Week 8 — Minimum Spanning Trees & Shortest Paths

Félix Chavelli felix.chavelli@inria.fr

April 15, 2026 · Semester 2

PSL

Today's Agenda

Motivation & Recap

Minimum Spanning Trees

Single-Source Shortest Paths

Summary & What's Next

Part 1

Motivation & Recap

Last Week: Elementary Graph Algorithms

What we covered:

- Graph representations (adj. list / matrix)
- BFS: shortest paths in *unweighted* graphs
- DFS: timestamps, edge classification
- Topological sort (DAGs)
- Strongly connected components (Kosaraju)

Today: weighted graphs!

- BFS finds shortest paths when all edges have weight 1
- What if edges have different weights?
- Two fundamental problems:
  - **MST**: connect everything, minimize total cost
  - **Shortest paths**: reach a destination, minimize total cost

Weighted Graphs — Where Are They?

Network Design (MST)

- Lay cable to connect cities at minimum cost
- Design circuit boards (minimum wiring)
- Cluster analysis in ML (single-linkage)
- Approximation algorithms (TSP)

Navigation (Shortest Paths)

- GPS routing (Dijkstra / A*)
- Network packet routing (OSPF protocol)
- Robot motion planning

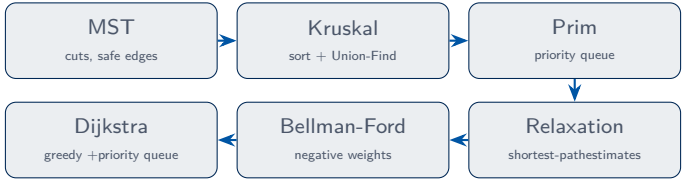
AI & ML

- **Image segmentation:** pixels = vertices, similarity = edge weights, MST-based clustering
- **Knowledge graphs:** weighted relations, shortest semantic paths
- **Reinforcement learning:** state graphs with transition costs

Greedy $\neq$ always wrong

Both MST algorithms are **greedy** and **provably optimal**!

Today's Roadmap



Part 2

Minimum Spanning Trees

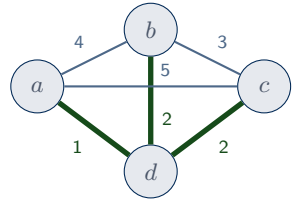
The MST Problem

Minimum Spanning Tree

Given a connected, undirected graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, find an acyclic subset $T \subseteq E$ that **connects all vertices** and minimizes the total weight:

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

- T is a **tree** (acyclic, connected)
- T has exactly $|V| - 1$ edges
- Not necessarily unique!



MST weight $= 1 + 2 + 2 = 5$

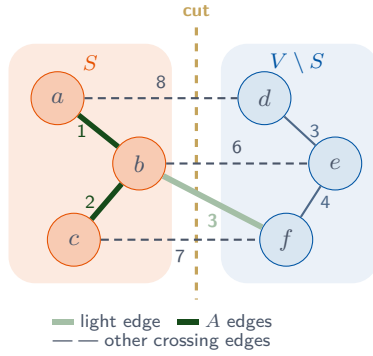
Cuts, Light Edges, and Safe Edges — Definitions

Cut

A **cut** $(S, V \setminus S)$ of an undirected graph $G = (V, E)$ is a partition of V into two non-empty subsets $S \subset V$ and $V \setminus S$.

Crossing edge

An edge $(u, v) \in E$ **crosses** the cut $(S, V \setminus S)$ iff $u \in S \ \wedge \ v \in V \setminus S$ (or vice versa).



Respecting a set of edges

A cut $(S, V \setminus S)$ **respects** a set $A \subseteq E$ iff *no* edge in A crosses the cut: $\forall (u, v) \in A : \ u \in S \Leftrightarrow v \in S$.

Light edge

(u, v) is a **light edge** crossing a cut iff $w(u, v) = \min\{ w(x, y) \mid (x, y) \text{ crosses the cut} \}$.

