Algorithms for Graphs: Dijkstra, Bellman–Ford, Borůvka, and Fredman–Tarjan Methods

Prepared for students

September 6, 2025

Abstract

This document contains lecture notes on several core graph algorithms:

- Dijkstra's single-source shortest paths (nonnegative weights).
- Bellman–Ford (handles negative weights, detects negative cycles).
- Borůvka's step (phase of Borůvka's MST algorithm).
- Fredman–Tarjan ideas (Fibonacci heaps and near-linear MST bounds).

For each algorithm we present: problem statement, algorithmic idea, pseudocode, a small worked example, a real-life motivating example, and an enriched mathematical time-complexity analysis (with proofs/derivations where instructive). The Fredman–Tarjan section includes a careful explanation of the slowly-growing function $\beta(m, n)$ that appears in the MST bounds.

Contents

1 Preliminaries and notation

We consider a graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. Edge weights are denoted $w(e)$. Unless stated otherwise graphs are simple (no parallel edges or loops), but many algorithms work unchanged with slight adaptations when parallel edges exist.

Common notation:

- For a source vertex s, the shortest-path distance $d(s, v)$ is the minimum total weight of any path from s to v.
- A priority queue supports Insert, ExtractMin, and DecreaseKey. The running times of these operations determine the asymptotic cost of Dijkstra/Prim.

2 Dijkstra's algorithm

2.1 Problem

Given nonnegative weights $w(e) \geq 0$ and source s, compute shortest distances $d(s, v)$ for all $v \in V$.

2.2 High-level idea

Maintain a set S of vertices whose shortest distance from s has been finalized. Keep a priority queue keyed by the best-known distances for vertices not yet in S. Repeatedly extract the vertex u with smallest tentative distance, finalize it, and *relax* its outgoing edges: if $d[v] > d[u]+w(u, v)$ then set $d[v] \leftarrow d[u] + w(u, v)$ and update v's key.

2.3 Pseudocode

Algorithm 1 Dijkstra(G, w, s)

Require: Weighted graph $G = (V, E)$ with $w(e) \geq 0$, source s
Ensure: Shortest distances $d[v]$ for all v

for each $v \in V$ **do**
  $d[v] \leftarrow +\infty$; $\pi[v] \leftarrow$ nil
end for
$d[s] \leftarrow 0$
Insert all vertices into priority queue Q keyed by $d[\cdot]$
while Q not empty **do**
  $u \leftarrow \text{ExtractMin}(Q)$
  for each edge $(u, v) \in E$ **do**
    if $d[v] > d[u] + w(u, v)$ **then**
      $d[v] \leftarrow d[u] + w(u, v)$
      $\pi[v] \leftarrow u$
      DecreaseKey$(Q, v, d[v])$
    end if
  end for
end while

2.4 Worked example (numeric)

Let G be directed with vertices s, a, b, t and edges:

$$(s, a, 2),\ (s, b, 5),\ (a, t, 1),\ (b, t, 1),\ (a, b, 3).$$

Initialize $d[s] = 0$, others ∞. Extraction order and updates as shown in standard Dijkstra (see previous paragraph) yields final distances $d[a] = 2, d[t] = 3, d[b] = 5$.

2.5 Real-life example

GPS route planning on a road network where all road costs (travel time or distance) are nonnegative: Dijkstra computes shortest drives from your current location to all reachable nodes. It is also the backbone of many single-source routing computations in network devices when link weights represent nonnegative link costs.

2.6 Time complexity derivation (mathematical)

Let $T_{\text{Ins}}, T_{\text{Ext}}, T_{\text{Dec}}$ denote times for Insert, ExtractMin, DecreaseKey respectively.

