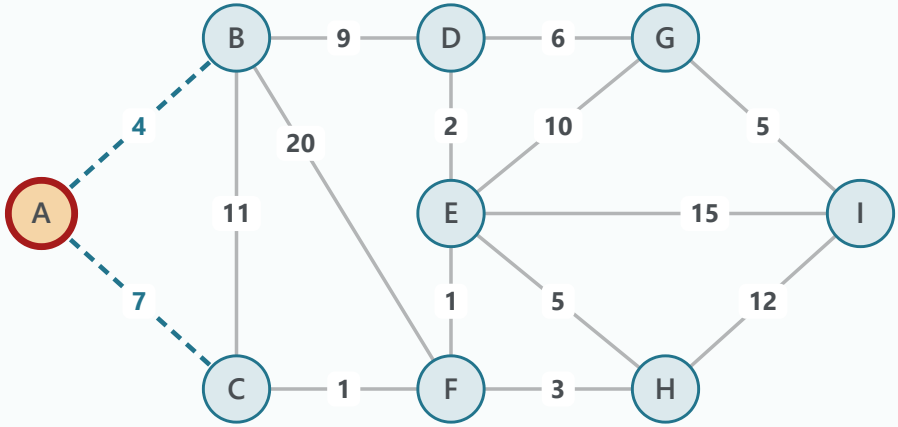
The queue is empty. MST edges (parent[v], v): 1–3 (2), 1–2 (3), 2–5 (2), 3–4 (4); total **11**.

Step 8 / 8

VARIABLES

`u = –` `key = [0, 3, 2, 4, 2]` `parent = [–, 1, 1, 3, 2]`

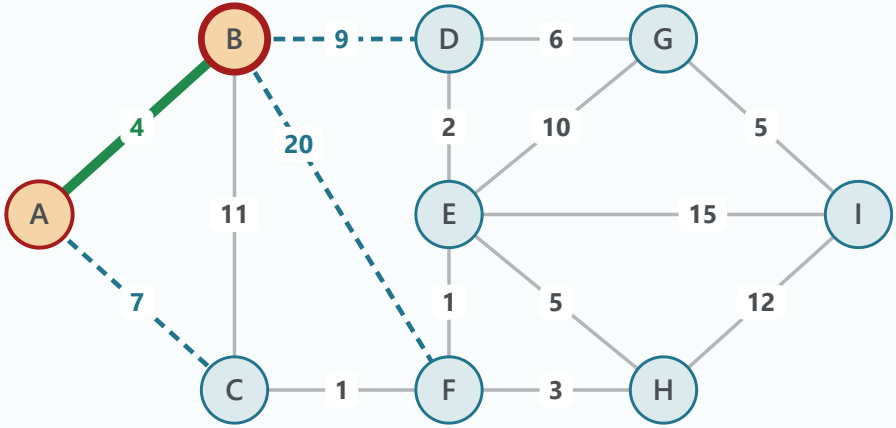
Prim Example: Nine Vertices (Starting at A)



	A	B	C	D	E	F	G	H	I
key	0	4	7	∞	∞	∞	∞	∞	∞
parent	–	A	A	–	–	–	–	–	–

Start at **A**: its neighbors B (4) and C (7) become candidates.

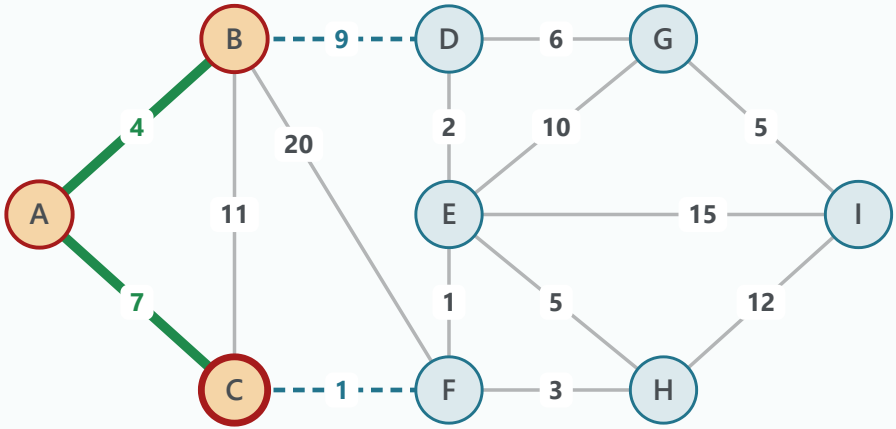
Prim Example: Nine Vertices (Starting at A)



	A	B	C	D	E	F	G	H	I
key	0	4	7	9	∞	20	∞	∞	∞
parent	–	A	A	B	–	B	–	–	–

B enters the tree: (A,B) = 4, total 4. Updated: D → 9, F → 20.

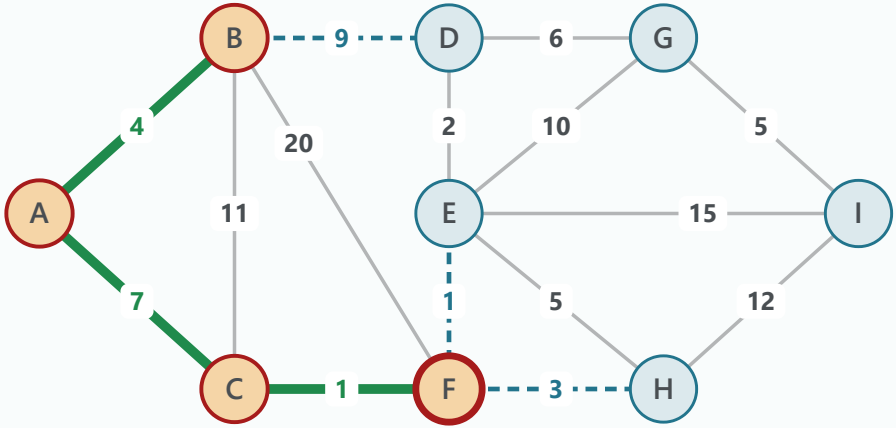
Prim Example: Nine Vertices (Starting at A)



	A	B	C	D	E	F	G	H	I
key	0	4	7	9	∞	1	∞	∞	∞
parent	–	A	A	B	–	C	–	–	–

C enters the tree: (A,C) = 7, total 11. Updated: F → 1.

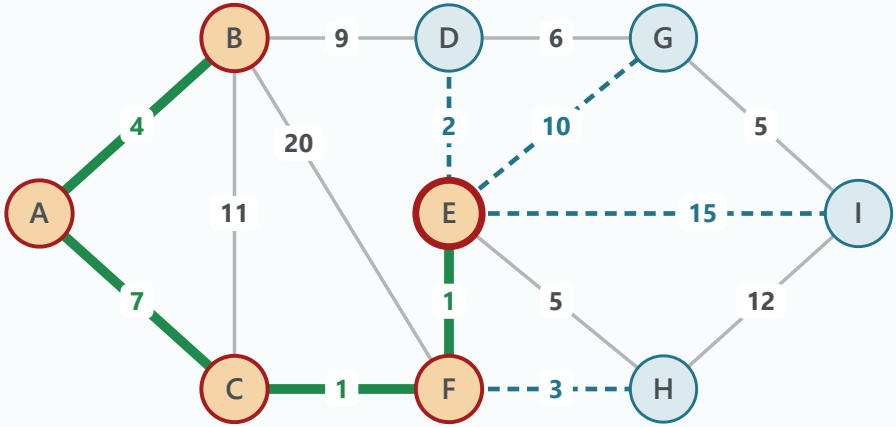
Prim Example: Nine Vertices (Starting at A)



	A	B	C	D	E	F	G	H	I
key	0	4	7	9	1	1	∞	3	∞
parent	–	A	A	B	F	C	–	F	–

F enters the tree: (C,F) = 1, total 12. Updated: E → 1, H → 3.

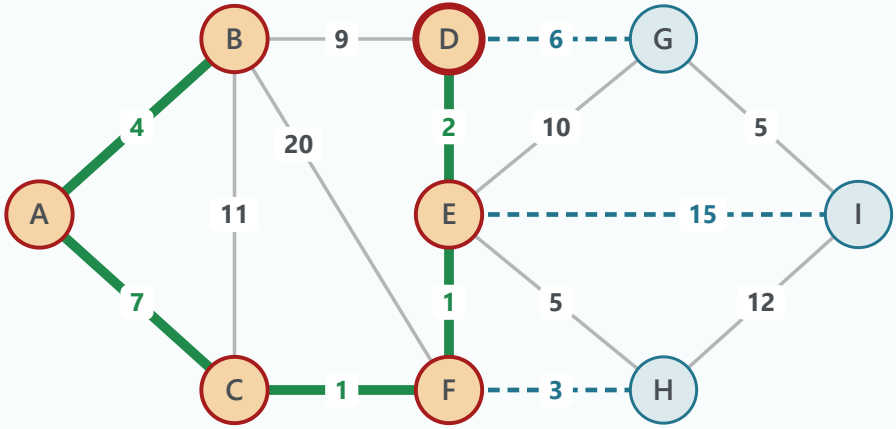
Prim Example: Nine Vertices (Starting at A)



	A	B	C	D	E	F	G	H	I
key	0	4	7	2	1	1	10	3	15
parent	–	A	A	E	F	C	E	F	E

E enters the tree: (F,E) = 1, total 13. Updated: D → 2, G → 10, I → 15.

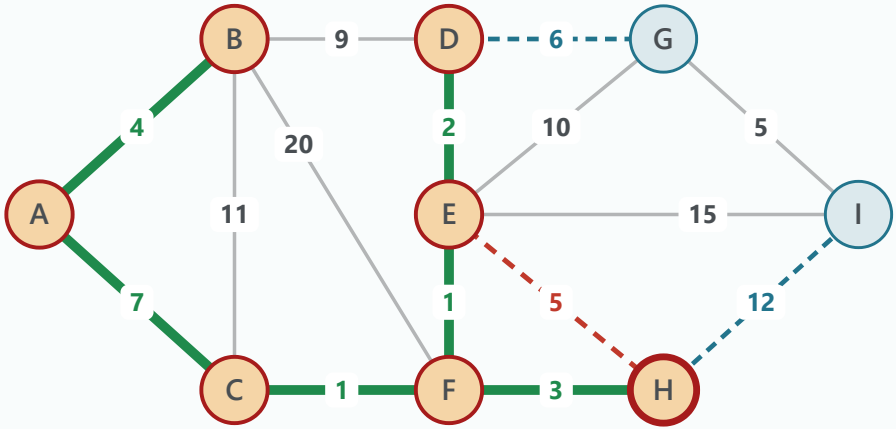
Prim Example: Nine Vertices (Starting at A)



	A	B	C	D	E	F	G	H	I
key	0	4	7	2	1	1	6	3	15
parent	–	A	A	E	F	C	D	F	E

D enters the tree: (E,D) = 2, total 15. Updated: G → 6.

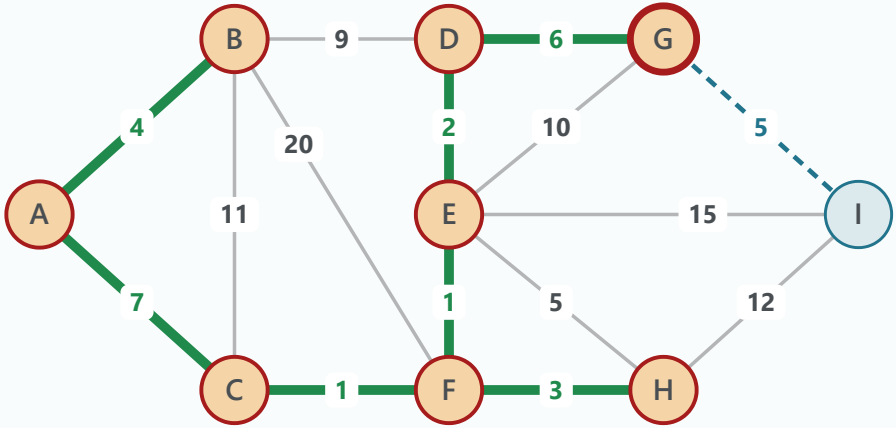
Prim Example: Nine Vertices (Starting at A)



	A	B	C	D	E	F	G	H	I
key	0	4	7	2	1	1	6	3	12
parent	–	A	A	E	F	C	D	F	H

H enters the tree: (F,H) = 3, total 18. Updated: I → 12. EH = 5 can no longer be used: E and H are both in the tree (a cycle).