Cuts, Light Edges, and Safe Edges — Theorem

Theorem 21.1 (CLRS) — Safe edge

Let $G = (V, E, w)$ be a connected, undirected, weighted graph. Let $A \subseteq E$ be a subset of some MST of G, let $(S, V \setminus S)$ be any cut that **respects** A, and let (u, v) be a **light edge** crossing $(S, V \setminus S)$.

Then (u, v) is **safe for** A:

$$A \cup \{(u, v)\} \subseteq \text{some MST of } G.$$

Safe edge

An edge (u, v) is **safe for** A if $A \cup \{(u, v)\}$ is still included in some MST of G.

Key Idea

Both Kruskal and Prim exploit this theorem.
The only difference: how they choose the cut to find the safe edge.

Generic MST Algorithm

Input: Connected graph $G = (V, E)$, weight w

1: $A \leftarrow \emptyset$
2: **while** A does not form a spanning tree **do**
3: Find an edge (u, v) that is **safe for** A
4: $A \leftarrow A \cup \{(u, v)\}$
5: **end while**
6: **return** A

Loop invariant:
A is always a subset of some MST.

Termination:
After $|V| - 1$ safe edges, A is an MST.

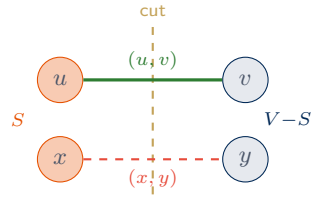
Key question:
How to find a safe edge efficiently?

Safe Edge Theorem — Proof Sketch

Setup: Let T be an MST with $A \subseteq T$. Suppose $(u, v) \notin T$.

Idea (cut-and-paste):

1. Adding (u, v) to T creates a **cycle**
2. This cycle must cross the cut $(S, V-S)$ at another edge (x, y)
3. $(x, y) \notin A$ (since the cut respects A)
4. Remove (x, y), add (u, v): get $T' = T - \{(x, y)\} \cup \{(u, v)\}$
5. Since (u, v) is light: $w(u, v) \leqslant w(x, y)$
6. $w(T') \leqslant w(T)$, so T' is also an MST
7. $A \cup \{(u, v)\} \subseteq T'$ ✓



Replace (x, y) by light edge (u, v)

Kruskal's Algorithm — Idea

Strategy: A is a **forest**.
At each step, add the **lightest edge** that connects two different trees.

Input: Connected graph $G = (V, E)$, weight w

1: $A \leftarrow \emptyset$
2: **for** each vertex $v \in V$ **do**
3: $\quad$ MAKE-SET(v)
4: **end for**
5: Sort edges by weight (increasing)
6: **for** each edge (u, v) in sorted order **do**
7: $\quad$ **if** FIND-SET$(u) \neq$ FIND-SET(v) **then**
8: $\quad\quad$ $A \leftarrow A \cup \{(u, v)\}$
9: $\quad\quad$ UNION(u, v)
10: $\quad$ **end if**
11: **end for**
12: **return** A

Key Idea

Union-Find data structure tracks connected components.

MAKE-SET: $\mathcal{O}(1)$
FIND-SET: $\mathcal{O}(\alpha(n)) \approx \mathcal{O}(1)$
UNION: $\mathcal{O}(\alpha(n)) \approx \mathcal{O}(1)$

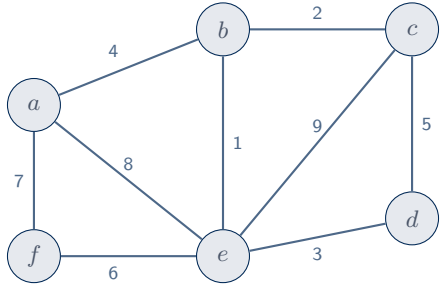
(α = inverse Ackermann, practically constant)

Complexity

$\mathcal{O}(E \log E)$. Note that it is also $\mathcal{O}(E \log V)$ since $E \leqslant V^2$

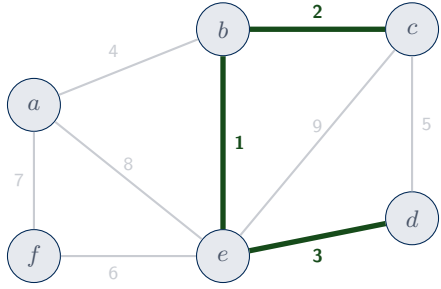
Dominated by sorting the edges.

