Chapter 17: Minimum Spanning Trees

Data Structures and Algorithms with C

Spanning trees | Isomorphism | Greedy method | Kruskal | Union-find | Prim | Heap

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/dsa_c/slides/Chapter_17_Minimum_Spanning_Trees.html

Learning Outcomes

Define

Explain spanning trees, tree and nontree edges, and depth-first and breadth-first spanning trees.

Count

Discuss ways to generate the spanning trees of a graph, and the idea of isomorphism.

Kruskal

Trace Kruskal's algorithm in C: edges in increasing cost, cycles caught with **union-find**.

Prim

Implement Prim's algorithm, which grows a single tree, with $O(n^2)$ arrays and with a heap.

Analyze

Compare the O(e log e), $O(n^2)$ and O(e log n) bounds and choose by graph density.

Approach

“Discovering the data structure”: how do a min heap and sets speed up the algorithm?

A Traversal Splits the Edges in Two

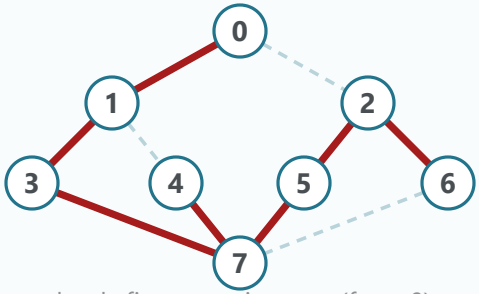
When DFS or BFS visits every vertex of a connected graph, the edges fall into two groups:

T: tree edges

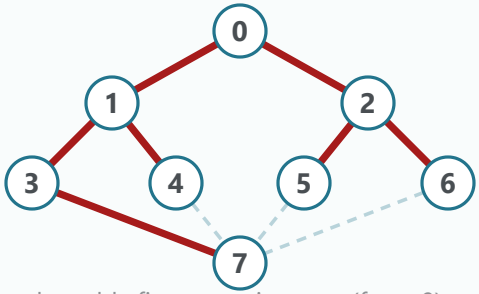
Edges that were followed to a new vertex (thick red). There are exactly n − 1 of them.

N: nontree edges

Edges never followed (dashed). In DFS they are *back* edges, in BFS *cross* edges.



depth-first spanning tree (from 0)

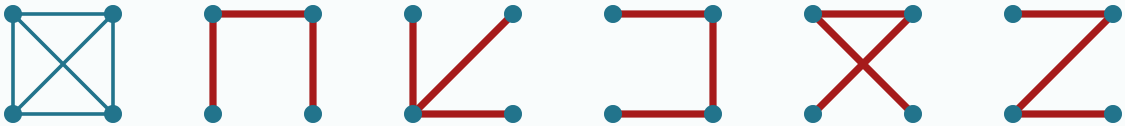


breadth-first spanning tree (from 0)

DFS: N = {(0,2), (1,4), (6,7)} · BFS: N = {(4,7), (5,7), (6,7)}

Definition: Spanning Tree

Definition: a spanning tree is a **tree** made of edges of the graph that connects all the vertices: n vertices, **n − 1** edges, no cycle.



K4 a) b) c) d) e)

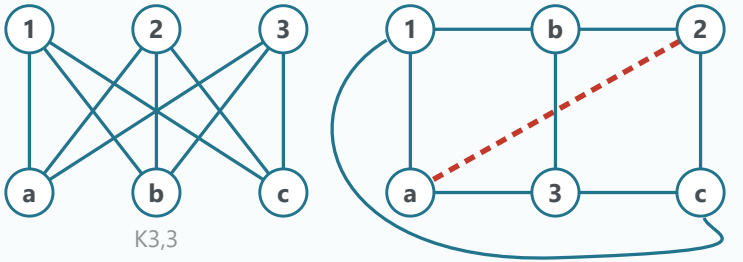
Isomorphism

K4 is a complete graph. a and b are two **different** spanning trees (a is a path, b a star); c, d and e are **isomorphic** to a: a ≡ c ≡ d ≡ e.

How many?

K4 has 16 spanning trees (Cayley: $n^{n-2} = 4^2 = 16$): 12 paths + 4 stars. Once weights come in, “how many?” is asked again.

Isomorphism, Drawings and Planarity



The same graph ($K_{3,3}$), two drawings

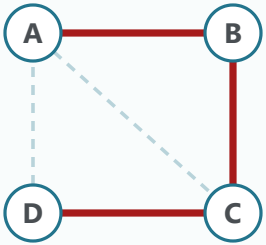
- DFS/BFS help test isomorphism, but in general it is a **hard** problem.
- A graph that can be drawn without crossing edges is **planar**. $K_{3,3}$ is not planar: at least one crossing is needed.
- In circuit design, a $K_{3,3}$-shaped circuit needs at least **two layers**.

Instructor: "This would make a nice senior project: a program, an algorithm that shows a ≡ e with a 3D view… Which vertices have to swap places?"

Instructor: "These are hard problems… You can work on them in a master's or PhD. And by that we are also saying: 'Without a master's or PhD, the problem won't get solved.'"

Question: How Many Spanning Trees Are There?

n − 1 of the e edges can be chosen in **C(e, n − 1)** ways; each choice is checked for a cycle and isomorphic copies are removed. Here 8 of the C(5, 3) = 10 choices are trees.



initial tree T (3 edges)

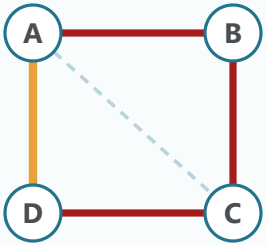
First find **one** spanning tree (e.g. with DFS).

Step 1 / 6

The algorithm is long, so it is not a sensible choice. (For the exact count, use Kirchhoff's matrix-tree theorem, an advanced topic.)

Question: How Many Spanning Trees Are There?

n − 1 of the e edges can be chosen in **C(e, n − 1)** ways; each choice is checked for a cycle and isomorphic copies are removed. Here 8 of the C(5, 3) = 10 choices are trees.



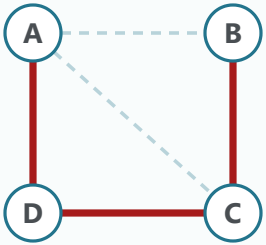
+ (D,A) → cycle A-B-C-D-A

Add a nontree edge: exactly **one** cycle appears.

The algorithm is long, so it is not a sensible choice. (For the exact count, use Kirchhoff's matrix-tree theorem, an advanced topic.)

Question: How Many Spanning Trees Are There?

n − 1 of the e edges can be chosen in **C(e, n − 1)** ways; each choice is checked for a cycle and isomorphic copies are removed. Here 8 of the C(5, 3) = 10 choices are trees.



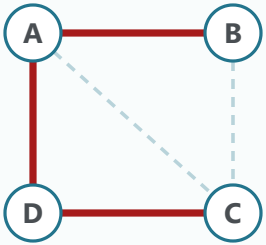
(A,B) removed → new tree

Remove (A,B) from the cycle → a different spanning tree. Put it back and remove the next one.

The algorithm is long, so it is not a sensible choice. (For the exact count, use Kirchhoff's matrix-tree theorem, an advanced topic.)

Question: How Many Spanning Trees Are There?

n − 1 of the e edges can be chosen in **C(e, n − 1)** ways; each choice is checked for a cycle and isomorphic copies are removed. Here 8 of the C(5, 3) = 10 choices are trees.



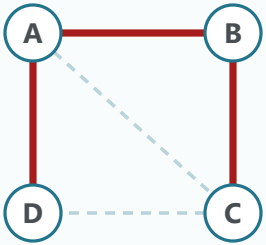
(B,C) removed → new tree

Remove (B,C) from the cycle → a different spanning tree. Put it back and remove the next one.

The algorithm is long, so it is not a sensible choice. (For the exact count, use Kirchhoff's matrix-tree theorem, an advanced topic.)

Question: How Many Spanning Trees Are There?

n − 1 of the e edges can be chosen in **C(e, n − 1)** ways; each choice is checked for a cycle and isomorphic copies are removed. Here 8 of the C(5, 3) = 10 choices are trees.



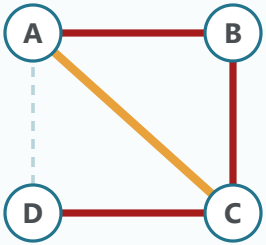
(C,D) removed → new tree

Remove (C,D) from the cycle → a different spanning tree. Put it back and remove the next one.

The algorithm is long, so it is not a sensible choice. (For the exact count, use Kirchhoff's matrix-tree theorem, an advanced topic.)

Question: How Many Spanning Trees Are There?

n − 1 of the e edges can be chosen in **C(e, n − 1)** ways; each choice is checked for a cycle and isomorphic copies are removed. Here 8 of the C(5, 3) = 10 choices are trees.

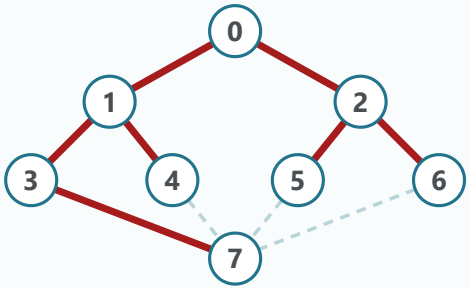


8 different trees in all

Repeat for every nontree edge. This graph has 8 spanning trees, but the method is long and produces duplicates.

The algorithm is long, so it is not a sensible choice. (For the exact count, use Kirchhoff's matrix-tree theorem, an advanced topic.)

Properties of Spanning Trees

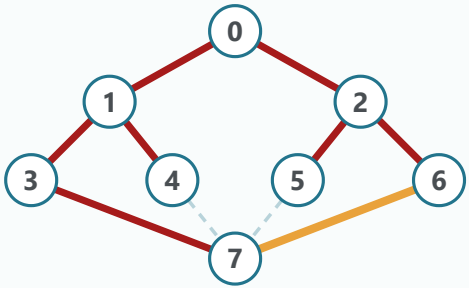


BFS spanning tree: 8 vertices, 7 edges (thick red); the 3 nontree edges are dashed.