If we insert all vertices once, extract each vertex once, and perform at most one decrease-key per edge relaxation (up to m times):

$$\text{Total time} = n \cdot T_{\text{Ins}} + n \cdot T_{\text{Ext}} + m \cdot T_{\text{Dec}} + O(m + n).$$

Common choices:

- **Binary heap (binary priority queue):** $T_{\text{Ins}} = O(\log n)$, $T_{\text{Ext}} = O(\log n)$, $T_{\text{Dec}} = O(\log n)$. Total:
$$O((n + m) \log n).$$
For connected graphs where $m \geq n - 1$, this is often written $O(m \log n)$.

- **Fibonacci heap:** amortized $T_{\text{Ins}} = O(1)$, $T_{\text{Dec}} = O(1)$, $T_{\text{Ext}} = O(\log n)$ (amortized). Thus:

$$O(m + n \log n).$$

  This is the Fredman–Tarjan improvement for Dijkstra/Prim; the key is that decrease-key becomes cheap.

Proof sketch for Fibonacci-heap bound. - There are n extracts $\Rightarrow n \cdot O(\log n) = O(n \log n)$ total for extracts. - There are at most m decrease-keys (one per edge relaxed) $\Rightarrow O(m)$ total. - Insertions contribute $O(n)$. Summing yields $O(m + n \log n)$.

2.7 Remarks

Although Fibonacci heaps give better asymptotic guarantees, in practice simpler heaps (binary, pairing) often perform better due to constants and memory layout. The asymptotic bound is still important for theoretical analysis and for very large graphs where decrease-key operations dominate.

3 Bellman–Ford algorithm

3.1 Problem

Single-source shortest paths in graphs that may have negative edge weights but no negative-weight cycles reachable from the source. Also reports if a negative cycle exists.

3.2 High-level idea

Relax every edge repeatedly. After $n - 1$ full passes over all edges, all shortest paths (that use at most $n - 1$ edges) are found. A n-th pass will detect any negative-weight cycle reachable from the source because some distance can be improved further.

3.3 Pseudocode

Algorithm 2 Bellman–Ford(G, w, s)

Require: Graph $G = (V, E)$, weights $w(e)$ (may be negative), source s
Ensure: Distances $d[v]$ or detection of negative cycle

```
for each v ∈ V do
    d[v] ← +∞; π[v] ← nil
end for
d[s] ← 0
for i = 1 to n − 1 do
    for each edge (u, v) ∈ E do
        if d[u] ≠ +∞ and d[v] > d[u] + w(u, v) then
            d[v] ← d[u] + w(u, v)
            π[v] ← u
        end if
    end for
end for
// Check for negative cycles
for each edge (u, v) ∈ E do
    if d[u] ≠ +∞ and d[v] > d[u] + w(u, v) then
        Report "Graph contains a negative-weight cycle reachable from s."
    end if
end for
Return d, π
```

3.4 Worked example (negative weights)

Graph with vertices s, a, b and edges:

$$(s, a, 4),\ (s, b, 5),\ (a, b, -3).$$

Paths: $s \to a \to b$ has weight $4 + (-3) = 1$ which can be smaller than direct $s \to b$ weight 5. Bellman–Ford relaxes edges across passes and eventually yields $d[a] = 4, d[b] = 1$. If an edge $b \to a$ of weight -2 were added creating a negative cycle $a \to b \to a$ of total weight -5, the n-th pass would detect further improvement and report a negative cycle.

3.5 Real-life example: currency arbitrage detection

Model currencies as vertices and the exchange multipliers as edge weights. Taking logarithms (negated), a profitable arbitrage cycle corresponds to a negative-weight cycle in this transformed graph. Bellman–Ford can detect such negative cycles and hence detect arbitrage opportunities (reachable from some currency).

3.6 Time complexity

Bellman–Ford does $n - 1$ passes across m edges, so:

$$\text{Time} = O(n \cdot m).$$

The final cycle detection pass costs an additional $O(m)$.

Space: $O(n + m)$ to store graph and $O(n)$ for distances.