Kruskal — Step-by-Step: The Graph



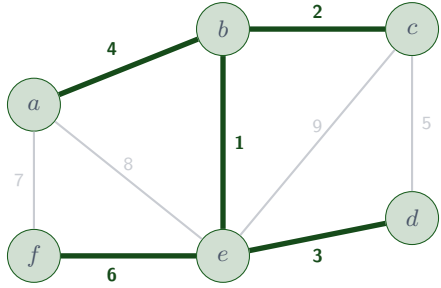
Sorted edges: (b,e):1, (b,c):2, (d,e):3, (a,b):4, (c,d):5, (e,f):6, (a,f):7, (a,e):8, (c,e):9

Kruskal — Steps 1–3: Add lightest edges



Step 1: (b, e):1 — different components $\rightarrow$ **add**
Step 2: (b, c):2 — b, c in different comp. $\rightarrow$ **add**
Step 3: (d, e):3 — d, e in different comp. $\rightarrow$ **add**

Kruskal — Steps 4–5: Complete the MST



Step 4: (a, b):4 — a alone, $\{b, c, d, e\}$ together $\rightarrow$ **add**
Step 5: (c, d):5 — same component! $\rightarrow$ skip (would create cycle)
Step 6: (e, f):6 — f alone $\rightarrow$ **add**. Done! $|V| - 1 = 5$ edges.

MST

$\{(b, e), (b, c), (d, e), (a, b), (e, f)\}$ Total weight: $1 + 2 + 3 + 4 + 6 = 16$

Prim's Algorithm — Idea

Strategy: A forms a **single tree**, grown from a root r.
At each step, add the **lightest edge** connecting the tree to a non-tree vertex.

Input: Graph $G = (V, E)$, weight w, root r

```
1:  for each u ∈ V do
2:      u.key ← ∞;  u.π ← NIL
3:  end for
4:  r.key ← 0
5:  Q ← min-priority queue of all V
6:  while Q ≠ ∅ do
7:      u ← EXTRACT-MIN(Q)
8:      for each v ∈ Adj[u] do
9:          if v ∈ Q and w(u, v) < v.key then
10:             v.π ← u;  v.key ← w(u, v)
11:         end if
12:     end for
13: end while
```

Key Idea

Very similar to **Dijkstra**!

Prim: $v.\text{key}$ = min weight edge to tree
Dijkstra: $v.d$ = min distance from source

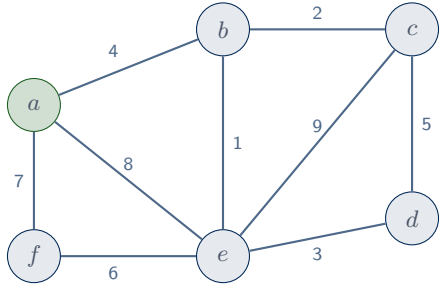
Both use a min-priority queue.

Complexity

Binary heap: $\mathcal{O}(E \log V)$
Fibonacci heap: $\mathcal{O}(E + V \log V)$

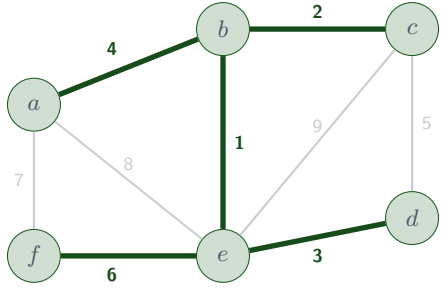
Same as Kruskal (binary heap).