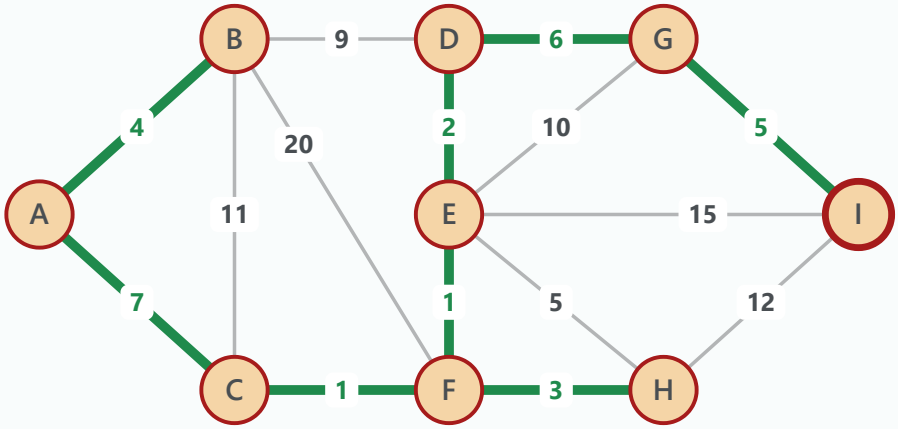
Prim Example: Nine Vertices (Starting at A)



	A	B	C	D	E	F	G	H	I
key	0	4	7	2	1	1	6	3	5
parent	–	A	A	E	F	C	D	F	G

G enters the tree: (D,G) = 6, total 24. Updated: I → 5.

Prim Example: Nine Vertices (Starting at A)



	A	B	C	D	E	F	G	H	I
key	0	4	7	2	1	1	6	3	5
parent	–	A	A	E	F	C	D	F	G

I enters the tree: (G,I) = 5, total 29. Every vertex is in the tree: **MST = 29** (same as Kruskal).

Prim: Complexity Analysis

Implementation	extract-min	decrease-key	Total
Adjacency matrix + array scan (the code above)	V × O(V)	O(1)	**$O(V^2)$**
Adjacency list + binary heap	V × O(log V)	E × O(log V)	**O((V + E) log V)**
Lazy version (pseudocode): edge copies in the queue	≤ E inserts/removals, each O(log E)		O(E log E)
Fibonacci heap	O(log V) amortized	O(1) amortized	O(E + V log V)

Dense graphs ($E \approx V^2$)

The matrix version, $O(V^2)$, is best; the heap version becomes $O(V^2 \log V)$.

Sparse graphs ($E \approx V$)

The heap version is O(V log V); the matrix version needlessly costs $O(V^2)$.

How Prim Differs from Dijkstra

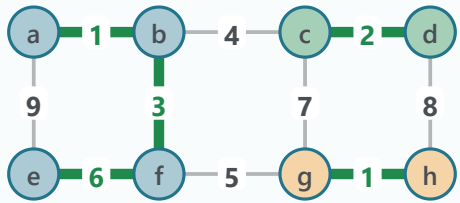
The two algorithms have almost the same code: start at a source, extract the vertex with the smallest key from a priority queue, and relax its neighbors. The only difference is **what the key means**.

	Prim (MST)	Dijkstra (shortest paths)
key[v]	the cheapest **single edge** joining v to the *tree*	length of the **whole path** from the source to v
Update	`w(u,v) < key[v]`	`dist[u] + w(u,v) < dist[v]`
Result	tree of minimum total weight	shortest-path tree (SPT)
Source	does not affect the result (same weight)	determines the result
Negative weights	no problem	fails

In the five-vertex example, the shortest-path tree from 1 can use the edge (1,4) = 6; the MST takes (3,4) = 4 instead.

Borůvka's (Sollin's) Algorithm

- The **oldest** MST algorithm: Otakar **Borůvka**, 1926 (the Moravian power grid). Also known as **Sollin's** algorithm.
- At first every vertex is an **independent component**.
- In each round **every component** picks the **cheapest edge** leaving it; all the picks are added at once.
- Components are merged with **union-find** until only one is left. On a connected graph it yields $|V| - 1$ edges.



round 1: each vertex picks its own cheapest edge
→ 8 components shrink to 3

Strengths

+ Components choose independently: well suited to **parallel** execution.
+ At most $\lceil \log_2 V \rceil$ rounds.

Weaknesses

– Harder to code than Kruskal or Prim.
– With equal weights a **consistent tie-breaker** (e.g. edge index) is essential; otherwise a cycle can form.

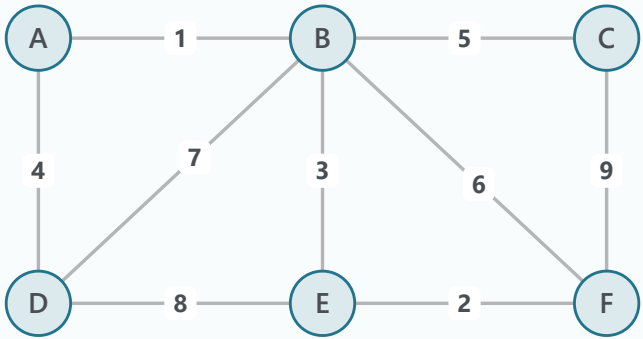
Borůvka: Steps and Pseudocode

1. At first each vertex counts as a component (set) on its own.
2. For each component, find the lightest edge connecting it to **another component**.
3. Add all the edges found and merge the components they connect (no edge is added twice).
4. Repeat steps 2 and 3 until one component remains; stop if a round adds no edge (disconnected graph → forest).

Note: Borůvka does **not sort** the edges; each round scans all edges once and keeps the lightest per component.

```
BORUVKA(G):
  MST = [ ]
  numComp = |V|
  while numComp > 1 do:
    cheapest = [ ]  // per component
    for each e = (u, v, weight) in E:
      cu = dsu.find(u)
      cv = dsu.find(v)
      if cu ≠ cv:
        cheapest[cu] = min(cheapest[cu], e)
        cheapest[cv] = min(cheapest[cv], e)
    added = false
    for each (u, v, weight) in cheapest:
      if dsu.find(u) ≠ dsu.find(v):
        MST.add((u, v, weight))
        dsu.union(u, v)
        numComp -= 1
        added = true
    if not added: break   // forest
  return MST
```

Borůvka Step by Step: Six Vertices

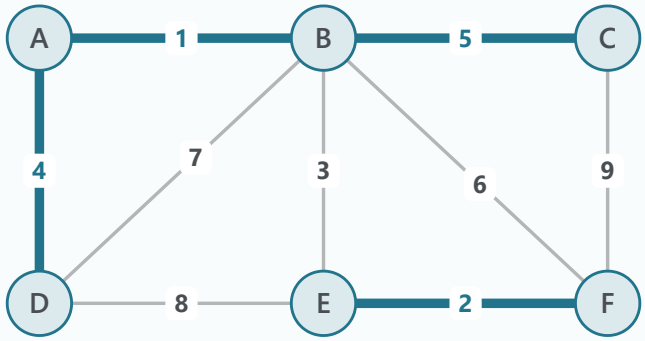


Start: each vertex is a separate component (6 components).

Step 1 / 5

Kruskal adds the edges 1, 2, 3, 4, 5 one at a time; Borůvka adds four edges **at once** in round 1. The resulting tree is the same.

Borůvka Step by Step: Six Vertices

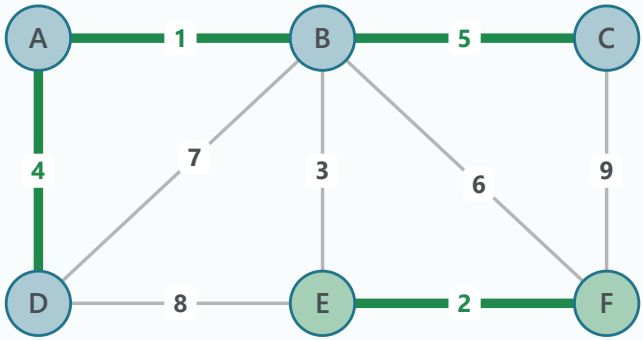


Component	Cheapest
{A}	AB 1
{B}	AB 1
{C}	BC 5
{D}	AD 4
{E}	EF 2
{F}	EF 2

Round 1: each component picks its own cheapest outgoing edge (blue). AB and EF are each picked by two components but added once.

Kruskal adds the edges 1, 2, 3, 4, 5 one at a time; Borůvka adds four edges **at once** in round 1. The resulting tree is the same.

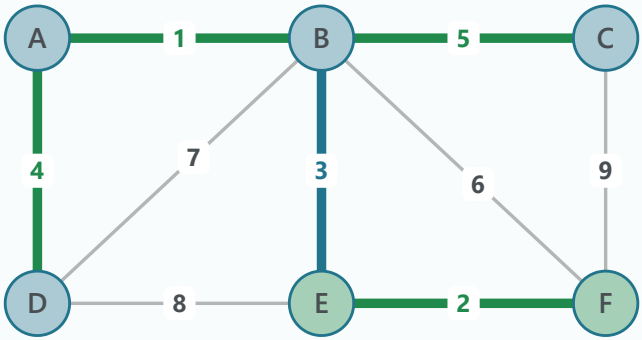
Borůvka Step by Step: Six Vertices



End of round 1: 4 distinct edges added (AB, BC, AD, EF), 2 components left: {A, B, C, D} {E, F}. Total 12.

Kruskal adds the edges 1, 2, 3, 4, 5 one at a time; Borůvka adds four edges **at once** in round 1. The resulting tree is the same.

Borůvka Step by Step: Six Vertices



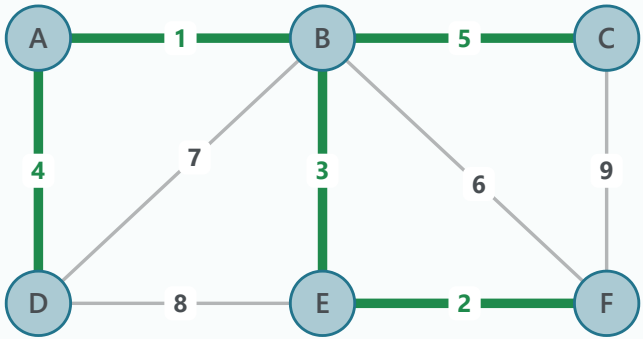
Component	Cheapest
{A,B,C,D}	BE 3
{E,F}	BE 3

Round 2: each component picks its own cheapest outgoing edge (blue). Both components pick the same edge.

Step 4 / 5

Kruskal adds the edges 1, 2, 3, 4, 5 one at a time; Borůvka adds four edges **at once** in round 1. The resulting tree is the same.

Borůvka Step by Step: Six Vertices

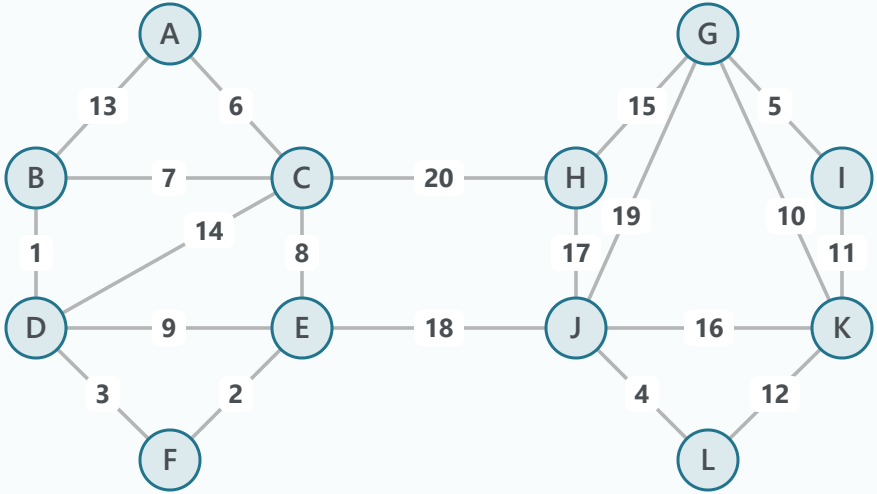


End of round 2: 1 distinct edge added (BE), 1 component left: {A, B, C, D, E, F}. Total 15.
MST = 15 (AB, BC, AD, EF, BE).