Step 1 / 3

- Exactly **n − 1** edges, no cycle, connected: any two of these imply the third.
- There is a **unique** path between any two vertices.
- Adding a nontree edge creates a single cycle; removing any edge of that cycle gives a new spanning tree (the generation method relies on this).
- Uses: dropping redundant links in a network, *broadcast* trees, circuit equations (Kirchhoff).

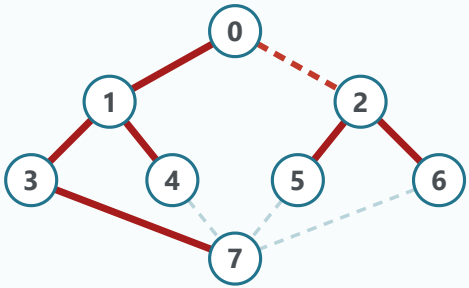
Properties of Spanning Trees



Adding the nontree edge (6,7) creates **exactly one cycle**: 6-2-0-1-3-7-6. It is no longer a tree.

- Exactly **n − 1** edges, no cycle, connected: any two of these imply the third.
- There is a **unique** path between any two vertices.
- Adding a nontree edge creates a single cycle; removing any edge of that cycle gives a new spanning tree (the generation method relies on this).
- Uses: dropping redundant links in a network, *broadcast* trees, circuit equations (Kirchhoff).

Properties of Spanning Trees

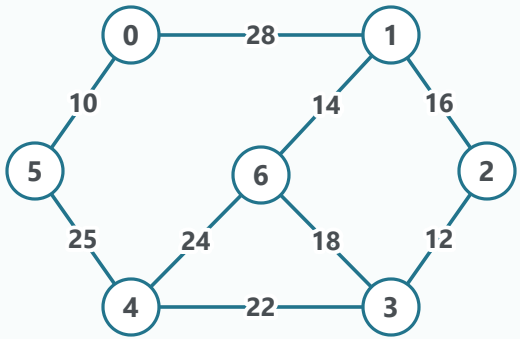


Deleting any tree edge, e.g. (0,2), **splits** the tree: {2, 5, 6} breaks off. A spanning tree is a minimal connected subgraph.

- Exactly **n − 1** edges, no cycle, connected: any two of these imply the third.
- There is a **unique** path between any two vertices.
- Adding a nontree edge creates a single cycle; removing any edge of that cycle gives a new spanning tree (the generation method relies on this).
- Uses: dropping redundant links in a network, *broadcast* trees, circuit equations (Kirchhoff).

Minimum Cost Spanning Tree (MST)

- The edges of an undirected graph can be **weighted**: **G(V, E, w)** instead of G(V, E).
- The spanning tree(s) with the **least** total edge cost: the minimum spanning tree.
- **Ex. a:** edge costs are the costs of links between computers.
- **Ex. b:** power or fiber for 7 villages: a feasibility study gives the km of cable between village pairs. Pairs with no link cost ∞ (not shown).



Sample graph

Instructor: "When we pick edges for a minimum spanning tree we avoid cycles… 'If the power doesn't come from Lower Tepecik, let it come from Upper Tepecik.' No way!"

Algorithms and the Greedy Approach

Algorithms

Kruskal, Prim, Sollin (Borůvka), and also *reverse-delete*. This chapter covers the first two.

Design methods

Backtracking, *greedy*, dynamic programming…

Greedy = stubborn

Each step makes the best choice for the moment and **never revises it**; there is no going back.

Three rules

1) Never take an edge that forms a cycle. 2) Prefer the cheapest edge. 3) Choose exactly **n − 1** edges for n vertices.

Why is it correct? (extra) *Cut property:* for any cut that splits the vertices in two, the cheapest edge crossing the cut is in some MST. Kruskal and Prim pick such an edge at every step, so the greedy choice is never regretted.

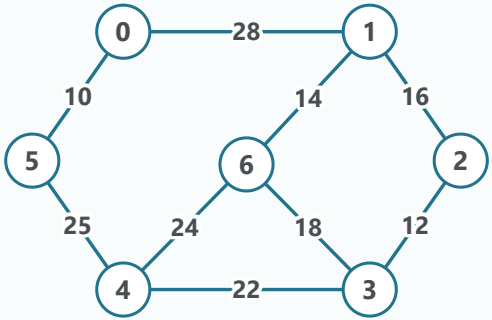
"Beauty is the purgation of superfluities." — Michelangelo

Kruskal's Algorithm (1956)

```
T = {};                    /* chosen edges */
while (|T| < n-1 && E ≠ ∅) {
    choose a least-cost edge (v,w) from E;
    E = E - {(v,w)};
    if ((v,w) does not create a cycle in T)
        T = T ∪ {(v,w)};
    else
        discard (v,w);
}
if (|T| < n-1)
    printf("no spanning tree\n");
```

- T: chosen edges, E: remaining edges, |T|: number of chosen edges.
- The loop runs until n − 1 edges are chosen; it also stops if no edges are left before that → the graph is not connected.
- In Kruskal the chosen edges may form **separate pieces** (a forest); at the end there is a single tree.
- Joseph B. Kruskal, 1956; its correctness is proved. “Reading the proof makes it clearer.”

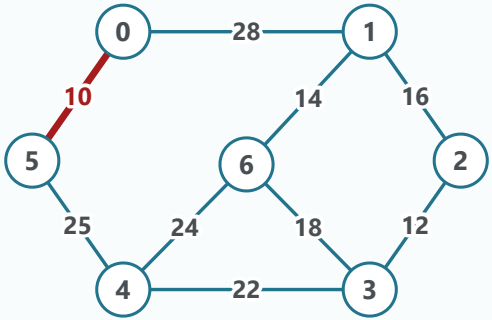
Kruskal Step by Step



Edge	w	Action
(0,5)	10	
(2,3)	12	
(1,6)	14	
(1,2)	16	
(3,6)	18	
(3,4)	22	
(4,6)	24	
(4,5)	25	
(0,1)	28	

The edges are sorted by cost; every vertex is a piece of its own.

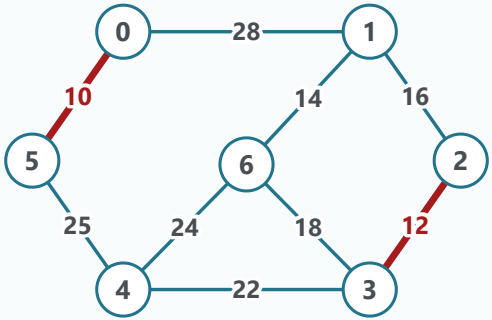
Kruskal Step by Step



Edge	w	Action
(0,5)	10	Add to tree
(2,3)	12	
(1,6)	14	
(1,2)	16	
(3,6)	18	
(3,4)	22	
(4,6)	24	
(4,5)	25	
(0,1)	28	

(0,5) = 10: ends in different pieces → **added**. Total 10.

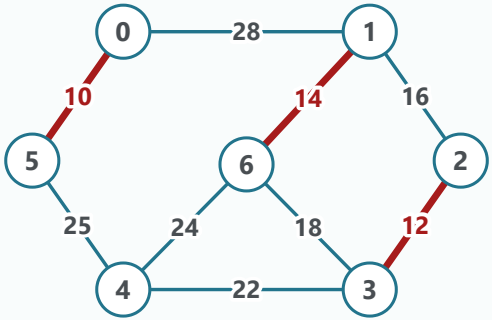
Kruskal Step by Step



Edge	w	Action
(0,5)	10	Add to tree
(2,3)	12	Add to tree
(1,6)	14	
(1,2)	16	
(3,6)	18	
(3,4)	22	
(4,6)	24	
(4,5)	25	
(0,1)	28	

(2,3) = 12: ends in different pieces → **added**. Total 22.

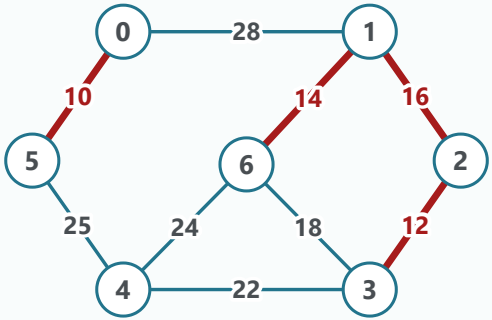
Kruskal Step by Step



Edge	w	Action
(0,5)	10	Add to tree
(2,3)	12	Add to tree
(1,6)	14	Add to tree
(1,2)	16	
(3,6)	18	
(3,4)	22	
(4,6)	24	
(4,5)	25	
(0,1)	28	

(1,6) = 14: ends in different pieces → **added**. Total 36.

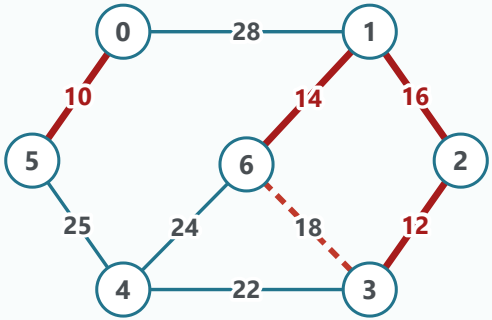
Kruskal Step by Step



Edge	w	Action
(0,5)	10	Add to tree
(2,3)	12	Add to tree
(1,6)	14	Add to tree
(1,2)	16	Add to tree
(3,6)	18	
(3,4)	22	
(4,6)	24	
(4,5)	25	
(0,1)	28	

(1,2) = 16: ends in different pieces → **added**. Total 52.

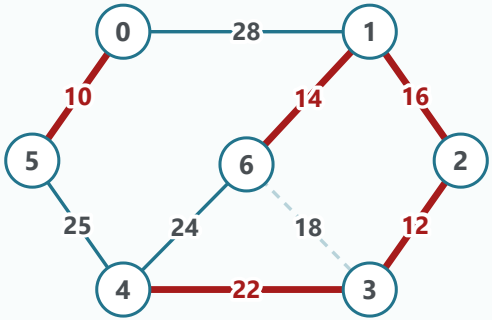
Kruskal Step by Step



Edge	w	Action
(0,5)	10	Add to tree
(2,3)	12	Add to tree
(1,6)	14	Add to tree
(1,2)	16	Add to tree
(3,6)	18	Cycle, skip
(3,4)	22	
(4,6)	24	
(4,5)	25	
(0,1)	28	

(3,6) = 18: both ends in **the same piece** → it would form a cycle; discarded.