Prim — Step-by-Step: Start from a



Step	Extract-Min	Keys updated
0	a (key 0)	b:4, e:8, f:7
1	b (key 4)	c:2, e:**1** (was 8)
2	e (key 1)	d:3, f:**6** (was 7)
3	c (key 2)	d:**5**→stays 3
4	d (key 3)	—
5	f (key 6)	—

Prim — Result



Same MST as Kruskal! Total weight: $4 + 1 + 2 + 3 + 6 = 16$

Key Idea

Prim grows one tree from a root. **Kruskal** grows a forest, merging components.
Both are greedy, both are optimal, both run in $\mathcal{O}(E \log V)$.

Kruskal in Python — Union-Find

Python

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n
    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # path compression
        return self.parent[x]
    def union(self, x, y):
        rx, ry = self.find(x), self.find(y)
        if rx == ry:
            return False
        if self.rank[rx] < self.rank[ry]:  # union by rank
            rx, ry = ry, rx
        self.parent[ry] = rx
        if self.rank[rx] == self.rank[ry]:
            self.rank[rx] += 1
        return True
```

Kruskal in Python — Main Algorithm

Python

```python
def kruskal(n, edges):
    """edges = [(w, u, v), ...]. Returns MST edges."""
    edges.sort()
    uf = UnionFind(n)
    mst = []
    for w, u, v in edges:
        if uf.union(u, v):
            mst.append((u, v, w))
            if len(mst) == n - 1:
                break
    return mst
```

Key Idea

Sorting dominates: $\mathcal{O}(E \log V)$.
Union-Find operations are nearly $\mathcal{O}(1)$ each (inverse Ackermann).

Prim in Python — Setup

Python

```python
import heapq

def prim(graph, r=0):
    """graph[u] = [(v, w), ...]. Returns MST edges."""
    n = len(graph)
    key = [float('inf')] * n
    parent = [None] * n
    in_tree = [False] * n
    key[r] = 0
    pq = [(0, r)]    # (key, vertex)
    mst = []
```

`key[v]`: lightest edge to tree, `parent[v]`: tree neighbor, `in_tree[v]`: extracted?

Prim in Python — Main Loop

Python