Step 5 / 5

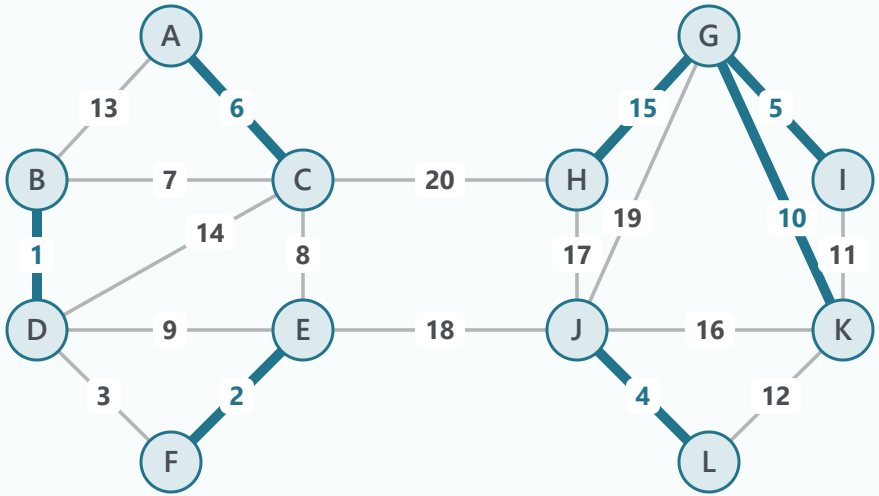
Kruskal adds the edges 1, 2, 3, 4, 5 one at a time; Borůvka adds four edges **at once** in round 1. The resulting tree is the same.

Borůvka Example: Twelve Vertices



Start: each vertex is a separate component (12 components).

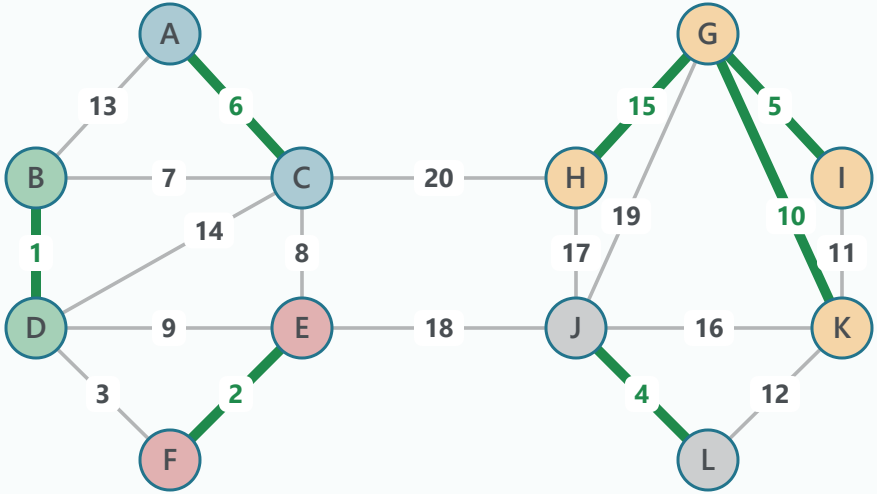
Borůvka Example: Twelve Vertices



Component	Cheapest
{A}	AC 6
{B}	BD 1
{C}	AC 6
{D}	BD 1
{E}	EF 2
{F}	EF 2
{G}	GI 5
{H}	GH 15
{I}	GI 5
{J}	JL 4
{K}	GK 10
{L}	JL 4

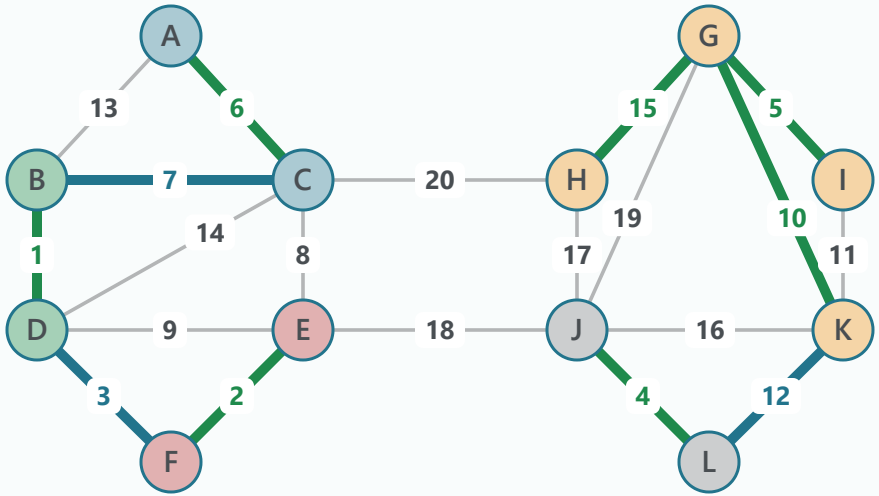
Round 1: the cheapest outgoing edge of each component (blue).

Borůvka Example: Twelve Vertices



End of round 1: 7 distinct edges added (AC, BD, EF, GI, GH, JL, GK), 5 components left: {G, H, I, K} {A, C} {B, D} {E, F} {J, L}. Total 43.

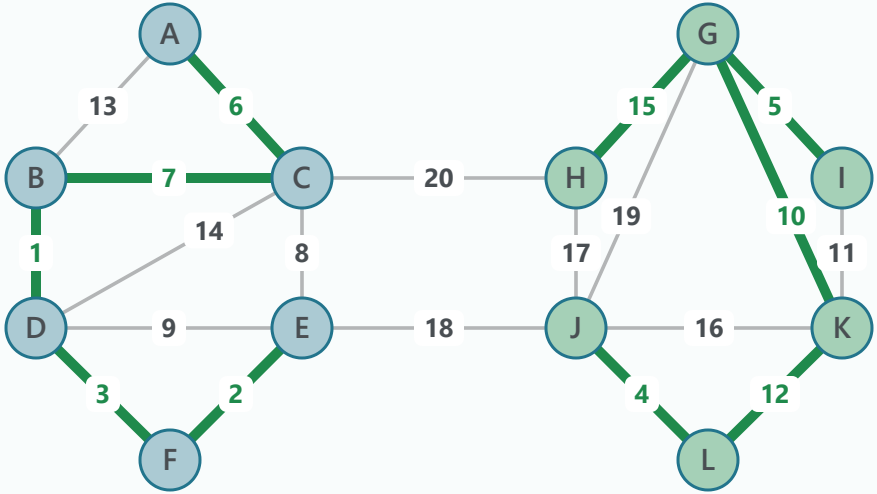
Borůvka Example: Twelve Vertices



Component	Cheapest
{A,C}	BC 7
{B,D}	DF 3
{E,F}	DF 3
{G,H,I,K}	KL 12
{J,L}	KL 12

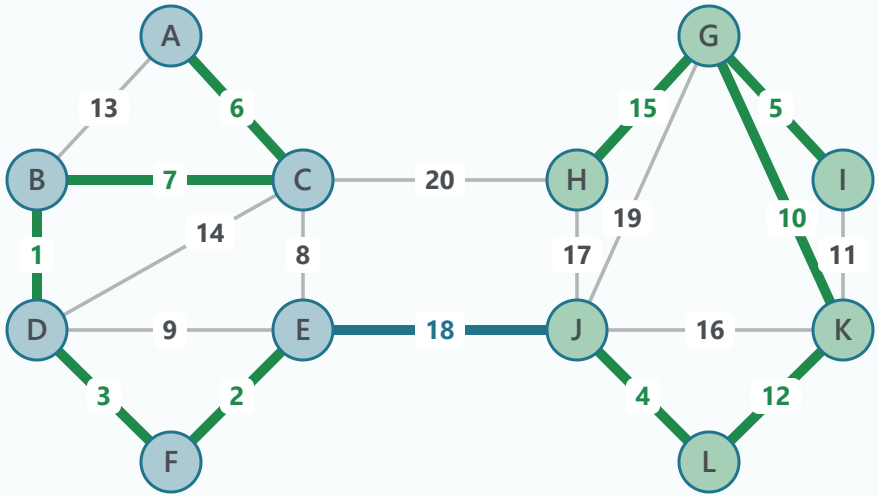
Round 2: the cheapest outgoing edge of each component (blue).

Borůvka Example: Twelve Vertices



End of round 2: 3 distinct edges added (BC, DF, KL), 2 components left: {A, B, C, D, E, F} {G, H, I, J, K, L}. Total 65.

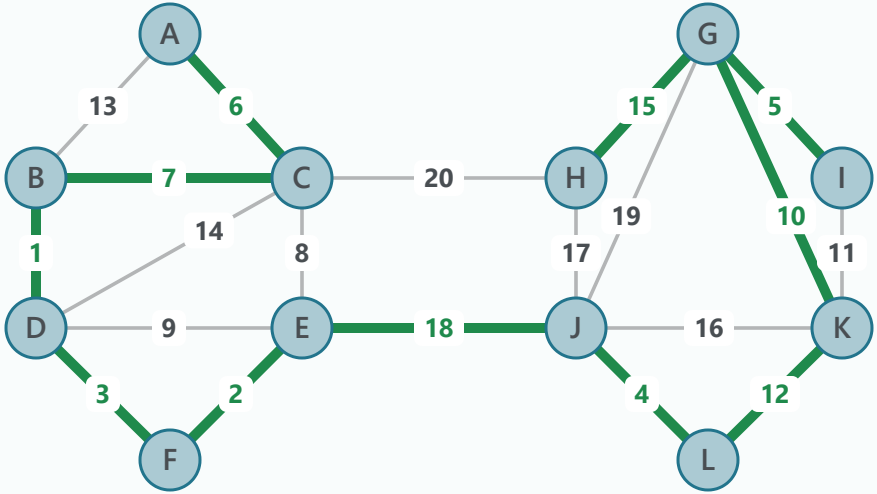
Borůvka Example: Twelve Vertices



Component	Cheapest
{A,B,C,D,E,F}	EJ 18
{G,H,I,J,K,L}	EJ 18

Round 3: the cheapest outgoing edge of each component (blue).

Borůvka Example: Twelve Vertices



End of round 3: 1 distinct edge added (EJ), 1 component left: {A, B, C, D, E, F, G, H, I, J, K, L}.

MST = 83, in just 3 rounds.

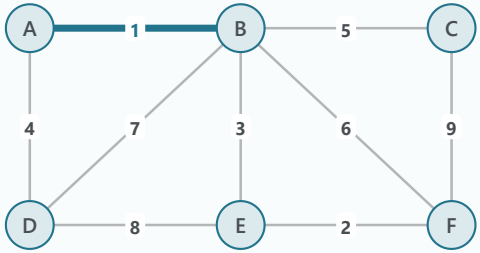
Borůvka: Java Code and Complexity

```
while (numComp > 1) {
  Edge[] cheapest = new Edge[V];
  for (Edge e : edges) {          // 1) cheapest
    int k1 = find(parent, e.u), k2 = find(parent, e.v);
    if (k1 == k2) continue;          // same comp.
    if (cheapest[k1] == null || e.w < cheapest[k1].w)
      cheapest[k1] = e;
    if (cheapest[k2] == null || e.w < cheapest[k2].w)
      cheapest[k2] = e;
  }
  boolean added = false;
  for (Edge e : cheapest) {       // 2) add them
    if (e == null) continue;
    int k1 = find(parent, e.u), k2 = find(parent, e.v);
    if (k1 == k2) continue;          // already in
    tree.add(e);
    parent[k1] = k2;                 // union
    numComp--;
    added = true;
  }
  if (!added) break;  // forest
}
```

- In each round every component merges with **at least one** other → the number of components at least **halves**.
- Number of rounds ≤ $\lceil \log_2 V \rceil$; each round scans all edges once: O(E).
- Total: **O(E log V)**. No sorting is done; no "weight order" is needed.
- If edges are added at the same time, a **consistent tie-breaker** is needed; in this sequential code the `find(k1) == find(k2)` check already prevents cycles.
- If a round adds no edge (`added` false), the loop exits: a minimum spanning **forest** for a disconnected graph.