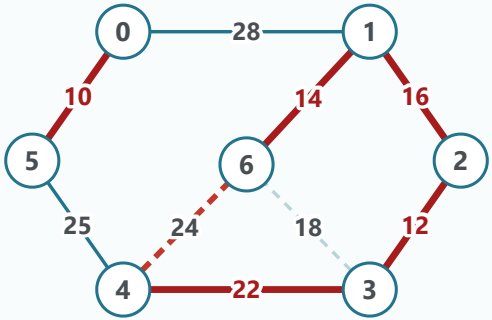
Kruskal Step by Step



Edge	w	Action
(0,5)	10	Add to tree
(2,3)	12	Add to tree
(1,6)	14	Add to tree
(1,2)	16	Add to tree
(3,6)	18	Cycle, skip
(3,4)	22	Add to tree
(4,6)	24	
(4,5)	25	
(0,1)	28	

(3,4) = 22: ends in different pieces → **added**. Total 74.

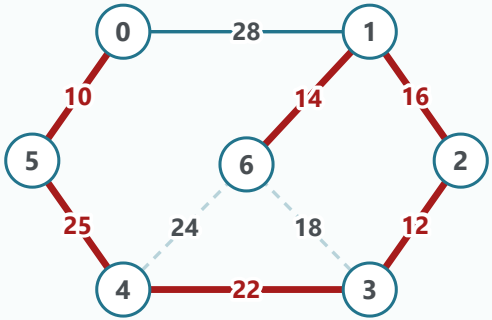
Kruskal Step by Step



Edge	w	Action
(0,5)	10	Add to tree
(2,3)	12	Add to tree
(1,6)	14	Add to tree
(1,2)	16	Add to tree
(3,6)	18	Cycle, skip
(3,4)	22	Add to tree
(4,6)	24	Cycle, skip
(4,5)	25	
(0,1)	28	

(4,6) = 24: both ends in **the same piece** → it would form a cycle; discarded.

Kruskal Step by Step



Edge	w	Action
(0,5)	10	Add to tree
(2,3)	12	Add to tree
(1,6)	14	Add to tree
(1,2)	16	Add to tree
(3,6)	18	Cycle, skip
(3,4)	22	Add to tree
(4,6)	24	Cycle, skip
(4,5)	25	Add to tree
(0,1)	28	

(4,5) = 25: ends in different pieces → **added**. Total 99. n − 1 = 6 edges chosen: **done** (28 is never examined).

How Not to Code Kruskal

- Observation: first there were 2 components, then 3, then back to 2. “Got it! I'll keep the chosen edges in adjacency lists.”
- The unchosen edges sit in an array. **Bad:** each time, search for the minimum, delete it, shift the rest. **Better:** keep the edges sorted by cost (end1, end2, cost).
- (0,5) arrives: search the lists, neither 0 nor 5 is there → new list. (3,2) and (6,1) also start new lists. (1,2): in different lists → no cycle, merge the two lists.
- (3,6): both in the same list → cycle! It works, but it pays **a search cost every time**.

Instructor: “We detected the cycle, but I am describing the wrong algorithm. You don't code it like this, or you won't get the points. I say ‘don't code it like this’, yet it works correctly; that's what I'm trying to show.”

Instructor: “I may give coding Kruskal's algorithm as homework. Let's see whether you can discover the data structures it needs. I'm curious about that.”

KRUSKAL

How to Code Kruskal: Two Questions, Two Data Structures

1) Which edge is cheapest?

A **min priority queue** (min heap). Inserting edges one by one is O(e log e); turning the array into a heap bottom-up is O(e). Or sort the edges once (`qsort`).

2) Does it form a cycle?

No cycle if the ends belong to **two different sets**: **union-find**. Do `find(3)` and `find(2)` return the same root?

> **Instructor:** “So the instructor had taught us these data structures all along and we never noticed!! And there we were, keeping lists… Look how useful the sets we learned in primary school turn out to be.”

Cost: $|E| \cdot \lg|E| + |V| \cdot (\lg|E| + \Theta(1))$; with a bottom-up heap, $2|E| + |V| \cdot \lg|E|$ ($\Theta(1)$: inverse Ackermann, $\alpha \leq \lg|E|$). In the worst case the cycle-forming edges may force e *deleteMin*s rather than n − 1; a safe upper bound is **O(e log e) = O(e log n)**.

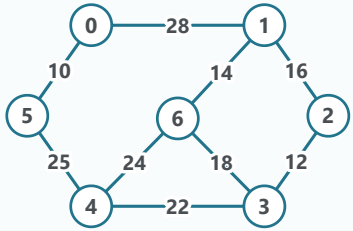
C: Sets (Union-Find) and the Comparator

```c
typedef struct { int u, v, w; } edge;

int find(int parent[], int i)    /* root: < 0 */
{
    while (parent[i] >= 0) i = parent[i];
    return i;
}
void unite(int parent[], int a, int b)   /* roots */
{
    if (parent[a] <= parent[b]) {  /* a is larger */
        parent[a] += parent[b];  parent[b] = a;
    } else {
        parent[b] += parent[a];  parent[a] = b;
    }
}
int cmp(const void *x, const void *y)
{
    return ((const edge *)x)->w - ((const edge *)y)->w;
}
```

- `parent[i] < 0` : i is a root and `-parent[i]` is the set size (initially all −1).
- **Weighted union:** the smaller set goes under the larger → tree height O(log n), `find` O(log n).
- Add *path compression* and the amortized cost per operation is ≈ O(α(n)), constant in practice.
- `cmp` : compares costs for `qsort` (small integers, so no overflow).

Code Trace: The `kruskal()` Main Loop



parent

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

edges sorted

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

Every vertex is the root of its own set (parent = −1). The edges are sorted by cost: O(e log e).

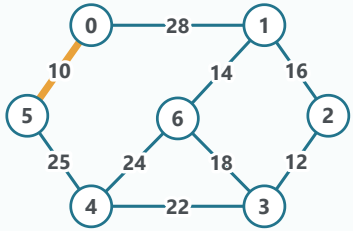
Step 1 / 9

```
▶ for (i = 0; i < n; i++) parent[i] = -1;
▶ qsort(edges, e, sizeof(edge), cmp);
3 for (i = 0, t = 0; i < e && t < n - 1; i++) {
4     int a = find(parent, edges[i].u);
5     int b = find(parent, edges[i].v);
6     if (a != b) {                  /* no cycle */
7         T[t++] = edges[i];
8         unite(parent, a, b);
9     }                              /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`n = 7` `e = 9` `t = 0`

Code Trace: The `kruskal()` Main Loop



parent

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

edges sorted

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

i = 0: (0,5) = 10. `find` : 0 → root 0, 5 → root 5.

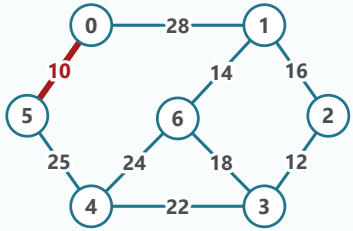
Step 2 / 9

```
1 for (i = 0; i < n; i++) parent[i] = -1;
2 qsort(edges, e, sizeof(edge), cmp);
▶ for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶     int a = find(parent, edges[i].u);
▶     int b = find(parent, edges[i].v);
6     if (a != b) {                    /* no cycle */
7         T[t++] = edges[i];
8         unite(parent, a, b);
9     }                                /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`i = 0` `t = 0` `edges[i] = (0,5,10)` `a = 0` `b = 5`

Code Trace: The `kruskal()` Main Loop



parent

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

edges sorted

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

0 ≠ 5 → no cycle: added to T and the sets are united (the larger set's root stays). t = 1.

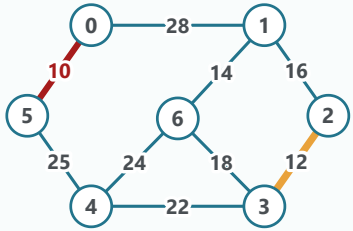
Step 3 / 9

```
1 for (i = 0; i < n; i++) parent[i] = -1;
2 qsort(edges, e, sizeof(edge), cmp);
3 for (i = 0, t = 0; i < e && t < n - 1; i++) {
4     int a = find(parent, edges[i].u);
5     int b = find(parent, edges[i].v);
▶     if (a != b) {                    /* no cycle */
▶         T[t++] = edges[i];
▶         unite(parent, a, b);
9     }                                /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`i = 0` `t = 1` `edges[i] = (0,5,10)` `a = 0` `b = 5`

Code Trace: The `kruskal()` Main Loop



parent

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

edges sorted

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

i = 1: (2,3) = 12. `find` : 2 → root 2, 3 → root 3.

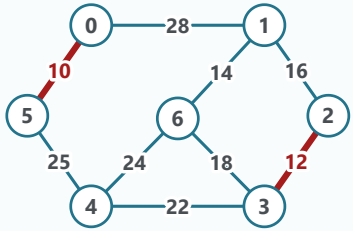
Step 4 / 9

```
1  for (i = 0; i < n; i++) parent[i] = -1;
2  qsort(edges, e, sizeof(edge), cmp);
▶  for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶      int a = find(parent, edges[i].u);
▶      int b = find(parent, edges[i].v);
6      if (a != b) {                    /* no cycle */
7          T[t++] = edges[i];
8          unite(parent, a, b);
9      }                                /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`i = 1` `t = 1` `edges[i] = (2,3,12)` `a = 2` `b = 3`

Code Trace: The `kruskal()` Main Loop



parent

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

edges sorted

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

2 ≠ 3 → no cycle: added to T and the sets are united (the larger set's root stays). t = 2.