```python
# inside def prim(graph, r=0): ...
while pq:
    k, u = heapq.heappop(pq)
    if in_tree[u]:
        continue            # stale entry
    in_tree[u] = True
    if parent[u] is not None:
        mst.append((parent[u], u, k))
    for v, w in graph[u]:
        if not in_tree[v] and w < key[v]:
            key[v] = w
            parent[v] = u
            heapq.heappush(pq, (w, v))
return mst
```

Lazy decrease-key: push new entries, skip stale pops. Overall: $\mathcal{O}(E \log V)$.

Kruskal vs. Prim — When to Use Which?

	Kruskal	Prim
Strategy	Sort edges + Union-Find	Grow tree + Priority Queue
A is a...	forest	single tree
Time (binary heap)	$\mathcal{O}(E \log V)$	$\mathcal{O}(E \log V)$
Time (Fibonacci heap)	—	$\mathcal{O}(E + V \log V)$
Best when...	sparse graphs	dense graphs

Common Pitfall

MST algorithms require an **undirected, connected** graph.
For directed graphs, the problem is called **minimum spanning arborescence** (much harder: see Edmonds/Chu-Liu algorithm).

Part 3

Single-Source Shortest Paths

The Shortest-Path Problem

Shortest Path

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ and a source vertex s, find for every vertex v the **shortest-path weight**:

$$\delta(s, v) = \begin{cases} \min\{w(p) : s \overset{p}{\rightsquigarrow} v\} & \text{if } v \text{ reachable from } s \\ \infty & \text{otherwise} \end{cases}$$

where $w(p) = \sum_{(u,v) \in p} w(u, v)$.

Key property (Lemma 22.1, CLRS):
Subpaths of shortest paths are shortest paths.

This is the **optimal substructure** that makes shortest-path algorithms work.

Negative cycles

If a negative-weight cycle is reachable from s, then $\delta(s, v) = -\infty$ for vertices on/reachable from the cycle.

Relaxation — The Core Idea

For each vertex v, maintain an **estimate** $v.d \geqslant \delta(s, v)$.

Initialize:

- $s.d \leftarrow 0, \quad v.d \leftarrow \infty$ for $v \neq s$
- $v.\pi \leftarrow \text{NIL}$ for all v

Relax(u, v, w):

1: **if** $v.d > u.d + w(u, v)$ **then**
2: $\quad v.d \leftarrow u.d + w(u, v)$
3: $\quad v.\pi \leftarrow u$
4: **end if**

Before:

After:

$9 > 5 + 2$, so update $v.d \leftarrow 7$

Key Idea

All shortest-path algorithms are just **different strategies for choosing which edges to relax**.

Properties of Relaxation

Key Properties (CLRS 22.5)

After initialization and any sequence of relaxations:

1. **Triangle inequality:** $\delta(s, v) \leqslant \delta(s, u) + w(u, v)$ for all $(u, v) \in E$
2. **Upper-bound:** $v.d \geqslant \delta(s, v)$ always; once $v.d = \delta(s, v)$, it never changes
3. **Convergence:** if $s \rightsquigarrow u \to v$ is a shortest path and $u.d = \delta(s, u)$ before relaxing (u, v), then $v.d = \delta(s, v)$ after
4. **Path-relaxation:** if edges of a shortest path $\langle v_0, v_1, \ldots, v_k \rangle$ are relaxed in order, then $v_k.d = \delta(s, v_k)$

Key Idea

Bellman-Ford: relaxes *all* edges $|V| - 1$ times $\to$ path-relaxation property guarantees correctness.
Dijkstra: relaxes each edge *once*, in the right order $\to$ convergence property guarantees correctness.

Bellman-Ford Algorithm

Handles negative weights!

Input: Graph $G = (V, E)$, weight w, source s