Borůvka: Step Through the Code

```
1  while (numComp > 1) {
2    Edge[] cheapest = new Edge[V];
3    for (Edge e : edges) {      // 1) cheapest
▶      int k1 = find(parent, e.u), k2 = find(parent,
e.v);
▶      if (k1 == k2) continue;
▶      if (cheapest[k1] == null || e.w <
cheapest[k1].w)
▶        cheapest[k1] = e;
▶      if (cheapest[k2] == null || e.w <
cheapest[k2].w)
▶        cheapest[k2] = e;
10   }
11   boolean added = false;
12   for (Edge e : cheapest) {   // 2) add them
13     if (e == null) continue;
14     int k1 = find(parent, e.u), k2 = find(parent,
e.v);
15     if (k1 == k2) continue;
16     tree.add(e);
17     parent[k1] = k2;          // union
18     numComp--;
19     added = true;
20   }
```



Six components (each vertex alone); edges are scanned in list order. Edge AB(1): cheapest[A] and cheapest[B] = AB(1).

Step 1 / 14

VARIABLES

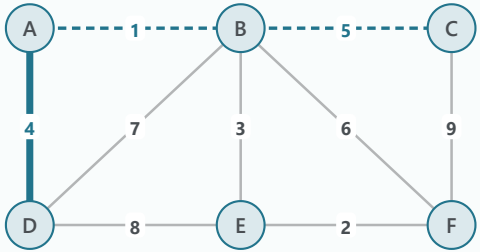
`e = AB(1)` `parent = [A, B, C, D, E, F]`

`cheapest = [AB, AB, –, –, –, –]`

`numComp = 6`

Borůvka: Step Through the Code

```
1  while (numComp > 1) {
2    Edge[] cheapest = new Edge[V];
3    for (Edge e : edges) {      // 1) cheapest
▶      int k1 = find(parent, e.u), k2 = find(parent,
e.v);
▶      if (k1 == k2) continue;
▶      if (cheapest[k1] == null || e.w <
cheapest[k1].w)
▶        cheapest[k1] = e;
▶      if (cheapest[k2] == null || e.w <
cheapest[k2].w)
▶        cheapest[k2] = e;
10   }
11   boolean added = false;
12   for (Edge e : cheapest) {   // 2) add them
13     if (e == null) continue;
14     int k1 = find(parent, e.u), k2 = find(parent,
e.v);
15     if (k1 == k2) continue;
16     tree.add(e);
17     parent[k1] = k2;          // union
18     numComp--;
19     added = true;
20   }
```



Edge BC(5): cheapest[C] = BC(5). Edge AD(4): cheapest[D] = AD(4).

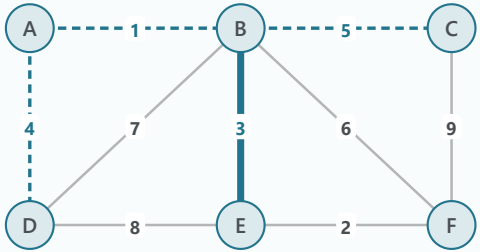
Step 2 / 14

VARIABLES

`e = AD(4)` `parent = [A, B, C, D, E, F]` `cheapest = [AB, AB, BC, AD, –, –]` `numComp = 6`

Borůvka: Step Through the Code

```
1  while (numComp > 1) {
2    Edge[] cheapest = new Edge[V];
3    for (Edge e : edges) {      // 1) cheapest
▶      int k1 = find(parent, e.u), k2 = find(parent,
e.v);
▶      if (k1 == k2) continue;
▶      if (cheapest[k1] == null || e.w <
cheapest[k1].w)
▶        cheapest[k1] = e;
▶      if (cheapest[k2] == null || e.w <
cheapest[k2].w)
▶        cheapest[k2] = e;
10   }
11   boolean added = false;
12   for (Edge e : cheapest) {   // 2) add them
13     if (e == null) continue;
14     int k1 = find(parent, e.u), k2 = find(parent,
e.v);
15     if (k1 == k2) continue;
16     tree.add(e);
17     parent[k1] = k2;          // union
18     numComp--;
19     added = true;
20   }
```



Edge BE(3): cheapest[E] = BE(3).

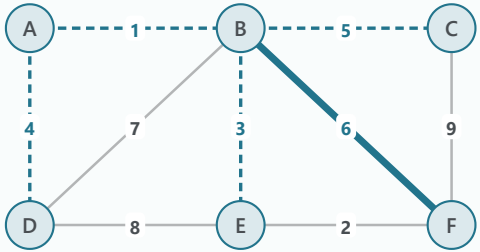
Step 3 / 14

VARIABLES

`e = BE(3)` `parent = [A, B, C, D, E, F]` `cheapest = [AB, AB, BC, AD, BE, –]` `numComp = 6`

Borůvka: Step Through the Code

```
1  while (numComp > 1) {
2    Edge[] cheapest = new Edge[V];
3    for (Edge e : edges) {      // 1) cheapest
▶      int k1 = find(parent, e.u), k2 = find(parent,
e.v);
▶      if (k1 == k2) continue;
▶      if (cheapest[k1] == null || e.w <
cheapest[k1].w)
▶        cheapest[k1] = e;
▶      if (cheapest[k2] == null || e.w <
cheapest[k2].w)
▶        cheapest[k2] = e;
10   }
11   boolean added = false;
12   for (Edge e : cheapest) {   // 2) add them
13     if (e == null) continue;
14     int k1 = find(parent, e.u), k2 = find(parent,
e.v);
15     if (k1 == k2) continue;
16     tree.add(e);
17     parent[k1] = k2;          // union
18     numComp--;
19     added = true;
20   }
```



BD(7) beat no component's cheapest. Edge BF(6): cheapest[F] = BF(6).

Step 4 / 14

VARIABLES

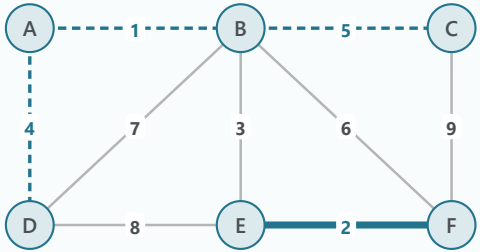
`e = BF(6)` `parent = [A, B, C, D, E, F]`

`cheapest = [AB, AB, BC, AD, BE, BF]`

`numComp = 6`

Borůvka: Step Through the Code

```
1  while (numComp > 1) {
2    Edge[] cheapest = new Edge[V];
3    for (Edge e : edges) {      // 1) cheapest
▶      int k1 = find(parent, e.u), k2 = find(parent,
e.v);
▶      if (k1 == k2) continue;
▶      if (cheapest[k1] == null || e.w <
cheapest[k1].w)
▶        cheapest[k1] = e;
▶      if (cheapest[k2] == null || e.w <
cheapest[k2].w)
▶        cheapest[k2] = e;
10   }
11   boolean added = false;
12   for (Edge e : cheapest) {   // 2) add them
13     if (e == null) continue;
14     int k1 = find(parent, e.u), k2 = find(parent,
e.v);
15     if (k1 == k2) continue;
16     tree.add(e);
17     parent[k1] = k2;          // union
18     numComp--;
19     added = true;
20   }
```



CF(9) and DE(8) lost to cheaper edges. Edge EF(2): cheapest[E] and cheapest[F] = EF(2), cheaper than 3 for E and 6 for F.

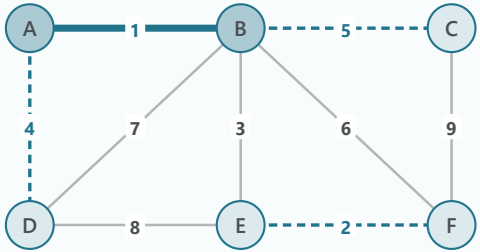
Step 5 / 14

VARIABLES

`e = EF(2)` `parent = [A, B, C, D, E, F]` `cheapest = [AB, AB, BC, AD, EF, EF]` `numComp = 6`

Borůvka: Step Through the Code

```
1  while (numComp > 1) {
2    Edge[] cheapest = new Edge[V];
3    for (Edge e : edges) {      // 1) cheapest
4      int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
5      if (k1 == k2) continue;
6      if (cheapest[k1] == null || e.w <
   cheapest[k1].w)
7        cheapest[k1] = e;
8      if (cheapest[k2] == null || e.w <
   cheapest[k2].w)
9        cheapest[k2] = e;
10   }
11   boolean added = false;
12   for (Edge e : cheapest) {   // 2) add them
13     if (e == null) continue;
▶    int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
15     if (k1 == k2) continue;
▶    tree.add(e);
▶    parent[k1] = k2;        // union
▶    numComp--;
▶    added = true;
20   }
```



AB(1) joins the tree; find gives roots A and B → parent[A] = B. Components: 5.

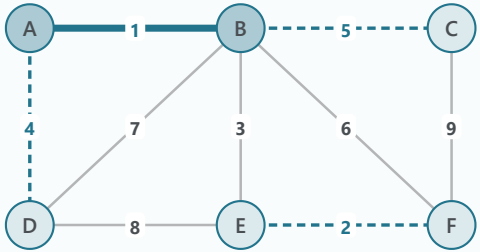
Step 6 / 14

VARIABLES

`e = AB(1)` `parent = [B, B, C, D, E, F]` `cheapest = [AB, AB, BC, AD, EF, EF]` `numComp = 5`

Borůvka: Step Through the Code

```
1  while (numComp > 1) {
2    Edge[] cheapest = new Edge[V];
3    for (Edge e : edges) {      // 1) cheapest
4      int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
5      if (k1 == k2) continue;
6      if (cheapest[k1] == null || e.w <
   cheapest[k1].w)
7        cheapest[k1] = e;
8      if (cheapest[k2] == null || e.w <
   cheapest[k2].w)
9        cheapest[k2] = e;
10   }
11   boolean added = false;
12   for (Edge e : cheapest) {   // 2) add them
▶      if (e == null) continue;
▶      int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
▶      if (k1 == k2) continue;
16     tree.add(e);
17     parent[k1] = k2;          // union
18     numComp--;
19     added = true;
20   }
```



cheapest[B] = AB(1), but both ends are now in one component: already added, skip.

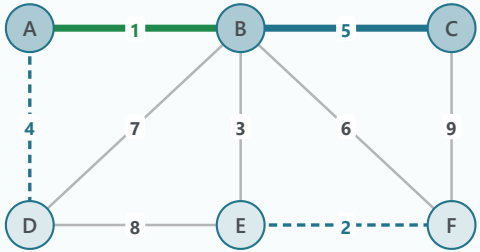
Step 7 / 14

VARIABLES

`e = AB(1)` `parent = [B, B, C, D, E, F]` `cheapest = [AB, AB, BC, AD, EF, EF]` `numComp = 5`

Borůvka: Step Through the Code

```
1  while (numComp > 1) {
2    Edge[] cheapest = new Edge[V];
3    for (Edge e : edges) {      // 1) cheapest
4      int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
5      if (k1 == k2) continue;
6      if (cheapest[k1] == null || e.w <
   cheapest[k1].w)
7        cheapest[k1] = e;
8      if (cheapest[k2] == null || e.w <
   cheapest[k2].w)
9        cheapest[k2] = e;
10   }
11   boolean added = false;
12   for (Edge e : cheapest) {   // 2) add them
13     if (e == null) continue;
▶    int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
15     if (k1 == k2) continue;
▶    tree.add(e);
▶    parent[k1] = k2;          // union
▶    numComp--;
▶    added = true;
20   }
```



BC(5) joins the tree; find gives roots B and C → parent[B] = C. Components: 4.