3.7 Correctness sketch

Any simple path has at most $n-1$ edges. Each pass increasing the allowed path length by 1 guarantees that after k passes, all shortest paths using $\leq k$ edges are found. Hence after $n-1$ passes all shortest (acyclic) paths are found. If an improvement exists in the n-th pass, it must be due to a negative cycle.

3.8 Remarks

Bellman–Ford is used where negative edges are present or needed for detection (e.g., arbitrage), and in many network protocols (distance-vector routing like RIP) variants appear (with caveats about distributed convergence and count-to-infinity).

4 Borůvka's algorithm (Borůvka phase / step)

4.1 Problem

Compute a Minimum Spanning Tree (MST) of an undirected weighted graph $G = (V, E)$.

4.2 Borůvka idea (single phase)

In a Borůvka phase, each component (initially every vertex is its own component) finds its lightest outgoing edge and those edges are added to the forest simultaneously. After adding those edges and contracting components, the number of components decreases significantly. Repeating phases yields the MST.

4.3 Pseudocode: one Borůvka phase

Algorithm 3 BoruvkaPhase$(G, \mathcal{C})$

Require: Graph $G = (V, E)$, partition $\mathcal{C}$ of vertices into components
Ensure: Set of chosen edges F (one per current component, possibly with duplicates)

```
For each component C ∈ 𝒞 set BestEdge[C] ← null
for each edge (u, v) ∈ E do
    C_u ← component of u; C_v ← component of v
    if C_u ≠ C_v then
        if BestEdge[C_u] is null or w(u, v) < weight(BestEdge[C_u]) then
            BestEdge[C_u] ← (u, v)
        end if
        if BestEdge[C_v] is null or w(u, v) < weight(BestEdge[C_v]) then
            BestEdge[C_v] ← (u, v)
        end if
    end if
end for
F ← set of non-null BestEdge entries
return F
```

4.4 Worked example

Graph with vertices $\{1, 2, 3, 4\}$ and weighted edges:

$$(1, 2, 1),\ (2, 3, 2),\ (3, 4, 1),\ (4, 1, 3),\ (1, 3, 4).$$

Initially components are singletons. Each component picks its lightest outgoing edge:

$$\text{comp } 1 \to (1,2), \quad \text{comp } 2 \to (1,2), \quad \text{comp } 3 \to (3,4), \quad \text{comp } 4 \to (3,4).$$

Add $(1,2)$ and $(3,4)$; contract to two components $\{1,2\}$ and $\{3,4\}$. Next phase picks the cheapest connecting edge, e.g., $(2,3)$ with weight 2, completing the MST $\{(1,2),(3,4),(2,3)\}$.

4.5 Real-life example

Designing a minimal-cost backbone network (e.g., fiber-optic connections between data centers): Borůvka's approach is amenable to parallel implementations because each component can find its cheapest outgoing edge independently, making it attractive for distributed/parallel MST construction in large-scale network design.

4.6 Complexity analysis (mathematical)

Time per phase: Each phase scans all edges and does constant-time checks, so $O(m)$.

How many phases? We show that the number of components decreases rapidly:

Let c be the number of components at the start of a phase. Each component selects at least one outgoing edge to another component (unless it's isolated). Consider the directed edges chosen (from each component to the endpoint component). Contracting all chosen edges reduces the number of components: any tree formed by chosen edges with k vertices results from adding $k-1$ edges; therefore each connected subgraph of chosen edges reduces the count of components by at least a factor 2 in many cases. A classical argument shows that the number of components at least halves in every phase that is not trivial (proof sketch below), hence at most $O(\log n)$ phases suffice.

Simple halving proof sketch: Each chosen edge joins two distinct components into one. Consider the forest formed by chosen edges: every tree in this forest with t components yields at least $\lceil t/2 \rceil$ merges (since each chosen edge is between different components and adding all chosen edges pairs components). More precise combinatorial arguments show that the total number of components after a phase is at most $\lfloor c/2 \rfloor$. Therefore the component count reduces geometrically and the number of phases is $O(\log n)$.