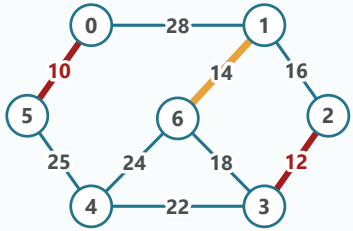
Step 5 / 9

```
1  for (i = 0; i < n; i++) parent[i] = -1;
2  qsort(edges, e, sizeof(edge), cmp);
3  for (i = 0, t = 0; i < e && t < n - 1; i++) {
4      int a = find(parent, edges[i].u);
5      int b = find(parent, edges[i].v);
▶      if (a != b) {                    /* no cycle */
▶          T[t++] = edges[i];
▶          unite(parent, a, b);
9      }                                /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`i = 1` `t = 2` `edges[i] = (2,3,12)` `a = 2` `b = 3`

Code Trace: The `kruskal()` Main Loop



parent

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

edges sorted

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

i = 2: (1,6) = 14. `find` : 1 → root 1, 6 → root 6.

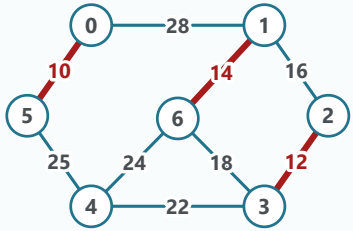
Step 6 / 9

```
1 for (i = 0; i < n; i++) parent[i] = -1;
2 qsort(edges, e, sizeof(edge), cmp);
▶ for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶     int a = find(parent, edges[i].u);
▶     int b = find(parent, edges[i].v);
6     if (a != b) {                    /* no cycle */
7         T[t++] = edges[i];
8         unite(parent, a, b);
9     }                                /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`i = 2` `t = 2` `edges[i] = (1,6,14)` `a = 1` `b = 6`

Code Trace: The `kruskal()` Main Loop



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

	0	1	2	3	4	5	6	7	8
edges sorted	0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28
			▲ i						

1 ≠ 6 → no cycle: added to T and the sets are united (the larger set's root stays). t = 3.

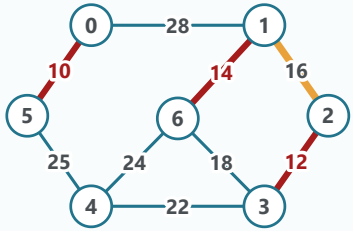
Step 7 / 9

```
1 for (i = 0; i < n; i++) parent[i] = -1;
2 qsort(edges, e, sizeof(edge), cmp);
3 for (i = 0, t = 0; i < e && t < n - 1; i++) {
4     int a = find(parent, edges[i].u);
5     int b = find(parent, edges[i].v);
▶     if (a != b) {                    /* no cycle */
▶         T[t++] = edges[i];
▶         unite(parent, a, b);
9     }                                /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`i = 2` `t = 3` `edges[i] = (1,6,14)` `a = 1` `b = 6`

Code Trace: The `kruskal()` Main Loop



parent

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

edges sorted

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

i = 3: (1,2) = 16. `find` : 1 → root 1, 2 → root 2.

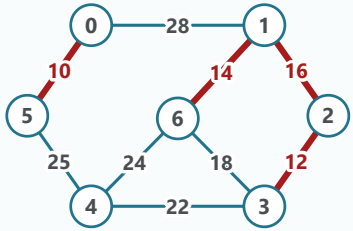
Step 8 / 9

```
1  for (i = 0; i < n; i++) parent[i] = -1;
2  qsort(edges, e, sizeof(edge), cmp);
▶  for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶      int a = find(parent, edges[i].u);
▶      int b = find(parent, edges[i].v);
6      if (a != b) {                    /* no cycle */
7          T[t++] = edges[i];
8          unite(parent, a, b);
9      }                                /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`i = 3` `t = 3` `edges[i] = (1,2,16)` `a = 1` `b = 2`

Code Trace: The `kruskal()` Main Loop



parent

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

edges sorted

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

1 ≠ 2 → no cycle: added to T and the sets are united (the larger set's root stays). t = 4.

Step 9 / 9

```
1  for (i = 0; i < n; i++) parent[i] = -1;
2  qsort(edges, e, sizeof(edge), cmp);
3  for (i = 0, t = 0; i < e && t < n - 1; i++) {
4      int a = find(parent, edges[i].u);
5      int b = find(parent, edges[i].v);
▶      if (a != b) {                    /* no cycle */
▶          T[t++] = edges[i];
▶          unite(parent, a, b);
9      }                                /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

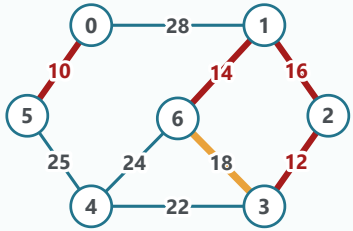
VARIABLES

`i = 3` `t = 4`

`edges[i] = (1,2,16)` `a = 1`

`b = 2`

Code Trace: `kruskal()` (Continued)



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

	0	1	2	3	4	5	6	7	8
edges	0-5	2-3	1-6	1-2	3-6	3-4	4-6	4-5	0-1
sorted	10	12	14	16	18	22	24	25	28
					▲ i				

i = 4: (3,6) = 18. `find` : 3 → root 1, 6 → root 1.

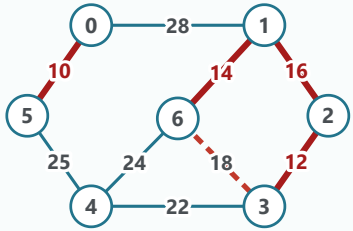
Step 1 / 9

```
1  for (i = 0; i < n; i++) parent[i] = -1;
2  qsort(edges, e, sizeof(edge), cmp);
▶  for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶      int a = find(parent, edges[i].u);
▶      int b = find(parent, edges[i].v);
6      if (a != b) {                    /* no cycle */
7          T[t++] = edges[i];
8          unite(parent, a, b);
9      }                                /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`i = 4` `t = 4` `edges[i] = (3,6,18)` `a = 1` `b = 1`

Code Trace: `kruskal()` (Continued)



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

	0	1	2	3	4	5	6	7	8
edges	0-5	2-3	1-6	1-2	3-6	3-4	4-6	4-5	0-1
sorted	10	12	14	16	18	22	24	25	28
					▲ i				

a = b = 1: both ends are in the same set → the edge **forms a cycle**; discarded.

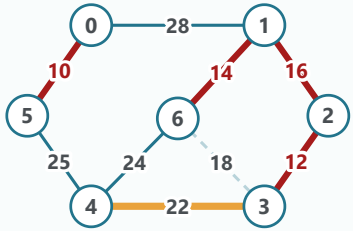
Step 2 / 9

```
1 for (i = 0; i < n; i++) parent[i] = -1;
2 qsort(edges, e, sizeof(edge), cmp);
3 for (i = 0, t = 0; i < e && t < n - 1; i++) {
4     int a = find(parent, edges[i].u);
5     int b = find(parent, edges[i].v);
▶     if (a != b) {                    /* no cycle */
7         T[t++] = edges[i];
8         unite(parent, a, b);
9     }                                /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`i = 4` `t = 4` `edges[i] = (3,6,18)` `a = 1` `b = 1`

Code Trace: `kruskal()` (Continued)



parent

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

edges sorted

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

i = 5: (3,4) = 22. `find` : 3 → root 1, 4 → root 4.

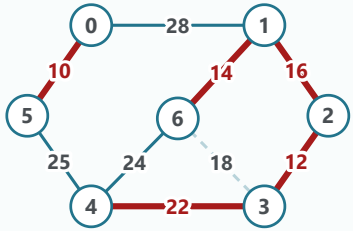
Step 3 / 9

```
1  for (i = 0; i < n; i++) parent[i] = -1;
2  qsort(edges, e, sizeof(edge), cmp);
▶  for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶      int a = find(parent, edges[i].u);
▶      int b = find(parent, edges[i].v);
6      if (a != b) {                    /* no cycle */
7          T[t++] = edges[i];
8          unite(parent, a, b);
9      }                                /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`i = 5` `t = 4` `edges[i] = (3,4,22)` `a = 1` `b = 4`

Code Trace: `kruskal()` (Continued)



parent

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

edges sorted

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

1 ≠ 4 → no cycle: added to T and the sets are united (the larger set's root stays). t = 5.

Step 4 / 9

```
1  for (i = 0; i < n; i++) parent[i] = -1;
2  qsort(edges, e, sizeof(edge), cmp);
3  for (i = 0, t = 0; i < e && t < n - 1; i++) {
4      int a = find(parent, edges[i].u);
5      int b = find(parent, edges[i].v);
▶      if (a != b) {                    /* no cycle */
▶          T[t++] = edges[i];
▶          unite(parent, a, b);
9      }                                /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

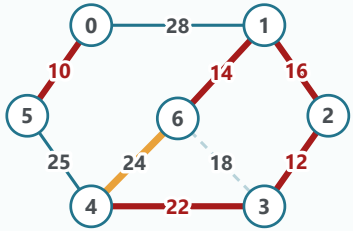
VARIABLES

`i = 5` `t = 5`

`edges[i] = (3,4,22)` `a = 1`

`b = 4`

Code Trace: `kruskal()` (Continued)



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

	0	1	2	3	4	5	6	7	8
edges sorted	0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28
							▲ i		

i = 6: (4,6) = 24. `find` : 4 → root 1, 6 → root 1.

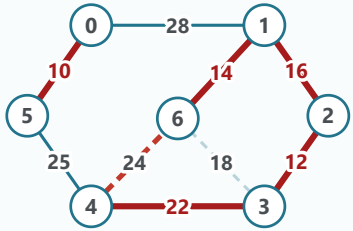
Step 5 / 9

```
1  for (i = 0; i < n; i++) parent[i] = -1;
2  qsort(edges, e, sizeof(edge), cmp);
▶  for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶      int a = find(parent, edges[i].u);
▶      int b = find(parent, edges[i].v);
6      if (a != b) {                    /* no cycle */
7          T[t++] = edges[i];
8          unite(parent, a, b);
9      }                                /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`i = 6` `t = 5` `edges[i] = (4,6,24)` `a = 1` `b = 1`

Code Trace: `kruskal()` (Continued)



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

	0	1	2	3	4	5	6	7	8
edges sorted	0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28
							▲ i		

a = b = 1: both ends are in the same set → the edge **forms a cycle**; discarded.