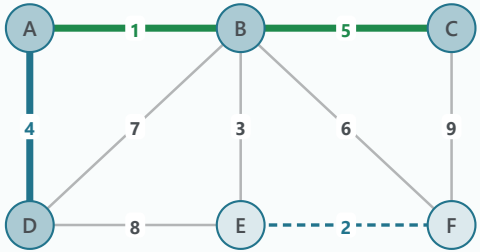
Step 8 / 14

VARIABLES

`e = BC(5)` `parent = [B, C, C, D, E, F]` `cheapest = [AB, AB, BC, AD, EF, EF]` `numComp = 4`

Borůvka: Step Through the Code

```
1  while (numComp > 1) {
2    Edge[] cheapest = new Edge[V];
3    for (Edge e : edges) {      // 1) cheapest
4      int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
5      if (k1 == k2) continue;
6      if (cheapest[k1] == null || e.w <
   cheapest[k1].w)
7        cheapest[k1] = e;
8      if (cheapest[k2] == null || e.w <
   cheapest[k2].w)
9        cheapest[k2] = e;
10   }
11   boolean added = false;
12   for (Edge e : cheapest) {   // 2) add them
13     if (e == null) continue;
▶      int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
15     if (k1 == k2) continue;
▶      tree.add(e);
▶      parent[k1] = k2;          // union
▶      numComp--;
▶      added = true;
20   }
```



AD(4) joins the tree; find gives roots C and D → parent[C] = D. Components: 3.

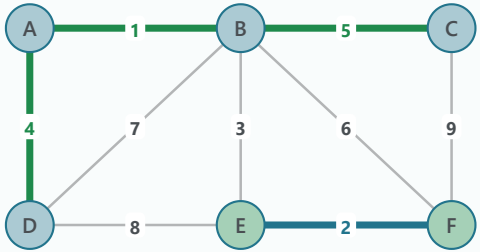
Step 9 / 14

VARIABLES

`e = AD(4)` `parent = [C, C, D, D, E, F]` `cheapest = [AB, AB, BC, AD, EF, EF]` `numComp = 3`

Borůvka: Step Through the Code

```
1  while (numComp > 1) {
2    Edge[] cheapest = new Edge[V];
3    for (Edge e : edges) {      // 1) cheapest
4      int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
5      if (k1 == k2) continue;
6      if (cheapest[k1] == null || e.w <
   cheapest[k1].w)
7        cheapest[k1] = e;
8      if (cheapest[k2] == null || e.w <
   cheapest[k2].w)
9        cheapest[k2] = e;
10   }
11   boolean added = false;
12   for (Edge e : cheapest) {   // 2) add them
13     if (e == null) continue;
▶      int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
15     if (k1 == k2) continue;
▶      tree.add(e);
▶      parent[k1] = k2;        // union
▶      numComp--;
▶      added = true;
20   }
```



EF(2) joins the tree; find gives roots E and F → parent[E] = F. Components: 2.

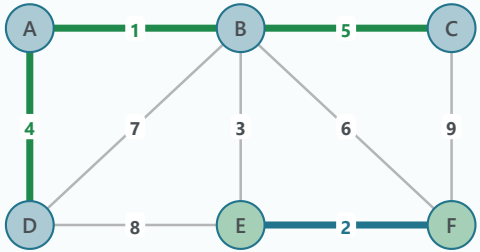
Step 10 / 14

VARIABLES

`e = EF(2)` `parent = [C, C, D, D, F, F]` `cheapest = [AB, AB, BC, AD, EF, EF]` `numComp = 2`

Borůvka: Step Through the Code

```
1  while (numComp > 1) {
2    Edge[] cheapest = new Edge[V];
3    for (Edge e : edges) {      // 1) cheapest
4      int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
5      if (k1 == k2) continue;
6      if (cheapest[k1] == null || e.w <
   cheapest[k1].w)
7        cheapest[k1] = e;
8      if (cheapest[k2] == null || e.w <
   cheapest[k2].w)
9        cheapest[k2] = e;
10   }
11   boolean added = false;
12   for (Edge e : cheapest) {   // 2) add them
▶      if (e == null) continue;
▶      int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
▶      if (k1 == k2) continue;
16     tree.add(e);
17     parent[k1] = k2;          // union
18     numComp--;
19     added = true;
20   }
```



cheapest[F] = EF(2), but both ends are now in one component: already added, skip.

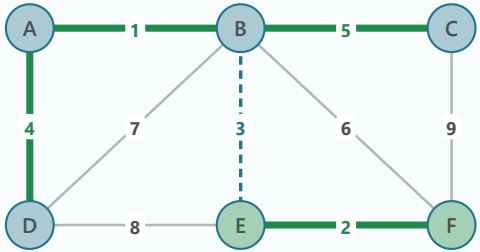
Step 11 / 14

VARIABLES

`e = EF(2)` `parent = [C, C, D, D, F, F]` `cheapest = [AB, AB, BC, AD, EF, EF]` `numComp = 2`

Borůvka: Step Through the Code

```
1  while (numComp > 1) {
▶    Edge[] cheapest = new Edge[V];
▶    for (Edge e : edges) {      // 1) cheapest
▶      int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
▶      if (k1 == k2) continue;
▶      if (cheapest[k1] == null || e.w <
   cheapest[k1].w)
▶        cheapest[k1] = e;
▶      if (cheapest[k2] == null || e.w <
   cheapest[k2].w)
▶        cheapest[k2] = e;
10   }
11   boolean added = false;
12   for (Edge e : cheapest) {   // 2) add them
13     if (e == null) continue;
14     int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
15     if (k1 == k2) continue;
16     tree.add(e);
17     parent[k1] = k2;          // union
18     numComp--;
19     added = true;
20   }
```



Round 2: AB, BC, AD, BD, EF stay inside a component (continue). Among the edges between components, BE(3), BF(6), CF(9), DE(8), the cheapest is BE(3): both roots' cheapest is BE.

Step 12 / 14

VARIABLES

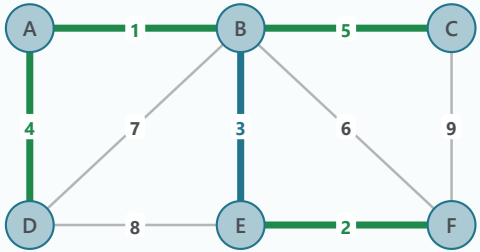
`e = –` `parent = [D, D, D, D, F, F]`

`cheapest = [–, –, –, BE, –, BE]`

`numComp = 2`

Borůvka: Step Through the Code

```
while (numComp > 1) {
  Edge[] cheapest = new Edge[V];
  for (Edge e : edges) {      // 1) cheapest
    int k1 = find(parent, e.u), k2 = find(parent, e.v);
    if (k1 == k2) continue;
    if (cheapest[k1] == null || e.w < cheapest[k1].w)
      cheapest[k1] = e;
    if (cheapest[k2] == null || e.w < cheapest[k2].w)
      cheapest[k2] = e;
  }
  boolean added = false;
  for (Edge e : cheapest) {   // 2) add them
    if (e == null) continue;
    int k1 = find(parent, e.u), k2 = find(parent, e.v);
    if (k1 == k2) continue;
    tree.add(e);
    parent[k1] = k2;          // union
    numComp--;
    added = true;
  }
```



BE(3) joins the tree; find gives roots D and F → parent[D] = F. Components: 1. One component is left.

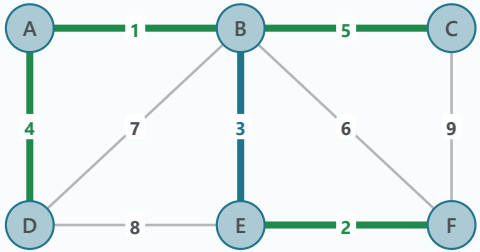
Step 13 / 14

VARIABLES

`e = BE(3)` `parent = [D, D, D, F, F, F]` `cheapest = [–, –, –, BE, –, BE]` `numComp = 1`

Borůvka: Step Through the Code

```
1  while (numComp > 1) {
2    Edge[] cheapest = new Edge[V];
3    for (Edge e : edges) {      // 1) cheapest
4      int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
5      if (k1 == k2) continue;
6      if (cheapest[k1] == null || e.w <
   cheapest[k1].w)
7        cheapest[k1] = e;
8      if (cheapest[k2] == null || e.w <
   cheapest[k2].w)
9        cheapest[k2] = e;
10   }
11   boolean added = false;
12   for (Edge e : cheapest) {   // 2) add them
▶      if (e == null) continue;
▶      int k1 = find(parent, e.u), k2 = find(parent,
   e.v);
▶      if (k1 == k2) continue;
16     tree.add(e);
17     parent[k1] = k2;          // union
18     numComp--;
19     added = true;
20   }
```



cheapest[F] = BE(3), but both ends are now in one component: already added, skip. `numComp` = 1: the loop ends, MST weight **15**.

Step 14 / 14

VARIABLES

`e = BE(3)` `parent = [D, F, D, F, F, F]`

`cheapest = [–, –, –, BE, –, BE]`

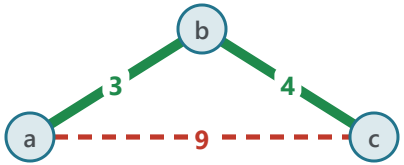
`numComp = 1`

The Reverse-Delete Algorithm

- A **greedy** MST algorithm (Kruskal proposed it in his 1956 paper).
- Sorts the edges by weight in **decreasing** order.
- Starting from the heaviest edge, deletes every edge whose removal **keeps the graph connected**.
- Relies on the **cycle property**: a deleted edge is the heaviest edge on some cycle.

Kruskal's mirror image

Kruskal **adds** the **lightest**, reverse-delete **deletes** the **heaviest** edges. One starts from the empty graph, the other from the full graph.



Use

Mostly of **theoretical and teaching** value; since every deletion needs a connectivity test, it is usually slow in practice.

Reverse-Delete: Steps, Pseudocode, Complexity

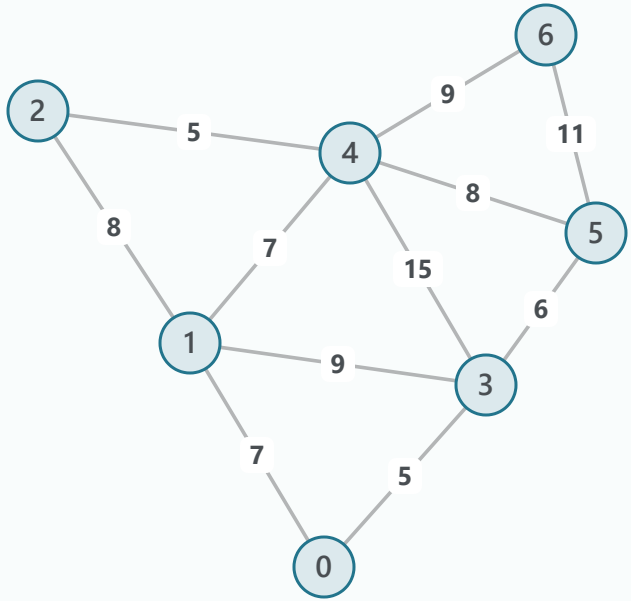
1. Sort all edges by weight in **decreasing** order.
2. Delete the next heaviest edge.
3. If the graph is **no longer connected**, put the deleted edge back.
4. Repeat steps 2 and 3 until every edge has been processed.

Complexity: one BFS/DFS per edge (or a union-find rebuild) → **O(E · (V + E))**. The O(E log E) cost of sorting is small next to that.

```
REVERSE_DELETE(G):
  MST = all edges
  edges.sort(by decreasing weight)
  for each (u, v, weight) in edges:
    MST.remove((u, v, weight))
    dsu.reset()
    for each (x, y) in MST:
      dsu.union(x, y)
    root = dsu.find(V[0])
    if dsu.find(v) ≠ root for some v:
      MST.add((u, v, weight))  // put back
  return MST
```

The connectivity test looks at every vertex; if **at least one** vertex is cut off from the root, the deleted edge is put back **once**.