Total time:

$$\text{Total time} = O(m \cdot \#\text{phases}) = O(m \log n).$$

Note: In practice Borůvka is used as a building block with other methods (Karger–Klein–Tarjan, etc.) to obtain near-linear MST algorithms.

4.7 Remarks

Borůvka is particularly useful:

- as a *parallel* MST algorithm (each component can independently find its cheapest outgoing edge),
- as an initial contraction step in randomized and deterministic near-linear MST algorithms,
- for graphs where many edges can be discarded early.

5 Fredman–Tarjan ideas and near-linear MST bounds

5.1 Two separate but related things

The Fredman–Tarjan papers introduced two influential ideas:

1. **Fibonacci heaps**, which reduce the amortized cost of DecreaseKey to $O(1)$ and yield improved Dijkstra/Prim running times $O(m + n \log n)$.
2. **MST algorithms with improved bounds** that exploit heap efficiencies and carefully organized component-growth/contract steps to achieve times of the order $O(m\beta(m, n))$ for a very slowly growing function β.

We explain both: the heap effect (briefly) and then the MST complexity notion using $\beta(m, n)$.

5.2 Fibonacci heaps (brief)

A Fibonacci heap supports:

Insert, Meld : $O(1)$ amortized; DecreaseKey : $O(1)$ amortized; ExtractMin : $O(\log n)$ amortized.

The key consequence: algorithms that perform many decrease-key operations (like Dijkstra or Prim) can be accelerated to $O(m + n \log n)$. The structure uses lazy melding of trees and amortized analysis via potential functions.

5.3 The slowly-growing function $\beta(m, n)$

In the Fredman–Tarjan MST analyses a slowly growing function $\beta(m, n)$ appears. There are multiple equivalent ways to define such functions; one natural definition useful here uses iterated logarithms.

Iterated logarithm notation: Define

$$\log^{(0)} n = n, \qquad \log^{(1)} n = \log n, \qquad \log^{(i+1)} n = \log(\log^{(i)} n),$$

where log is base 2 (choice of base affects constants only).

Definition (one common form): For $m \geq n$ define

$$\beta(m, n) = \min\{i \geq 0 \ : \ \log^{(i)} n \leq m/n\}.$$

Equivalently, $\beta(m, n)$ is the number of times we must iterate the logarithm on n until the result becomes less than or equal to m/n. When m/n is large, β may be 0 or 1; when m is near linear in n ($m = O(n)$) β is small (often constant); when m grows faster, β grows extremely slowly. In many regimes $\beta(m, n) = O(\log^* n)$ or even smaller.

Intuition and magnitude: $\beta(m, n)$ grows far more slowly than any polylogarithm. For practical graphs (say n up to 10^{12} and m between n and $n \log n$), β is a very small constant (often ≤ 3). This is why writing complexity as $O(m\beta(m, n))$ is effectively nearly linear for all practical inputs.

5.4 Fredman–Tarjan MST bound (sketch)

Fredman and Tarjan show that with careful component-growth, contraction, and use of Fibonacci heaps (or similar structures), one can organize the MST computation so that the total time is bounded by:

$$T(m, n) = O\big(m \cdot \beta(m, n)\big).$$

Sketch of how such a bound arises:

1. Use Borůvka phases or controlled Prim-like growth to reduce graph size in rounds. Each round contracts components of controlled size.
2. Analyze the work performed in each round: scanning edges, performing heap operations, and handling contractions.
3. Carefully choose size thresholds so that the number of rounds is $\beta(m, n)$; each round costs roughly $O(m)$ when amortized, yielding the $O(m\beta(m, n))$ total.

The full proof uses an inductive amortized accounting argument: by shrinking the problem size sufficiently in a round the next round's cost per vertex drops, and the number of rounds required before the graph becomes trivial equals $\beta(m, n)$.