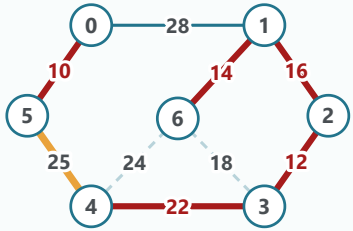
Step 6 / 9

```
1 for (i = 0; i < n; i++) parent[i] = -1;
2 qsort(edges, e, sizeof(edge), cmp);
3 for (i = 0, t = 0; i < e && t < n - 1; i++) {
4     int a = find(parent, edges[i].u);
5     int b = find(parent, edges[i].v);
▶     if (a != b) {                  /* no cycle */
7         T[t++] = edges[i];
8         unite(parent, a, b);
9     }                              /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`i = 6` `t = 5` `edges[i] = (4,6,24)` `a = 1` `b = 1`

Code Trace: `kruskal()` (Continued)



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

	0	1	2	3	4	5	6	7	8
edges sorted	0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28
								▲ i	

i = 7: (4,5) = 25. `find` : 4 → root 1, 5 → root 0.

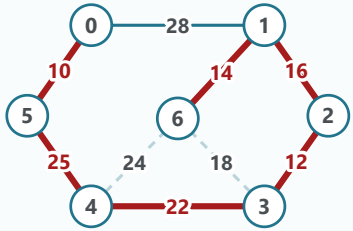
Step 7 / 9

```
1  for (i = 0; i < n; i++) parent[i] = -1;
2  qsort(edges, e, sizeof(edge), cmp);
▶  for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶      int a = find(parent, edges[i].u);
▶      int b = find(parent, edges[i].v);
6      if (a != b) {                    /* no cycle */
7          T[t++] = edges[i];
8          unite(parent, a, b);
9      }                                /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`i = 7` `t = 5` `edges[i] = (4,5,25)` `a = 1` `b = 0`

Code Trace: `kruskal()` (Continued)



parent

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

edges sorted

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28
							▲ i	

1 ≠ 0 → no cycle: added to T and the sets are united (the larger set's root stays). t = 6.

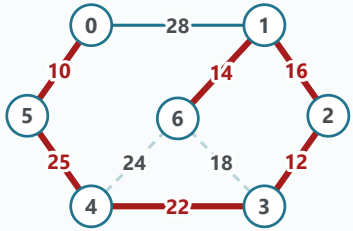
Step 8 / 9

```
1 for (i = 0; i < n; i++) parent[i] = -1;
2 qsort(edges, e, sizeof(edge), cmp);
3 for (i = 0, t = 0; i < e && t < n - 1; i++) {
4     int a = find(parent, edges[i].u);
5     int b = find(parent, edges[i].v);
▶     if (a != b) {                  /* no cycle */
▶         T[t++] = edges[i];
▶         unite(parent, a, b);
9     }                              /* else skip */
10 }
11 if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`i = 7` `t = 6` `edges[i] = (4,5,25)` `a = 1` `b = 0`

Code Trace: `kruskal()` (Continued)



parent

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

edges sorted

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

t = n − 1 = 6 → the loop ends; (0,1) = 28 is never examined. MST cost **10 + 12 + 14 + 16 + 22 + 25 = 99**.

Step 9 / 9

```
1  for (i = 0; i < n; i++) parent[i] = -1;
2  qsort(edges, e, sizeof(edge), cmp);
▶  for (i = 0, t = 0; i < e && t < n - 1; i++) {
4      int a = find(parent, edges[i].u);
5      int b = find(parent, edges[i].v);
6      if (a != b) {                  /* no cycle */
7          T[t++] = edges[i];
8          unite(parent, a, b);
9      }                              /* else skip */
10 }
▶  if (t < n - 1) printf("no spanning tree\n");
```

VARIABLES

`i = 8` `t = 6` `cost = 99`

Complexity of Kruskal (with Union-Find)

Step	Cost	Notes
Sort / build heap	O(e log e) / O(e)	qsort or bottom-up heap construction
Take the cheapest edge	O(log e) × at most e times	min heap deleteMin (O(1) in a sorted array)
find (2 per edge)	O(log n) weighted; ≈ O(α(n)) with path compression	at most 2e calls
unite	O(1)	exactly n − 1 times
Total	**O(e log e) = O(e log n)**	$e \leq n^2 \rightarrow \log e \leq 2 \log n$

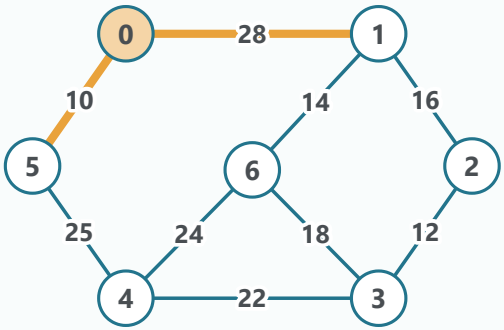
The “bad” version (search and merge lists) spends an O(n) search per edge: O(e · n). The choice of data structure moves the same algorithm to a completely different cost.

Prim's Algorithm

```
T  = {};
TV = {0};           /* tree vertices: start at 0 */
while (|T| < n-1) {
    pick a least-cost (u,v) with u ∈ TV, v ∉ TV;
    if (there is no such edge) break;
    TV = TV ∪ {v};
    T  = T  ∪ {(u,v)};
}
if (|T| < n-1)
    printf("no spanning tree\n");
```

- Jarník (1930), Prim (1957), Dijkstra (1959) → the DJP or Prim–Jarník algorithm.
- Kruskal may choose disconnected edges; Prim grows the chosen edges as **a single tree**.
- Candidate edges: one end in TV, the other end **not** in TV.
- A single tree keeps growing; no cycle check is needed ($v \notin TV$ already guarantees it).

Prim Step by Step

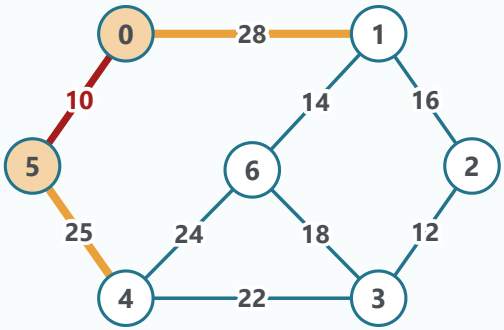


TV	T	Candidates
{0}	{}	(0,1), (0,5)

Candidate edges in yellow. The cheapest is **(0,5) = 10** → 5 joins the tree.

In the last steps (0,1) may look like a candidate, but both its ends are in TV, so it is not.

Prim Step by Step

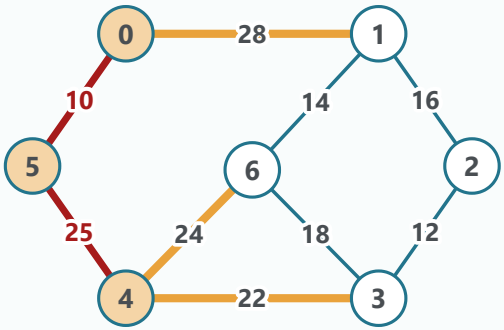


TV	T	Candidates
{0, 5}	{(0,5)}	(0,1), (5,4)

Candidate edges in yellow. The cheapest is **(5,4) = 25** → 4 joins the tree.

In the last steps (0,1) may look like a candidate, but both its ends are in TV, so it is not.

Prim Step by Step



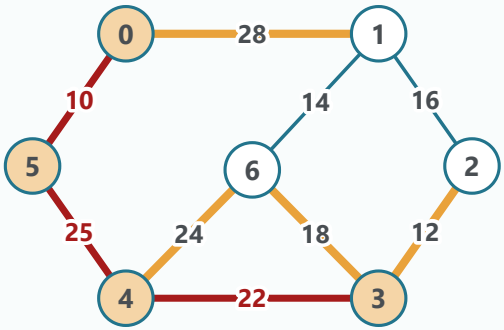
TV	T	Candidates
{0, 5, 4}	{(0,5), (5,4)}	(0,1), (4,3), (4,6)

Candidate edges in yellow. The cheapest is **(4,3) = 22** → 3 joins the tree.

Step 3 / 7

In the last steps (0,1) may look like a candidate, but both its ends are in TV, so it is not.

Prim Step by Step



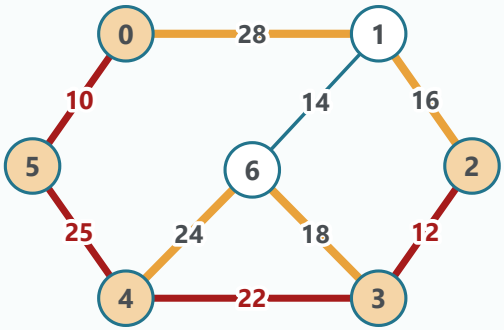
TV	T	Candidates
{0, 5, 4, 3}	{(0,5), (5,4), (4,3)}	(0,1), (3,2), (3,6), (4,6)

Candidate edges in yellow. The cheapest is **(3,2) = 12** → 2 joins the tree.

Step 4 / 7

In the last steps (0,1) may look like a candidate, but both its ends are in TV, so it is not.

Prim Step by Step



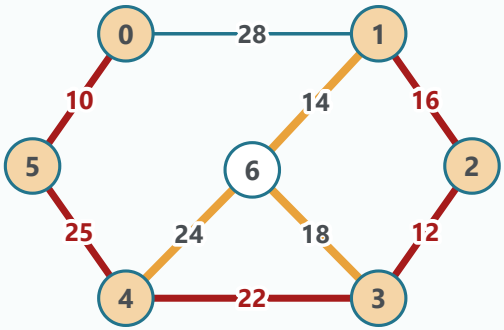
TV	T	Candidates
{0, 5, 4, 3, 2}	{(0,5), (5,4), (4,3), (3,2)}	(0,1), (2,1), (3,6), (4,6)

Candidate edges in yellow. The cheapest is **(2,1) = 16** → 1 joins the tree.

Step 5 / 7

In the last steps (0,1) may look like a candidate, but both its ends are in TV, so it is not.

Prim Step by Step



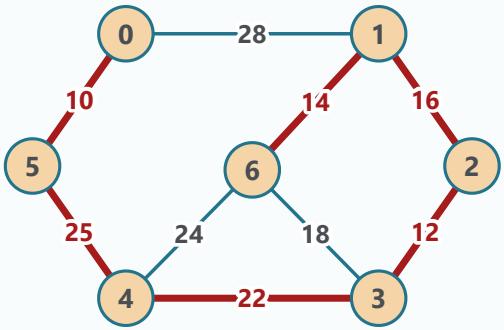
TV	T	Candidates
{0, 5, 4, 3, 2, 1}	{(0,5), (5,4), (4,3), (3,2), (2,1)}	(1,6), (3,6), (4,6)

Candidate edges in yellow. The cheapest is **(1,6) = 14** → 6 joins the tree.

Step 6 / 7

In the last steps (0,1) may look like a candidate, but both its ends are in TV, so it is not.

Prim Step by Step



TV	T	Candidates
{0, 5, 4, 3, 2, 1, 6}	{(0,5), (5,4), (4,3), (3,2), (2,1), (1,6)}	—

TV holds every vertex: 6 edges, total **99**, the same tree as Kruskal (the weights are distinct, so the MST is unique).

Step 7 / 7

In the last steps (0,1) may look like a candidate, but both its ends are in TV, so it is not.

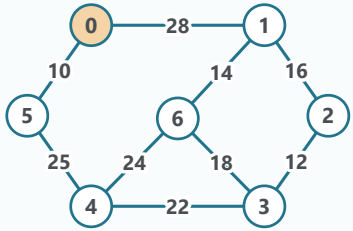
Coding Prim: The Real Difficulty

- The cheapest candidate edge must be found **in one move**.
- When an edge is chosen, new edges **enter** the candidate set and some **leave** it (those with both ends now in TV): set A becomes set B.
- When 0 is chosen, the data structure should see only (0,1) and (0,5) and not even look at the rest.
- Solution (extra): for each v ∉ TV, `lowcost[v]` = cheapest link to TV. $O(n^2)$ with arrays; O(e log n) with a min heap.

> **Instructor:** "If you can find the answer to this, data structures have taken you somewhere… Say 'Prim already wrote it, I'll just code it' and the complexity suddenly blows up."

> **Instructor:** "Give this problem to a graph theorist who has never heard the word 'algorithm' and this is how they will solve it. A computer engineer discovers what the data structure behind it must be."

Code Trace: `prim()` (lowcost/closest Arrays)



	0	1	2	3	4	5	6
in_tree	1	0	0	0	0	0	0
lowcost	0	28	∞	∞	∞	10	∞
closest	0	0	0	0	0	0	0

TV = {0}. `lowcost[v]` = w[0][v]: 1 (28) and 5 (10) are adjacent to 0; the rest are ∞. `closest[v]` = 0.

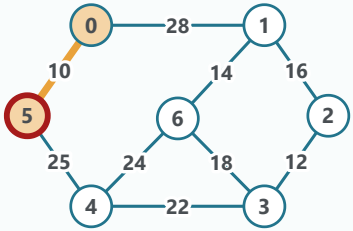
Step 1 / 13