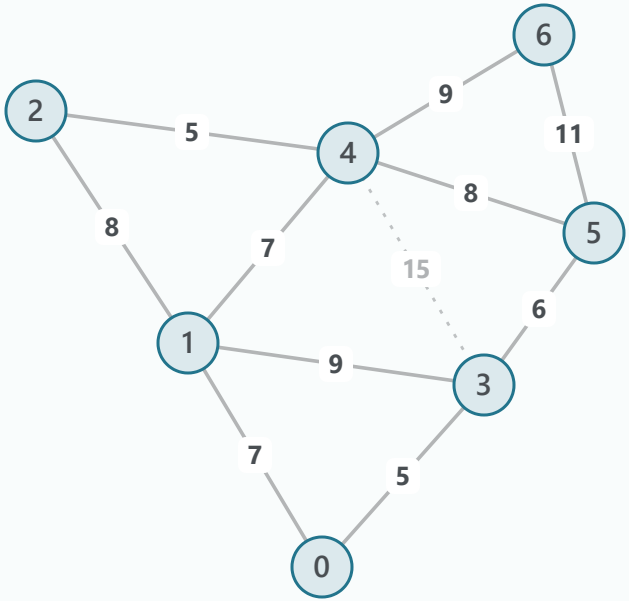
Reverse-Delete Step by Step: Seven Vertices



Edge	w	Status
3–4	15	
5–6	11	
1–3	9	
4–6	9	
1–2	8	
4–5	8	
1–4	7	
0–1	7	
3–5	6	
2–4	5	
0–3	5	

Edges in decreasing order: 15, 11, 9, 9, 8, 8, 7, 7, 6, 5, 5. Total weight 90.

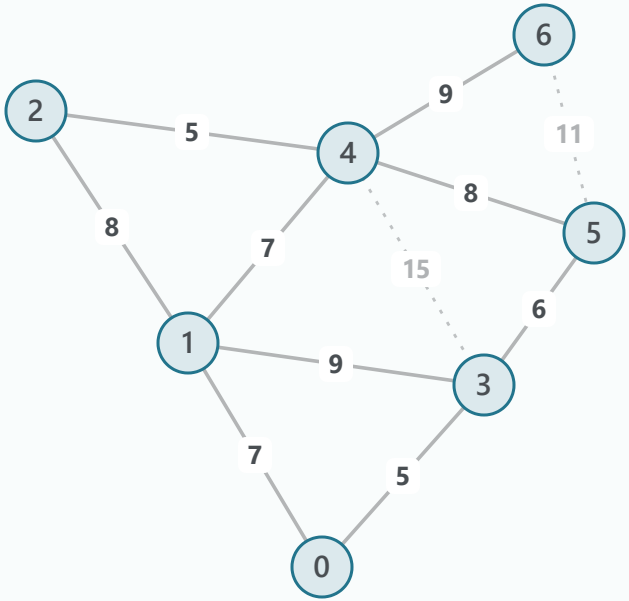
Reverse-Delete Step by Step: Seven Vertices



Edge	w	Status
~~3–4~~	~~15~~	~~deleted~~
5–6	11	
1–3	9	
4–6	9	
1–2	8	
4–5	8	
1–4	7	
0–1	7	
3–5	6	
2–4	5	
0–3	5	

Delete **3–4 = 15**: the graph stays connected (the edge is the heaviest on a cycle).

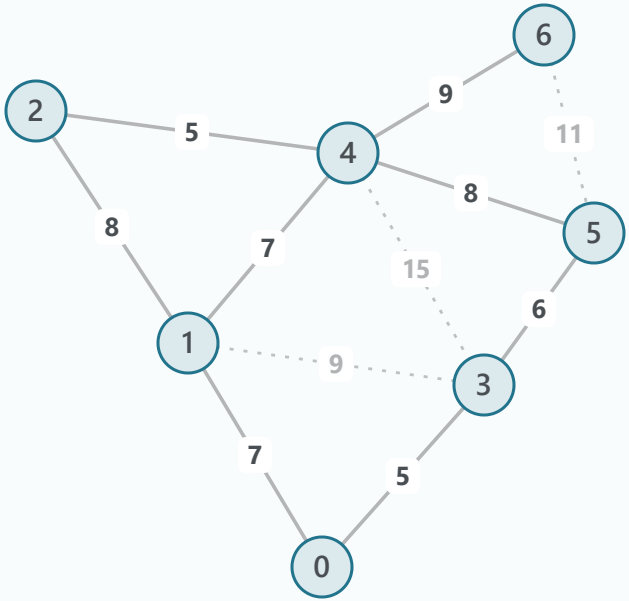
Reverse-Delete Step by Step: Seven Vertices



Edge	w	Status
~~3–4~~	~~15~~	~~deleted~~
~~5–6~~	~~11~~	~~deleted~~
1–3	9	
4–6	9	
1–2	8	
4–5	8	
1–4	7	
0–1	7	
3–5	6	
2–4	5	
0–3	5	

Delete **5–6 = 11**: the graph stays connected (the edge is the heaviest on a cycle).

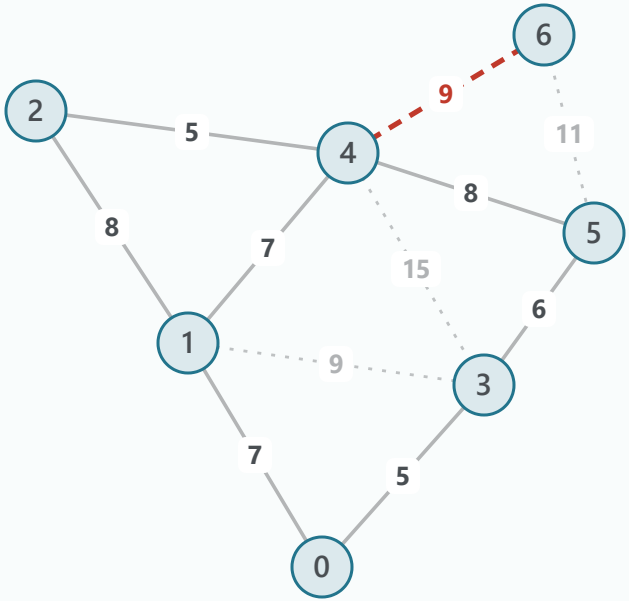
Reverse-Delete Step by Step: Seven Vertices



Edge	w	Status
~~3–4~~	~~15~~	~~deleted~~
~~5–6~~	~~11~~	~~deleted~~
~~1–3~~	~~9~~	~~deleted~~
4–6	9	
1–2	8	
4–5	8	
1–4	7	
0–1	7	
3–5	6	
2–4	5	
0–3	5	

Delete **1–3 = 9**: the graph stays connected (the edge is the heaviest on a cycle).

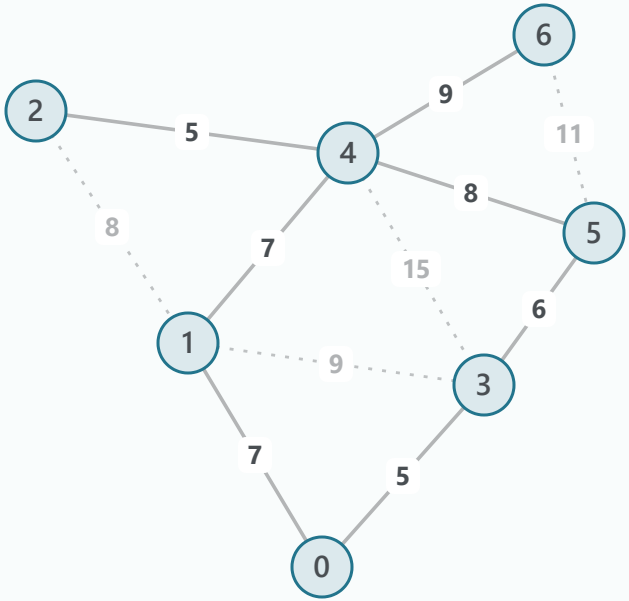
Reverse-Delete Step by Step: Seven Vertices



Edge	w	Status
~~3–4~~	~~15~~	~~deleted~~
~~5–6~~	~~11~~	~~deleted~~
~~1–3~~	~~9~~	~~deleted~~
4–6	**9**	**kept**
1–2	8	
4–5	8	
1–4	7	
0–1	7	
3–5	6	
2–4	5	
0–3	5	

Deleting **4–6 = 9** would isolate vertex 6 → connectivity breaks, so the edge is **put back**.

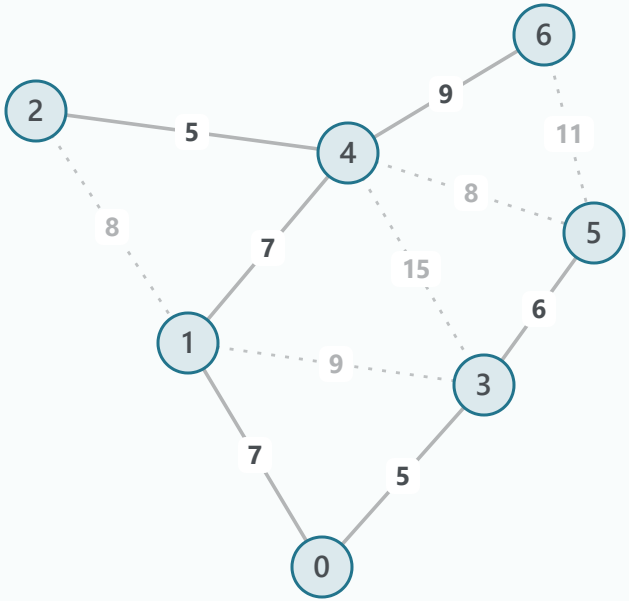
Reverse-Delete Step by Step: Seven Vertices



Edge	w	Status
~~3–4~~	~~15~~	~~deleted~~
~~5–6~~	~~11~~	~~deleted~~
~~1–3~~	~~9~~	~~deleted~~
4–6	**9**	**kept**
~~1–2~~	~~8~~	~~deleted~~
4–5	8	
1–4	7	
0–1	7	
3–5	6	
2–4	5	
0–3	5	

Delete **1–2 = 8**: the graph stays connected (the edge is the heaviest on a cycle).

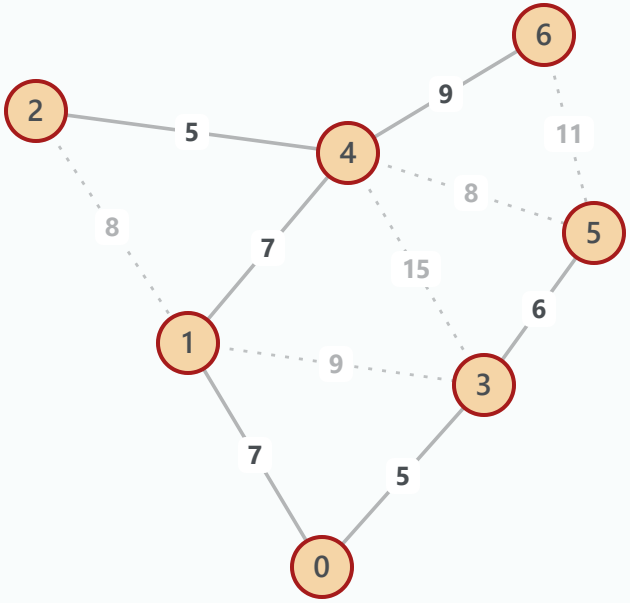
Reverse-Delete Step by Step: Seven Vertices



Edge	w	Status
~~3–4~~	~~15~~	~~deleted~~
~~5–6~~	~~11~~	~~deleted~~
~~1–3~~	~~9~~	~~deleted~~
4–6	**9**	**kept**
~~1–2~~	~~8~~	~~deleted~~
~~4–5~~	~~8~~	~~deleted~~
1–4	7	
0–1	7	
3–5	6	
2–4	5	
0–3	5	

Delete **4–5 = 8**: the graph stays connected (the edge is the heaviest on a cycle).