```
1: INITIALIZE-SINGLE-SOURCE(G, s)
2: for i = 1 to |V| − 1 do
3:     for each edge (u, v) ∈ E do
4:         RELAX(u, v, w)
5:     end for
6: end for
7: for each edge (u, v) ∈ E do
8:     if v.d > u.d + w(u, v) then
9:         return FALSE {neg. cycle!}
10:    end if
11: end for
12: return TRUE
```

Complexity

$\mathcal{O}(VE)$

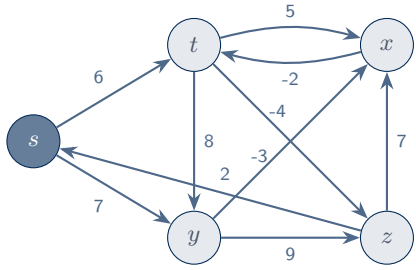
$|V| - 1$ passes $\times$ $|E|$ relaxations.

Why $|V| - 1$ passes?

- Any shortest path has $\leqslant |V| - 1$ edges
- After i passes, all shortest paths with $\leqslant i$ edges are correct
- **Path-relaxation property** ensures convergence

Negative-cycle detection:
If any $v.d$ can still decrease after $|V| - 1$ passes $\Rightarrow$ negative cycle exists!

Bellman-Ford — Step-by-Step



Pass	$s.d$	$t.d$	$x.d$	$y.d$	$z.d$
Init	0	∞	∞	∞	∞
1	0	6	∞	7	∞
2	0	6	4	7	2
3	0	2	4	7	2
4	0	2	4	7	-2

Note: relaxation order matters for intermediate steps, but the final result is always correct. Here we relax edges in the order: (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y).

Shortest Paths in DAGs

Special case: no cycles at all!

Input: DAG $G = (V, E)$, weight w, source s

1: Topologically sort G
2: INITIALIZE-SINGLE-SOURCE(G, s)
3: **for** each u in topo. order **do**
4: **for** each $v \in \text{Adj}[u]$ **do**
5: RELAX(u, v, w)
6: **end for**
7: **end for**

Complexity

$\Theta(V + E)$ — **linear time!**

Why it works:

- Topo. order ensures edges are relaxed left-to-right
- Each edge relaxed exactly once, in the right order
- Path-relaxation property $\Rightarrow$ correct

Key Idea

Works even with **negative weights** (no cycles $\Rightarrow$ no negative cycles).

Application: **PERT/CPM** scheduling (critical path = longest path in a DAG).

Dijkstra's Algorithm — Idea

Greedy + Min-Priority Queue

Like BFS, but for weighted graphs: waves expand to the **closest** vertex, not just the next hop.

Input: Graph $G = (V, E)$, $w \geqslant 0$, source s

1: INITIALIZE-SINGLE-SOURCE(G, s)
2: $S \leftarrow \emptyset$ {finalized vertices}
3: $Q \leftarrow V$ {min-priority queue on $v.d$}
4: **while** $Q \neq \emptyset$ **do**
5: $\quad u \leftarrow$ EXTRACT-MIN(Q)
6: $\quad S \leftarrow S \cup \{u\}$
7: $\quad$ **for** each $v \in$ Adj$[u]$ **do**
8: $\quad\quad$ RELAX(u, v, w)
9: $\quad\quad$ **if** $v.d$ decreased **then**
10: $\quad\quad\quad$ DECREASE-KEY$(Q, v, v.d)$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: **end while**

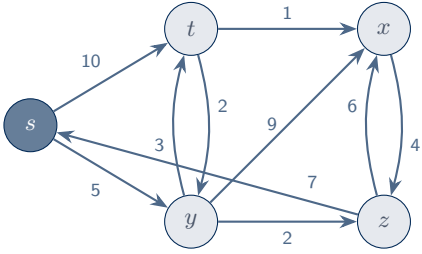
Requires $w \geqslant 0$

Dijkstra **fails** with negative weights!
Use Bellman-Ford instead.

Complexity

Array: $\mathcal{O}(V^2)$
Binary heap: $\mathcal{O}(E \log V)$
Fibonacci heap: $\mathcal{O}(V \log V + E)$

Dijkstra — Step-by-Step



Step	Extr.	$s.d$	$t.d$	$y.d$	$z.d$	$x.d$	Relax
Init	—	**0**	∞	∞	∞	∞	
1	s	0	10	**5**	∞	∞	$s\to t$, $s\to y$
2	y	0	8	5	**7**	14	$y\to t$, $y\to z$, $y\to x$
3	z	0	8	5	7	13	$z\to x$
4	t	0	**8**	5	7	9	$t\to x$
5	x	0	8	5	7	**9**	—

Bold = vertex extracted (finalized).

Each step: extract min from Q, relax outgoing edges.

Dijkstra — Correctness (Sketch)

Theorem 22.6 (CLRS)

Dijkstra's algorithm on a graph with **nonnegative** weights terminates with $u.d = \delta(s, u)$ for all $u \in V$.

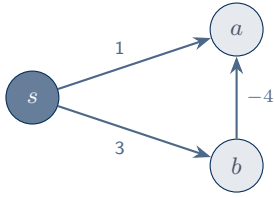
Proof idea (by induction on $|S|$):

1. When we extract u from Q, we claim $u.d = \delta(s, u)$
2. Let y be the first vertex on a shortest path $s \rightsquigarrow u$ that is $\notin S$, predecessor $x \in S$
3. Since $x \in S$: $x.d = \delta(s, x)$ (inductive hypothesis)
4. Edge (x, y) was relaxed when x was extracted $\Rightarrow y.d = \delta(s, y)$
5. Since **all weights** $\geqslant 0$: $\delta(s, y) \leqslant \delta(s, u)$
6. u has min key in Q: $u.d \leqslant y.d = \delta(s, y) \leqslant \delta(s, u) \leqslant u.d$
7. Therefore $u.d = \delta(s, u)$

Why nonneg. weights?

Step 5 fails with negative weights: $\delta(s, y) \leqslant \delta(s, u)$ requires nonneg. edges!

Why Dijkstra Fails with Negative Weights



Dijkstra extracts a first (key $= 1$), finalizes $a.d = 1$.
But the true shortest path is $s \to b \to a$ with weight $3 + (-4) = -1$.

Dijkstra misses it because a was already "finalized"!

Key Idea

Rule of thumb:

- All weights $\geqslant 0$? $\to$ **Dijkstra** ($\mathcal{O}(E \log V)$)
- Negative weights, no neg. cycles? $\to$ **Bellman-Ford** ($\mathcal{O}(VE)$)
- DAG? $\to$ **Topo sort + relax** ($\Theta(V + E)$), works with any weights

Dijkstra in Python

Python