```
▶ in_tree[0] = 1;                    /* TV = {0} */
2 for (v = 0; v < n; v++)
▶     lowcost[v] = w[0][v], closest[v] = 0;
4 for (k = 1; k < n; k++) {
5     u = cheapest(lowcost, in_tree, n); /* O(n) */
6     if (u < 0) break;              /* no tree */
7     in_tree[u] = 1;  total += lowcost[u];
8     for (v = 0; v < n; v++)
9         if (!in_tree[v] && w[u][v] < lowcost[v])
10            lowcost[v] = w[u][v], closest[v] = u;
11 }
```

VARIABLES

k = – u = – total = 0

Code Trace: `prim()` (lowcost/closest Arrays)



	0	1	2	3	4	5	6
in_tree	1	0	0	0	0	1	0
lowcost	0	28	∞	∞	∞	10	∞
closest	0	0	0	0	0	0	0

Smallest `lowcost` : **u = 5** (10); edge (0,5) enters the tree. total = 10.

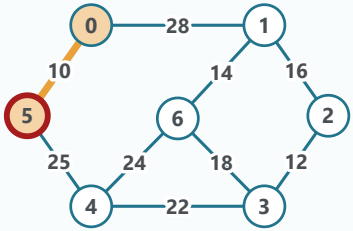
Step 2 / 13

```
 1 in_tree[0] = 1;                    /* TV = {0} */
 2 for (v = 0; v < n; v++)
 3     lowcost[v] = w[0][v], closest[v] = 0;
 4 for (k = 1; k < n; k++) {
 ▶     u = cheapest(lowcost, in_tree, n); /* O(n) */
 6     if (u < 0) break;              /* no tree */
 ▶     in_tree[u] = 1;  total += lowcost[u];
 8     for (v = 0; v < n; v++)
 9         if (!in_tree[v] && w[u][v] < lowcost[v])
10             lowcost[v] = w[u][v], closest[v] = u;
11 }
```

VARIABLES

`k = 1` `u = 5` `total = 10`

Code Trace: `prim()` (lowcost/closest Arrays)



	0	1	2	3	4	5	6
in_tree	1	0	0	0	0	1	0
lowcost	0	28	∞	∞	25	10	∞
closest	0	0	0	0	5	0	0

Cheaper links through 5: lowcost[4] = 25 (closest = 5).

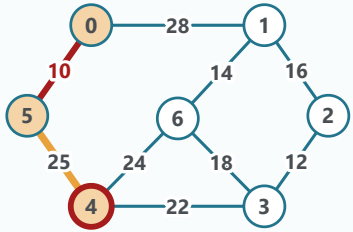
Step 3 / 13

```
1  in_tree[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      lowcost[v] = w[0][v], closest[v] = 0;
4  for (k = 1; k < n; k++) {
5      u = cheapest(lowcost, in_tree, n); /* O(n) */
6      if (u < 0) break;              /* no tree */
7      in_tree[u] = 1;  total += lowcost[u];
8      for (v = 0; v < n; v++)
▶          if (!in_tree[v] && w[u][v] < lowcost[v])
▶              lowcost[v] = w[u][v], closest[v] = u;
11 }
```

VARIABLES

`k = 1` `u = 5` `total = 10`

Code Trace: `prim()` (lowcost/closest Arrays)



	0	1	2	3	4	5	6
in_tree	1	0	0	0	1	1	0
lowcost	0	28	∞	∞	25	10	∞
closest	0	0	0	0	5	0	0

Smallest `lowcost` : **u = 4** (25); edge (5,4) enters the tree. total = 35.

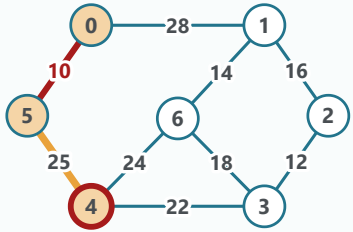
Step 4 / 13

```
 1 in_tree[0] = 1;                    /* TV = {0} */
 2 for (v = 0; v < n; v++)
 3     lowcost[v] = w[0][v], closest[v] = 0;
 4 for (k = 1; k < n; k++) {
 ▶     u = cheapest(lowcost, in_tree, n); /* O(n) */
 6     if (u < 0) break;              /* no tree */
 ▶     in_tree[u] = 1;  total += lowcost[u];
 8     for (v = 0; v < n; v++)
 9         if (!in_tree[v] && w[u][v] < lowcost[v])
10             lowcost[v] = w[u][v], closest[v] = u;
11 }
```

VARIABLES

`k = 2` `u = 4` `total = 35`

Code Trace: `prim()` (lowcost/closest Arrays)



	0	1	2	3	4	5	6
in_tree	1	0	0	0	1	1	0
lowcost	0	28	∞	22	25	10	24
closest	0	0	0	4	5	0	4

Cheaper links through 4: lowcost[3] = 22, lowcost[6] = 24 (closest = 4).

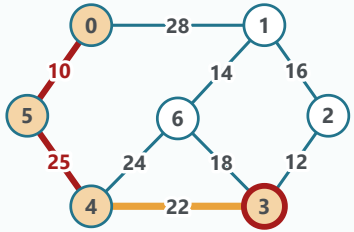
Step 5 / 13

```
1  in_tree[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      lowcost[v] = w[0][v], closest[v] = 0;
4  for (k = 1; k < n; k++) {
5      u = cheapest(lowcost, in_tree, n); /* O(n) */
6      if (u < 0) break;              /* no tree */
7      in_tree[u] = 1;  total += lowcost[u];
8      for (v = 0; v < n; v++)
▶          if (!in_tree[v] && w[u][v] < lowcost[v])
▶              lowcost[v] = w[u][v], closest[v] = u;
11 }
```

VARIABLES

`k = 2` `u = 4` `total = 35`

Code Trace: `prim()` (lowcost/closest Arrays)



	0	1	2	3	4	5	6
in_tree	1	0	0	1	1	1	0
lowcost	0	28	∞	22	25	10	24
closest	0	0	0	4	5	0	4

Smallest `lowcost` : **u = 3** (22); edge (4,3) enters the tree. total = 57.

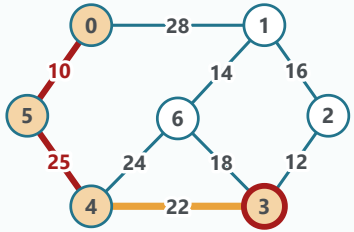
Step 6 / 13

```
1  in_tree[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      lowcost[v] = w[0][v], closest[v] = 0;
4  for (k = 1; k < n; k++) {
▶      u = cheapest(lowcost, in_tree, n); /* O(n) */
6      if (u < 0) break;              /* no tree */
▶      in_tree[u] = 1;  total += lowcost[u];
8      for (v = 0; v < n; v++)
9          if (!in_tree[v] && w[u][v] < lowcost[v])
10             lowcost[v] = w[u][v], closest[v] = u;
11 }
```

VARIABLES

`k = 3` `u = 3` `total = 57`

Code Trace: `prim()` (lowcost/closest Arrays)



	0	1	2	3	4	5	6
in_tree	1	0	0	1	1	1	0
lowcost	0	28	12	22	25	10	18
closest	0	0	3	4	5	0	3

Cheaper links through 3: lowcost[2] = 12, lowcost[6] = 18 (closest = 3).

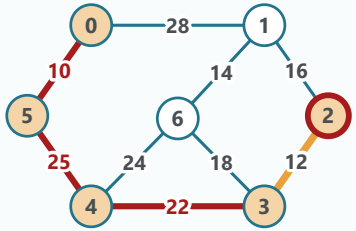
Step 7 / 13