Reverse-Delete Step by Step: Seven Vertices

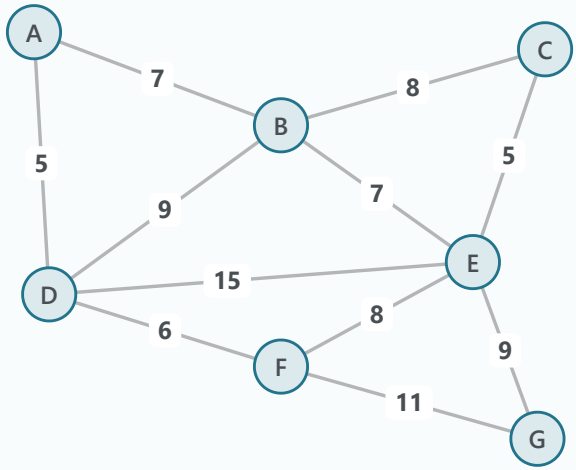


Edge	w	Status
~~3–4~~	~~15~~	~~deleted~~
~~5–6~~	~~11~~	~~deleted~~
~~1–3~~	~~9~~	~~deleted~~
4–6	**9**	**kept**
~~1–2~~	~~8~~	~~deleted~~
~~4–5~~	~~8~~	~~deleted~~
1–4	**7**	**kept**
0–1	**7**	**kept**
3–5	**6**	**kept**
2–4	**5**	**kept**
0–3	**5**	**kept**

The remaining 6 edges (7, 7, 6, 5, 5 and 9) are all **bridges**: none can be deleted. 6 = V − 1 edges, **MST = 39**.

Reverse-Delete: Step Through the Code

```
1  void reverseDelete(int V, List<Edge> es) {
▶    es.sort((a, b) -> b.w - a.w); // descending
3    int i = 0;
4    while (i < es.size()) {
5      Edge e = es.remove(i);      // try
   deleting
6      if (!isConnected(V, es)) {
7        es.add(i, e);             // put back
8        i++;
9      }
10   }
11 }
```



Sort the edges in decreasing order: DE 15, FG 11, BD 9, EG 9, BC 8, EF 8, AB 7, BE 7, DF 6, AD 5, CE 5.

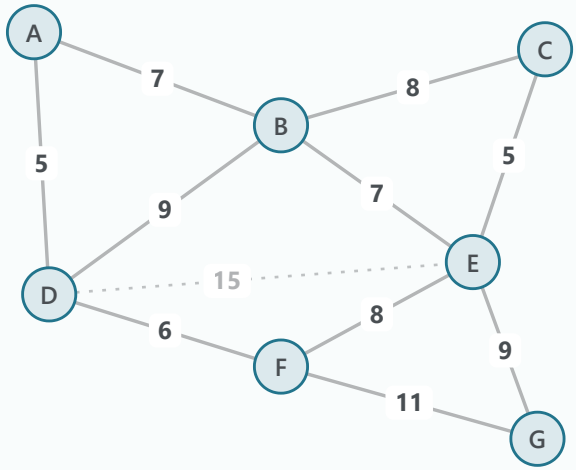
Step 1 / 13

VARIABLES

`i = 0` `size = 11` `total = 90`

Reverse-Delete: Step Through the Code

```
1  void reverseDelete(int V, List<Edge> es) {
2    es.sort((a, b) -> b.w - a.w); // descending
3    int i = 0;
4    while (i < es.size()) {
▶      Edge e = es.remove(i);       // try deleting
▶      if (!isConnected(V, es)) {
7        es.add(i, e);              // put back
8        i++;
9      }
10   }
11 }
```



Heaviest: **DE = 15** deleted, still connected.

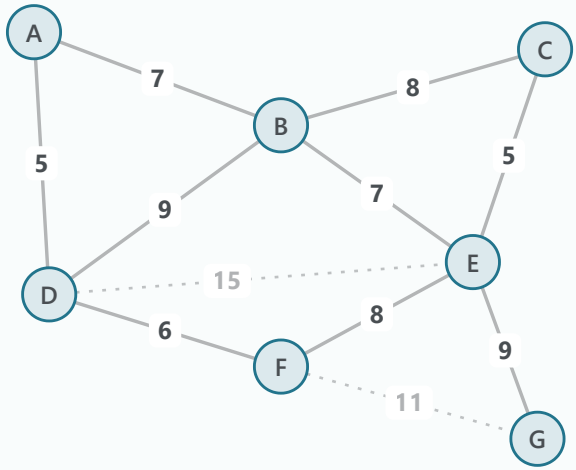
Step 2 / 13

VARIABLES

`i = 0` `e = DE(15)` `isConnected = true` `size = 10` `total = 75`

Reverse-Delete: Step Through the Code

```
1  void reverseDelete(int V, List<Edge> es) {
2    es.sort((a, b) -> b.w - a.w); // descending
3    int i = 0;
4    while (i < es.size()) {
▶      Edge e = es.remove(i);       // try deleting
▶      if (!isConnected(V, es)) {
7        es.add(i, e);              // put back
8        i++;
9      }
10   }
11 }
```



FG = 11 deleted, still connected.

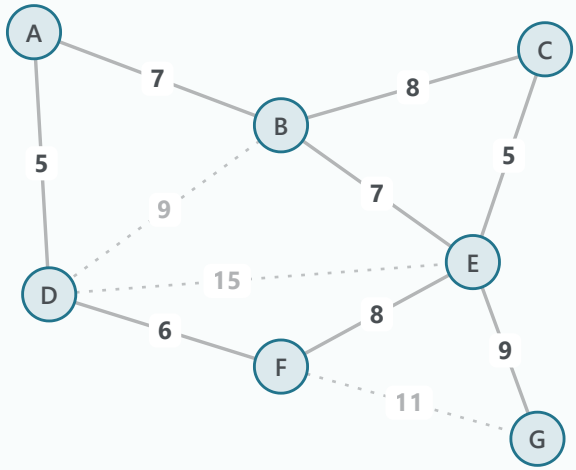
Step 3 / 13

VARIABLES

`i = 0` `e = FG(11)` `isConnected = true` `size = 9` `total = 64`

Reverse-Delete: Step Through the Code

```
1  void reverseDelete(int V, List<Edge> es) {
2    es.sort((a, b) -> b.w - a.w); // descending
3    int i = 0;
4    while (i < es.size()) {
▶      Edge e = es.remove(i);       // try deleting
▶      if (!isConnected(V, es)) {
7        es.add(i, e);              // put back
8        i++;
9      }
10   }
11 }
```



BD = 9 deleted, still connected.

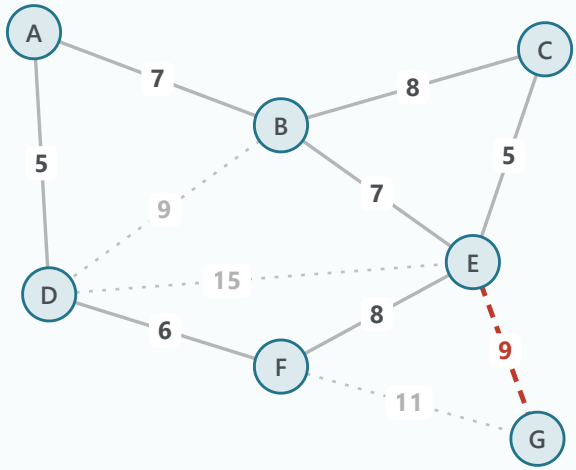
Step 4 / 13

VARIABLES

`i = 0` `e = BD(9)` `isConnected = true` `size = 8` `total = 55`

Reverse-Delete: Step Through the Code

```
1  void reverseDelete(int V, List<Edge> es) {
2    es.sort((a, b) -> b.w - a.w); // descending
3    int i = 0;
4    while (i < es.size()) {
5      Edge e = es.remove(i);       // try deleting
▶      if (!isConnected(V, es)) {
▶        es.add(i, e);              // put back
▶        i++;
9      }
10   }
11 }
```



EG = 9 would isolate G: back, `i` = 1.

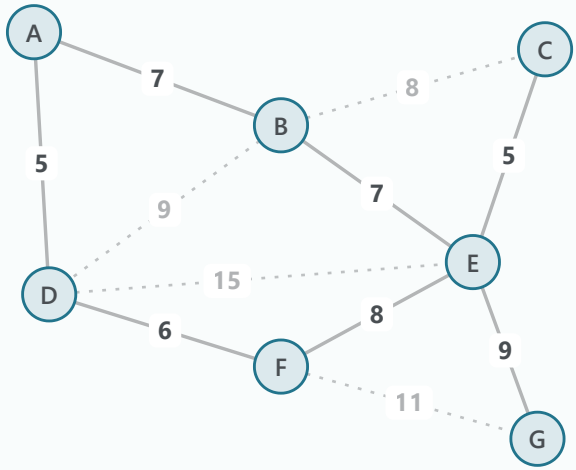
Step 5 / 13

VARIABLES

`i = 1` `e = EG(9)` `isConnected = false` `size = 8` `total = 55`

Reverse-Delete: Step Through the Code

```
1  void reverseDelete(int V, List<Edge> es) {
2    es.sort((a, b) -> b.w - a.w); // descending
3    int i = 0;
4    while (i < es.size()) {
▶      Edge e = es.remove(i);       // try deleting
▶      if (!isConnected(V, es)) {
7        es.add(i, e);              // put back
8        i++;
9      }
10   }
11 }
```



BC = 8 deleted, still connected.

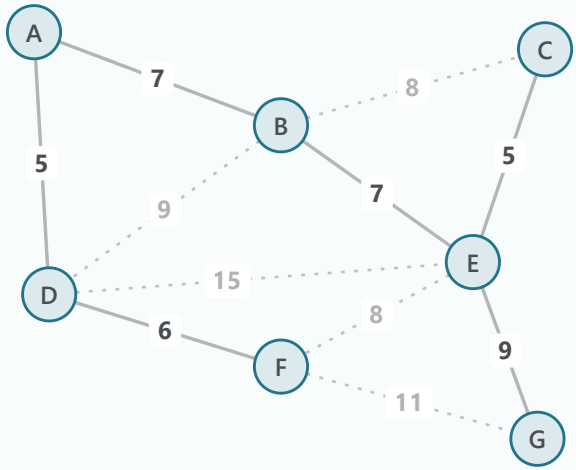
Step 6 / 13

VARIABLES

`i = 1` `e = BC(8)` `isConnected = true` `size = 7` `total = 47`

Reverse-Delete: Step Through the Code

```
1  void reverseDelete(int V, List<Edge> es) {
2    es.sort((a, b) -> b.w - a.w); // descending
3    int i = 0;
4    while (i < es.size()) {
▶      Edge e = es.remove(i);       // try deleting
▶      if (!isConnected(V, es)) {
7        es.add(i, e);              // put back
8        i++;
9      }
10   }
11 }
```



EF = 8 deleted, still connected.

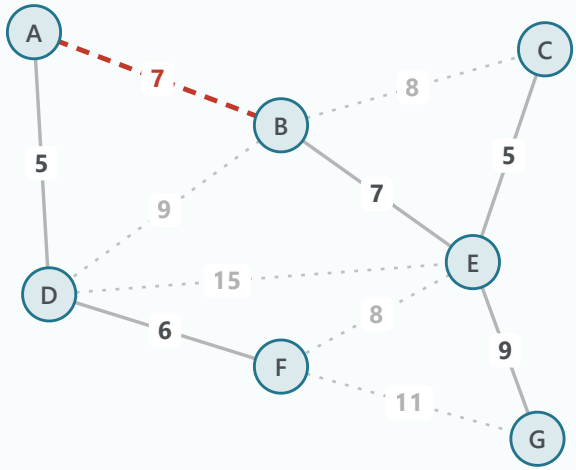
Step 7 / 13

VARIABLES

`i = 1` `e = EF(8)` `isConnected = true` `size = 6` `total = 39`

Reverse-Delete: Step Through the Code

```
1  void reverseDelete(int V, List<Edge> es) {
2    es.sort((a, b) -> b.w - a.w); // descending
3    int i = 0;
4    while (i < es.size()) {
5      Edge e = es.remove(i);      // try deleting
▶      if (!isConnected(V, es)) {
▶        es.add(i, e);             // put back
▶        i++;
9      }
10   }
11 }
```



AB = 7 cuts off {A, D, F}: back, `i` = 2.