```python
import heapq

def dijkstra(graph, s):
    """graph[u] = [(v, w), ...]. Returns distances + parents."""
    n = len(graph)
    dist = [float('inf')] * n
    parent = [None] * n
    dist[s] = 0
    pq = [(0, s)]
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]: continue  # stale entry
        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                parent[v] = u
                heapq.heappush(pq, (dist[v], v))
    return dist, parent
```

Bellman-Ford in Python

Python

```python
def bellman_ford(n, edges, s):
    """edges = [(u, v, w), ...]. Returns (dist, parent, has_neg_cycle)."""
    dist = [float('inf')] * n
    parent = [None] * n
    dist[s] = 0
    for _ in range(n - 1):
        for u, v, w in edges:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                parent[v] = u
    # Check for negative-weight cycles
    for u, v, w in edges:
        if dist[u] + w < dist[v]:
            return dist, parent, True  # negative cycle!
    return dist, parent, False
```

Part 4

Summary & What's Next

Algorithm Comparison

Algorithm	Time	Neg. weights?	Notes
Minimum Spanning Trees (undirected)			
Kruskal	$\mathcal{O}(E \log V)$	N/A	sort + Union-Find
Prim (binary heap)	$\mathcal{O}(E \log V)$	N/A	grow from root
Prim (Fibonacci)	$\mathcal{O}(E + V \log V)$	N/A	better for dense
Single-Source Shortest Paths (directed)			
DAG shortest paths	$\Theta(V + E)$	✓	topo sort + relax
Bellman-Ford	$\mathcal{O}(VE)$	✓	detects neg. cycles
Dijkstra (binary heap)	$\mathcal{O}(E \log V)$	×	greedy
Dijkstra (Fibonacci)	$\mathcal{O}(V \log V + E)$	×	best for dense

Key Takeaways

1. **MST** = connect everything at minimum total cost.
   Two greedy algorithms: Kruskal (edges sorted) and Prim (grow tree). Both $\mathcal{O}(E \log V)$.
2. **Shortest paths** = reach a destination at minimum cost from a source.
   Relaxation is the universal building block.
3. **Dijkstra** is the workhorse ($\mathcal{O}(E \log V)$), but requires nonneg. weights.
4. **Bellman-Ford** handles negative weights ($\mathcal{O}(VE)$) and detects negative cycles.
5. **DAGs** enjoy linear-time shortest paths via topological sort.
6. The greedy strategy is **provably optimal** for both MST and Dijkstra — a rare and beautiful result!

Key Idea

Understanding **cuts** (MST) and **relaxation** (shortest paths) gives you the mental model to tackle any graph optimization problem.

What's Next

- **Week 9:** All-Pairs Shortest Paths & Applications
  - Floyd-Warshall ($\mathcal{O}(V^3)$)
  - A* search (heuristic-guided Dijkstra)
  - Graph algorithms in AI: planning, knowledge graphs

Key Idea

With BFS, DFS, MST, and shortest paths, you now have the **complete toolkit** for graph problems. Every advanced algorithm (network flows, matching, TSP approximation) builds on these foundations.