```
1  in_tree[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      lowcost[v] = w[0][v], closest[v] = 0;
4  for (k = 1; k < n; k++) {
5      u = cheapest(lowcost, in_tree, n); /* O(n) */
6      if (u < 0) break;              /* no tree */
7      in_tree[u] = 1;  total += lowcost[u];
8      for (v = 0; v < n; v++)
▶          if (!in_tree[v] && w[u][v] < lowcost[v])
▶              lowcost[v] = w[u][v], closest[v] = u;
11 }
```

VARIABLES

`k = 3` `u = 3` `total = 57`

Code Trace: `prim()` (lowcost/closest Arrays)



	0	1	2	3	4	5	6
in_tree	1	0	1	1	1	1	0
lowcost	0	28	12	22	25	10	18
closest	0	0	3	4	5	0	3

Smallest `lowcost` : **u = 2** (12); edge (3,2) enters the tree. total = 69.

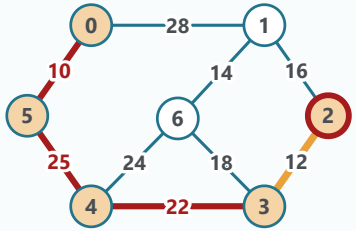
Step 8 / 13

```
1  in_tree[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      lowcost[v] = w[0][v], closest[v] = 0;
4  for (k = 1; k < n; k++) {
▶      u = cheapest(lowcost, in_tree, n); /* O(n) */
6      if (u < 0) break;              /* no tree */
▶      in_tree[u] = 1;  total += lowcost[u];
8      for (v = 0; v < n; v++)
9          if (!in_tree[v] && w[u][v] < lowcost[v])
10             lowcost[v] = w[u][v], closest[v] = u;
11 }
```

VARIABLES

`k = 4` `u = 2` `total = 69`

Code Trace: `prim()` (lowcost/closest Arrays)



	0	1	2	3	4	5	6
in_tree	1	0	1	1	1	1	0
lowcost	0	16	12	22	25	10	18
closest	0	2	3	4	5	0	3

Cheaper links through 2: lowcost[1] = 16 (closest = 2).

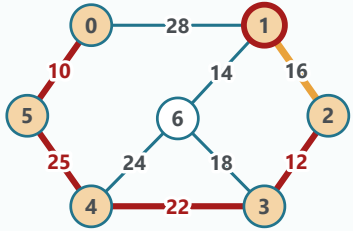
Step 9 / 13

```
1  in_tree[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      lowcost[v] = w[0][v], closest[v] = 0;
4  for (k = 1; k < n; k++) {
5      u = cheapest(lowcost, in_tree, n); /* O(n) */
6      if (u < 0) break;              /* no tree */
7      in_tree[u] = 1;  total += lowcost[u];
8      for (v = 0; v < n; v++)
▶          if (!in_tree[v] && w[u][v] < lowcost[v])
▶              lowcost[v] = w[u][v], closest[v] = u;
11 }
```

VARIABLES

`k = 4` `u = 2` `total = 69`

Code Trace: `prim()` (lowcost/closest Arrays)



	0	1	2	3	4	5	6
in_tree	1	1	1	1	1	1	0
lowcost	0	16	12	22	25	10	18
closest	0	2	3	4	5	0	3

Smallest `lowcost` : **u = 1** (16); edge (2,1) enters the tree. total = 85.

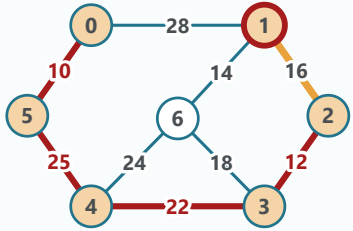
Step 10 / 13

```
1  in_tree[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      lowcost[v] = w[0][v], closest[v] = 0;
4  for (k = 1; k < n; k++) {
▶      u = cheapest(lowcost, in_tree, n); /* O(n) */
6      if (u < 0) break;              /* no tree */
▶      in_tree[u] = 1;  total += lowcost[u];
8      for (v = 0; v < n; v++)
9          if (!in_tree[v] && w[u][v] < lowcost[v])
10             lowcost[v] = w[u][v], closest[v] = u;
11 }
```

VARIABLES

`k = 5` `u = 1` `total = 85`

Code Trace: `prim()` (lowcost/closest Arrays)



	0	1	2	3	4	5	6
in_tree	1	1	1	1	1	1	0
lowcost	0	16	12	22	25	10	14
closest	0	2	3	4	5	0	1

Cheaper links through 1: lowcost[6] = 14 (closest = 1).

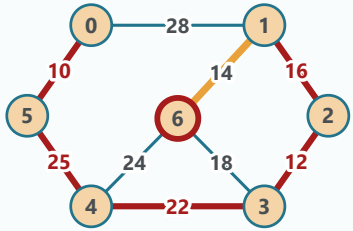
Step 11 / 13

```
1  in_tree[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      lowcost[v] = w[0][v], closest[v] = 0;
4  for (k = 1; k < n; k++) {
5      u = cheapest(lowcost, in_tree, n); /* O(n) */
6      if (u < 0) break;              /* no tree */
7      in_tree[u] = 1;  total += lowcost[u];
8      for (v = 0; v < n; v++)
▶          if (!in_tree[v] && w[u][v] < lowcost[v])
▶              lowcost[v] = w[u][v], closest[v] = u;
11 }
```

VARIABLES

`k = 5` `u = 1` `total = 85`

Code Trace: `prim()` (lowcost/closest Arrays)



	0	1	2	3	4	5	6
in_tree	1	1	1	1	1	1	1
lowcost	0	16	12	22	25	10	14
closest	0	2	3	4	5	0	1

Smallest `lowcost` : **u = 6** (14); edge (1,6) enters the tree. total = 99.

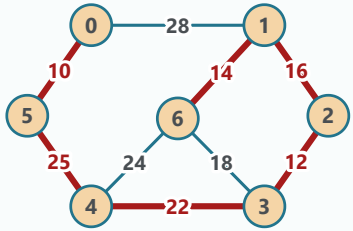
Step 12 / 13

```
1  in_tree[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      lowcost[v] = w[0][v], closest[v] = 0;
4  for (k = 1; k < n; k++) {
▶      u = cheapest(lowcost, in_tree, n); /* O(n) */
6      if (u < 0) break;              /* no tree */
▶      in_tree[u] = 1;  total += lowcost[u];
8      for (v = 0; v < n; v++)
9          if (!in_tree[v] && w[u][v] < lowcost[v])
10             lowcost[v] = w[u][v], closest[v] = u;
11 }
```

VARIABLES

`k = 6` `u = 6` `total = 99`

Code Trace: `prim()` (lowcost/closest Arrays)



	0	1	2	3	4	5	6
in_tree	1	1	1	1	1	1	1
lowcost	0	16	12	22	25	10	14
closest	0	2	3	4	5	0	1

n − 1 vertices added: total **99**. Each round is an O(n) search + O(n) update → **O(n²)**; ideal for dense graphs.

Step 13 / 13

```
1  in_tree[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      lowcost[v] = w[0][v], closest[v] = 0;
▶  for (k = 1; k < n; k++) {
5      u = cheapest(lowcost, in_tree, n); /* O(n) */
6      if (u < 0) break;              /* no tree */
7      in_tree[u] = 1;  total += lowcost[u];
8      for (v = 0; v < n; v++)
9          if (!in_tree[v] && w[u][v] < lowcost[v])
10             lowcost[v] = w[u][v], closest[v] = u;
11 }
```

VARIABLES

`k = 7` `u = −` `total = 99`

Speeding Up Prim with a Heap (Extra Topic)