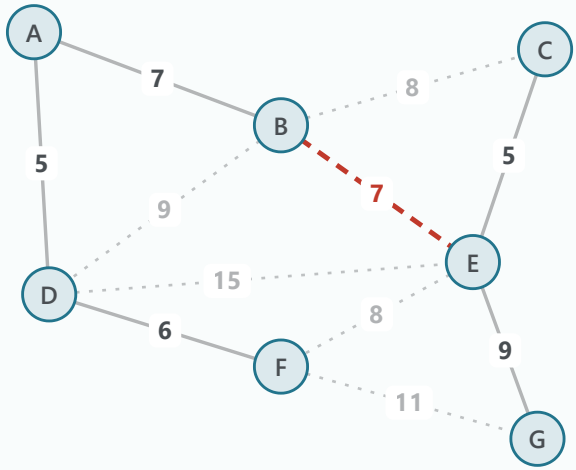
Step 8 / 13

VARIABLES

`i = 2` `e = AB(7)` `isConnected = false` `size = 6` `total = 39`

Reverse-Delete: Step Through the Code

```
1  void reverseDelete(int V, List<Edge> es) {
2    es.sort((a, b) -> b.w - a.w); // descending
3    int i = 0;
4    while (i < es.size()) {
5      Edge e = es.remove(i);      // try
   deleting
▶      if (!isConnected(V, es)) {
▶        es.add(i, e);             // put back
▶        i++;
9      }
10   }
11 }
```



BE = 7 cuts off {C, E, G}: back, `i` = 3.

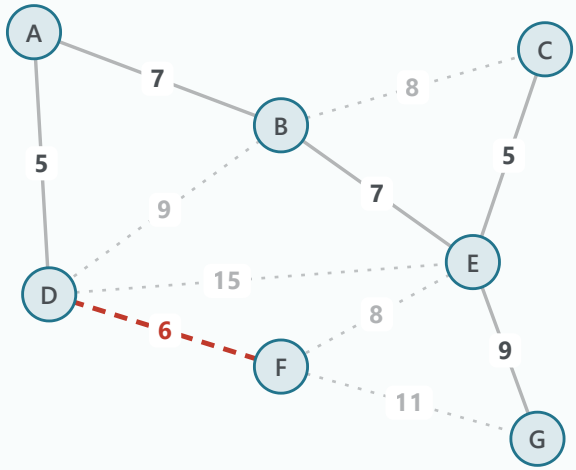
Step 9 / 13

VARIABLES

`i = 3` `e = BE(7)` `isConnected = false` `size = 6` `total = 39`

Reverse-Delete: Step Through the Code

```
1  void reverseDelete(int V, List<Edge> es) {
2    es.sort((a, b) -> b.w - a.w); // descending
3    int i = 0;
4    while (i < es.size()) {
5      Edge e = es.remove(i);      // try deleting
▶        if (!isConnected(V, es)) {
▶          es.add(i, e);           // put back
▶          i++;
9        }
10     }
11   }
```



DF = 6 cuts off {F}: back, `i` = 4.

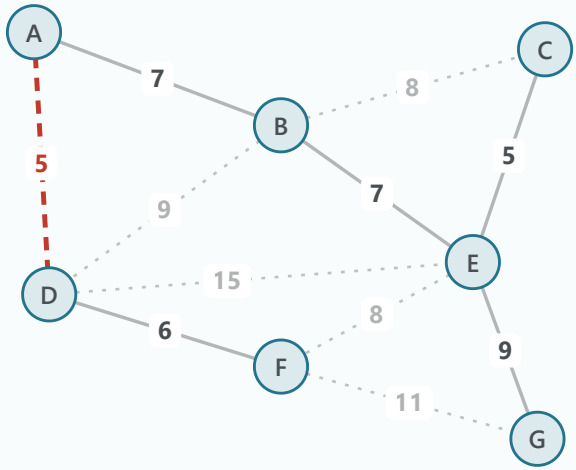
Step 10 / 13

VARIABLES

`i = 4` `e = DF(6)` `isConnected = false` `size = 6` `total = 39`

Reverse-Delete: Step Through the Code

```
1  void reverseDelete(int V, List<Edge> es) {
2    es.sort((a, b) -> b.w - a.w); // descending
3    int i = 0;
4    while (i < es.size()) {
5      Edge e = es.remove(i);      // try
   deleting
▶      if (!isConnected(V, es)) {
▶        es.add(i, e);             // put back
▶        i++;
9      }
10   }
11 }
```



AD = 5 cuts off {D, F}: back, `i` = 5.

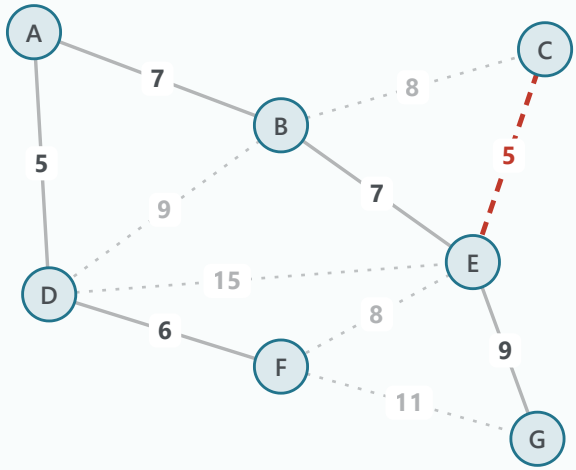
Step 11 / 13

VARIABLES

`i = 5` `e = AD(5)` `isConnected = false` `size = 6` `total = 39`

Reverse-Delete: Step Through the Code

```
1  void reverseDelete(int V, List<Edge> es) {
2    es.sort((a, b) -> b.w - a.w); // descending
3    int i = 0;
4    while (i < es.size()) {
5      Edge e = es.remove(i);      // try deleting
▶        if (!isConnected(V, es)) {
▶          es.add(i, e);           // put back
▶          i++;
9        }
10     }
11   }
```



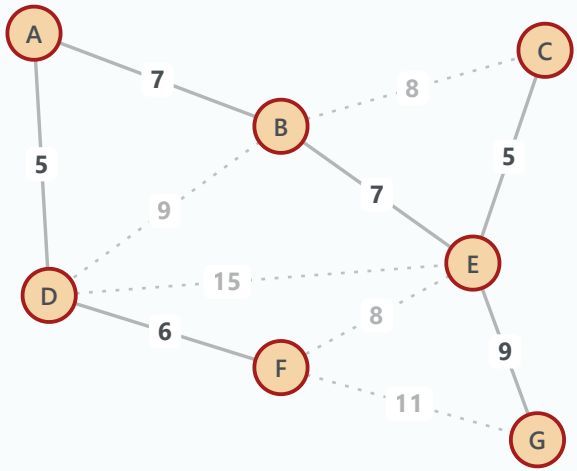
CE = 5 cuts off {C}: back, `i` = 6.

VARIABLES

`i = 6` `e = CE(5)` `isConnected = false` `size = 6` `total = 39`

Reverse-Delete: Step Through the Code

```
1  void reverseDelete(int V, List<Edge> es) {
2    es.sort((a, b) -> b.w - a.w); // descending
3    int i = 0;
▶    while (i < es.size()) {
5      Edge e = es.remove(i);      // try deleting
6      if (!isConnected(V, es)) {
7        es.add(i, e);             // put back
8        i++;
9      }
10   }
11 }
```



`i = size = 6` : the loop ends. No more edges to delete; the remaining tree is AB, AD, BE, CE, DF, EG → **w = 39**.

Step 13 / 13

VARIABLES

`i = 6` `size = 6` `total = 39`

COMPARISON

Comparing the Four Algorithms

	Kruskal	Prim	Borůvka	Reverse-delete
Viewpoint	edge-centered	vertex-centered	component-centered	edge-centered (deletion)
Intermediate state	forest	one tree	forest	connected graph
Property used	cut	cut	cut	cycle
Data structure	sorted edges + union-find	priority queue / array	union-find	BFS/DFS
Time	$O(E \log E)$	$O(V^2)$ or $O((V + E) \log V)$	$O(E \log V)$	$O(E \cdot (V + E))$
Best for	sparse graphs	dense graphs	parallel / distributed	teaching, theory
Disconnected graph	gives a forest	one component	gives a forest	keeps the components

On the nine-vertex graph Kruskal and Prim find the same tree (29) but add edges in a different **order**: Kruskal 1, 1, 2, 3, 4, 5, 6, 7; Prim 4, 7, 1, 1, 2, 3, 6, 5.

Summary

Topic	Key points
MST	spanning tree of minimum total weight in a connected, undirected, weighted graph; V − 1 edges
Properties	the lightest edge across a cut is safe; the heaviest edge on a cycle is out; distinct weights → unique MST
Kruskal	sort the edges; if `find(u) ≠ find(v)`, add the edge and merge the trees — O(E log E)
Union-Find	a `parent[]` forest; path compression + rank → O(α(V)) per operation
Prim	grow a single tree; key = cheapest edge to the tree — $O(V^2)$ / O((V + E) log V)
Borůvka	every component adds its cheapest outgoing edge at once; ≤ $\log_2$ V rounds — O(E log V)
Reverse-delete	delete in decreasing order, put back if connectivity breaks — O(E · (V + E))

SUMMARY

Common Mistakes

Confusing the MST with a shortest-path tree

The MST minimizes total weight; the MST path between two vertices need not be a shortest path.

Checking cycles by adjacency

In Kruskal, "are u and v directly connected?" is not enough; ask whether they are **in the same component** (find).

Adding up distances in Prim

Writing `key[u] + w` gives Dijkstra. In Prim the key is the weight of a **single edge**.

Fearing negative weights

MST algorithms work correctly with negative weights too (they only compare); the problem is specific to Dijkstra.

Disconnected graphs

There is no MST, only a minimum spanning **forest**. `while (tree.size() < V - 1)` runs past the end of the edge list.

Applying them to directed graphs

These algorithms are for undirected graphs. The directed counterpart (minimum spanning arborescence) is the Chu–Liu/Edmonds algorithm.

SUMMARY

Worked Problems

Graph: **AB = 2, AC = 3, BC = 1, BD = 4, CD = 5, CE = 6, DE = 7** (undirected).

1. List the edges Kruskal examines, in order; which are rejected? w(T) = ?
2. Run Prim from A: in what order do the vertices enter the tree? What is the key array after the first step?
3. How many rounds does Borůvka take? Which edges are chosen in round 1?
4. Which edges does reverse-delete remove?
5. How many edges does the MST of a connected graph with 100 vertices have?
6. If 5 is added to every weight, does the MST change? What is the new total?

▶ **Show answers**

SUMMARY

Exercises

1. Use the cut property to show that the MST is unique when all weights are distinct.
2. How would you change Kruskal to find a **maximum** spanning tree?
3. Use the MST of the nine-vertex example to split the vertices into **3 clusters**.
4. On 0..5 with `parent[find(x)] = find(y)`: union (0,1), (2,3), (1,3), (4,5), (3,5). What is the final `parent`? What does `find(0)` change?
5. V = 1000: compare the $O(V^2)$ and $O(E \log V)$ versions of Prim for E = 5,000 and 500,000.
6. Fix Kruskal so that it returns a forest on a disconnected graph.

▶ **Answers / hints**

References

- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapter 19: disjoint sets, Chapter 21: MSTs).
- Sedgewick and Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (Section 4.3: Minimum Spanning Trees).
- J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", *Proc. AMS* 7(1), 1956.
- R. C. Prim, "Shortest Connection Networks and Some Generalizations", *Bell System Technical Journal* 36(6), 1957.
- O. Borůvka, "O jistém problému minimálním", *Práce Moravské přírodovědecké společnosti* 3, 1926.
- R. E. Tarjan, "Efficiency of a Good But Not Linear Set Union Algorithm", *Journal of the ACM* 22(2), 1975.

The End

Chapter 7: Minimum Spanning Trees

Previous: Chapter 6: Shortest Paths · Next: Chapter 8: Network Flow