```
/* adjacency lists + min-heap (key = lowcost) */
heap_push(H, 0, 0);
while (!heap_empty(H)) {
    u = heap_pop_min(H);          /* O(log n) */
    if (in_tree[u]) continue;     /* stale entry */
    in_tree[u] = 1;
    for (p = graph[u]; p; p = p->link)
        if (!in_tree[p->vertex] &&
            p->weight < lowcost[p->vertex]) {
            lowcost[p->vertex] = p->weight;
            closest[p->vertex] = u;
            heap_push(H, p->vertex, p->weight);
        }
}
```

Implementation	Time	Best for
Arrays (above)	$O(n^2)$	dense graphs ($e \approx n^2$)
Binary heap + lists	$O(e \log n)$	sparse graphs
Fibonacci heap	$O(e + n \log n)$	theoretical
Kruskal	$O(e \log e)$	sparse, edge list at hand

Answer to the research question: array-based Prim ($O(n^2)$) wins on dense graphs; Kruskal or heap-based Prim wins on sparse ones.

How to Approach a Problem

> **Instructor:** “How has this problem been solved? Which data structures does the literature propose? How does the world solve it, and how should I?”

> **Instructor:** “‘We have mastered the algorithm.’ No! We are still working on it. Knowing what to use where while solving a problem: that is where data structures come in.”

“*To follow the path: look to the master, follow the master, walk with the master, see through the master, become the master.*”

First learn the master's solution (the literature), then produce your own.

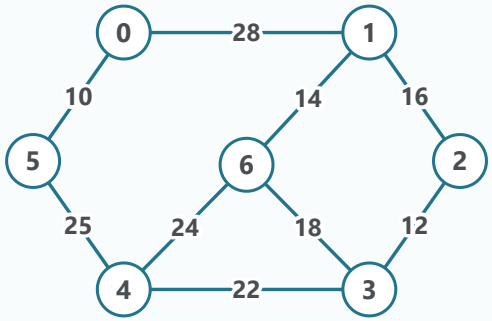
Homework: Code Kruskal with the Right Data Structures

"By next week everyone codes Kruskal's algorithm the way I described it." Expected: **min heap** + **union-find**; input an edge list, output the tree edges and the total cost.

- Edge structure `edge {u, v, w}`; build the heap bottom-up from the array (O(e)).
- Add path compression to `find`; make `unite` weighted.
- If the graph is not connected, print "no spanning tree" (|T| < n − 1).
- On the 7-village example, print output showing the total 99 and the rejection of (3,6) and (4,6).

▶ **Checklist**

Other MST Algorithms: Sollin (Borůvka), Reverse-Delete

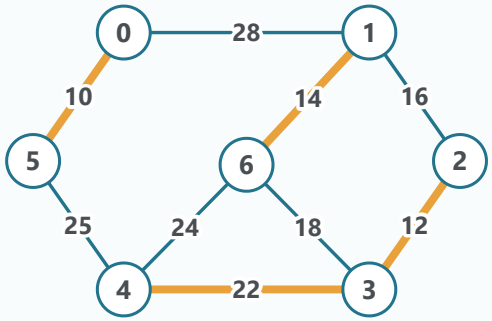


Initially every vertex is a component of its own.

Step 1 / 3

- **Sollin / Borůvka (1926):** in each round *every component* picks its cheapest outgoing edge; the number of components at least halves → at most log n rounds, **O(e log n)**. Well suited to parallelism.
- **Reverse-delete:** try the edges in decreasing cost; delete an edge if doing so keeps the graph connected. The reverse of Kruskal.
- All rely on the same cut/cycle properties and (with distinct weights) find the **same** tree.

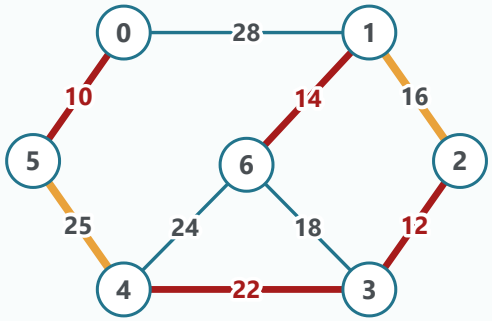
Other MST Algorithms: Sollin (Borůvka), Reverse-Delete



Round 1: each component picks its **cheapest** outgoing edge (yellow): (0,5)=10, (1,6)=14, (2,3)=12, (3,4)=22. Components left: 3.

- **Sollin / Borůvka (1926):** in each round *every component* picks its cheapest outgoing edge; the number of components at least halves → at most log n rounds, **O(e log n)**. Well suited to parallelism.
- **Reverse-delete:** try the edges in decreasing cost; delete an edge if doing so keeps the graph connected. The reverse of Kruskal.
- All rely on the same cut/cycle properties and (with distinct weights) find the **same** tree.

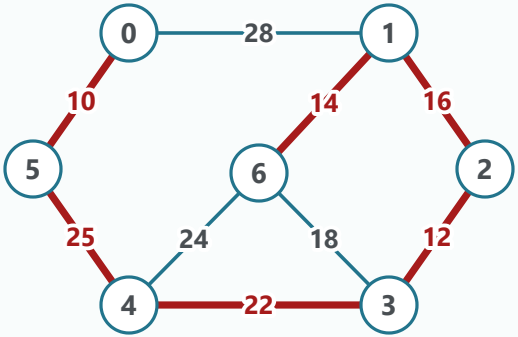
Other MST Algorithms: Sollin (Borůvka), Reverse-Delete



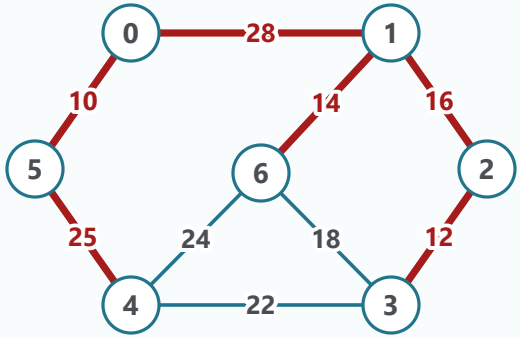
Round 2: each component picks its **cheapest** outgoing edge (yellow): (4,5)=25, (1,2)=16. Components left: 1. Total **99**.

- **Sollin / Borůvka (1926):** in each round *every component* picks its cheapest outgoing edge; the number of components at least halves → at most log n rounds, **O(e log n)**. Well suited to parallelism.
- **Reverse-delete:** try the edges in decreasing cost; delete an edge if doing so keeps the graph connected. The reverse of Kruskal.
- All rely on the same cut/cycle properties and (with distinct weights) find the **same** tree.

An MST Is Not a Shortest-Path Tree



MST (total 99)



shortest-path tree from 0 (total 105)

MST

Minimizes the total edge cost and has no root. Distance from 0 to 1 along the tree: 85.

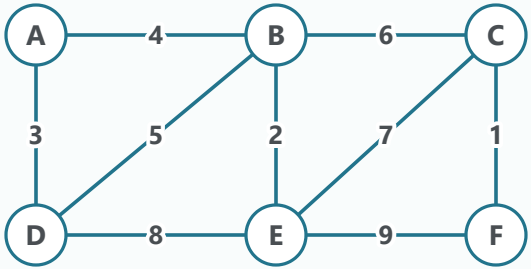
Shortest-path tree

Minimizes each vertex's **distance** from 0: d(1) = 28, but the total edge cost is 105 > 99.

Kruskal and Prim Compared

	Kruskal	**Prim**
What grows	a forest (separate pieces merge)	a single tree (TV)
Selection rule	cheapest of all edges	cheapest edge leaving TV
Cycle check	needs union-find	not needed ($v \notin TV$)
Data structure	sorted array / min heap + sets	lowcost[] array / min heap
Time	O(e log e)	$O(n^2)$ or O(e log n)
Good for	sparse graphs, edge lists	dense graphs, matrices
Disconnected graph	gives a minimum spanning **forest**	spans only 0's component

Exam-Style Question: Run Kruskal and Prim



Coding Kruskal was homework; the question below is the kind that may appear on an exam.

1. Find the MST with Kruskal; for each edge write “add / cycle”.
2. Run Prim from A and list the edges in the order chosen.
3. Give the final union-find `parent[]` array (weighted union).

▶ **Solution**

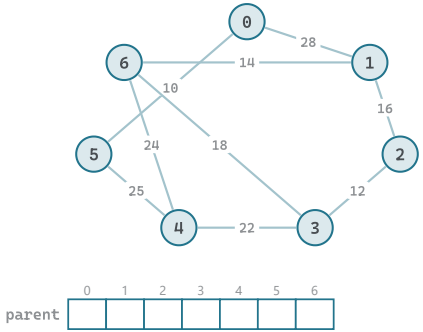
Lab: Kruskal and Prim

Kruskal: 7 vertices, 9 edges

Input (weighted edges, e.g. 0-1:28) `0-1:28, 0-5:10, 1-2:16, 1-6:14, 2-3:12, 3-4:22, 3-6:18, 4-` Apply

CODE: kruskal

```
1  for (i = 0; i < n; i++) parent[i] = -1;
2  qsort(edges, e, sizeof(edge), cmp);
3  for (i = 0, t = 0;
4       i < e && t < n - 1; i++) {
5      int a = find(parent, edges[i].u);
6      int b = find(parent, edges[i].v);
7      if (a != b) { /* no cycle */
8          T[t++] = edges[i];
9          unite(parent, a, b);
10     } /* else skip */
11 }
12 if (t < n - 1) printf("no spanning tree\n");
```



`total 0`

OUTPUT (none yet)

Kruskal: the edges are tried from lightest to heaviest; an edge whose ends are in different sets joins the tree, one whose ends are in the same set is skipped because it would form a cycle. The sets are kept with union-find.

SUMMARY

Complexity Summary

Algorithm	Structure	Time	Note
Try every subset	—	$C(e, n-1) \times O(n)$	exponential, impractical
Kruskal (bad version)	lists + search	$O(e \cdot n)$	“don't code it this way”
Kruskal	sorting/min heap + union-find	$O(e \log e)$	a forest grows and merges
Prim	arrays (lowcost, closest)	$O(n^2)$	dense graphs
Prim	binary heap + adjacency lists	$O(e \log n)$	sparse graphs
DFS/BFS spanning tree	stack / queue	$O(n + e)$	unweighted

SUMMARY

Common Mistakes

1. Checking for a cycle in Kruskal by asking “were both ends chosen before?” (wrong: they may be in different pieces).
2. In Prim, not dropping candidate edges whose ends are both in TV.
3. Confusing an MST with a shortest-path tree: the MST minimizes the total cost, shortest paths minimize individual distances.
4. Forgetting the $|T| < n - 1$ check for a disconnected graph.
5. Uniting non-root vertices in union-find (call `find` first!).
6. Using INT_MAX for ∞ and overflowing when adding.

SUMMARY

Exercises

1. In the 7-village example, what would the MST and its cost be if edge (3,6) cost 11 instead of 18?
2. When can Kruskal and Prim give different trees on the same graph?
3. On the 8-vertex sample graph, list the edges of the DFS and BFS spanning trees starting from 7.
4. How many spanning trees does K_4 have? How many are paths, how many stars?
5. If the edge weights are integers in 1..10, how does Kruskal's sorting step drop to O(e)?
6. Why is the $O(n^2)$ version of Prim poor on sparse graphs?

▶ **Answers / hints**

SUMMARY

Chapter Summary

Spanning tree

An acyclic subgraph with n − 1 edges that connects every vertex; DFS and BFS each produce one.

Minimum cost

The spanning tree of least total weight; the greedy choice is correct (cut property).

Kruskal

Try edges in increasing cost; catch cycles with union-find: O(e log e).

Prim

One tree grows by its cheapest edge: $O(n^2)$ with arrays, O(e log n) with a heap.

Data structure

Same idea, different structure: list search O(e·n) → sets O(e log e).

Choice

Sparse graph: Kruskal or heap-based Prim; dense graph: array-based Prim.

SUMMARY

References

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., §6.3 Minimum Cost Spanning Trees.
- T. H. Cormen et al., *Introduction to Algorithms*, 4th ed., Chapter 19 (Disjoint Sets) and Chapter 21 (Minimum Spanning Trees).
- R. Sedgewick, *Algorithms in C, Part 5*, Chapter 20 (Minimum Spanning Trees).

← Chapter 16: Sets and Union-Find

Course page

Chapter 18: Shortest Paths and Activity Networks

Related: Graphs · Sets and Union-Find

The End

Chapter 17: Minimum Spanning Trees

Next chapter: Chapter 18: Shortest Paths and Activity Networks · Course page