5.5 Mathematical enrichment: a simplified recursion derivation

We present a simplified recursion that captures the essence (this is an instructive caricature, not the full Fredman–Tarjan proof):

Suppose in each round we can reduce the vertex count from n to $n' = n/f(n)$ for some growth factor $f(n) > 1$ while spending $O(m)$ time processing edges and heaps. If we can choose f so that iterating the reduction t times yields a tiny instance, then total time is $O(tm)$. The function f can be chosen so that $t = \beta(m, n)$. A useful choice is to let $f(n) = \log n$ (or iterated variants). Then iterating $n \mapsto n/\log n$ reduces n to small size in about $\beta(m, n)$ iterations, where β counts logarithmic iterations. Thus $T(m, n) = O(m \cdot \beta(m, n))$.

Why this is near-linear: Because $\beta(m, n)$ is extremely small for practical sizes, $T(m, n)$ behaves almost like $O(m)$; subsequent theoretical improvements (Chazelle, Karger–Klein–Tarjan) reduce the multiplicative slowly-growing factor to inverse-Ackermann type factors or eliminate it via randomization.

5.6 Worked intuition example

For a sparse graph with $m = O(n)$ we have $m/n = O(1)$. Then $\beta(m, n)$ is the minimal i such that $\log^{(i)} n \leq O(1)$, which is essentially $\log^* n$ — extremely small. So the algorithm is $O(n \log^* n)$ in this caricature; with the real Fredman–Tarjan constants the bound becomes even smaller multiplicatively.

5.7 Real-life relevance

While Fredman–Tarjan methods are theoretically oriented, they matter when designing algorithms for very large graphs (e.g., national-scale backbone networks, very large-scale clustering) where nearly-linear behavior is crucial. In those settings careful choice of data structures (heaps) and contraction strategies yield practical performance improvements.

6 Summary comparison and practical advice

- **Dijkstra:** Best choice for nonnegative weights. Use binary heap for simplicity $O((n+m)\log n)$; use Fibonacci/pairing heaps in theory for $O(m + n\log n)$.
- **Bellman–Ford:** Handles negative weights and detects negative cycles. Time $O(nm)$ — use only when negative weights matter.
- **Borůvka:** MST primitive; phase cost $O(m)$, phases $\leq O(\log n)$, overall $O(m\log n)$ when used alone. Very parallel-friendly.
- **Fredman–Tarjan:** Algorithmic ideas producing near-linear MST algorithms $O(m\beta(m,n))$ and heap improvements. Important in theoretical algorithm design for large graphs.

Which to use when (practical):

- Use Dijkstra (binary heap) for moderate graphs with nonnegative weights.
- Use optimized Dijkstra (pairing heap or Fibonacci-like structures) when m is huge and decrease-key operations dominate.
- Use Bellman–Ford only when negative edges are unavoidable or for negative-cycle detection.
- Use Borůvka (parallel) or hybrid MST constructions for massive graphs and when you can exploit parallelism.

References

- E. W. Dijkstra. "A note on two problems in connexion with graphs." Numerische Mathematik (1959).
- R. Bellman. "On a routing problem." Quarterly of Applied Mathematics (1958); L. R. Ford, J. "Network flow and transportation." (Indices of early results).
- O. Borůvka. "O jistém problému minimálním" (1926) — earliest MST idea in Czech.
- M. L. Fredman and R. E. Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms." JACM (1987).
- D. R. Karger, P. N. Klein and R. E. Tarjan. "A randomized linear-time algorithm to find minimum spanning trees." JACM (1995).
- B. Chazelle. "A minimum spanning tree algorithm with inverse-Ackermann type complexity." JACM (2000).

If you want next:

- I can convert these notes to Beamer slides.
- I can add detailed step-by-step TikZ diagrams for each worked example (Dijkstra heap states, Bellman–Ford relaxations, Borůvka phase contractions, Fredman–Tarjan contraction schedule).
- I can produce runnable Python reference implementations (toy and optimized) for each algorithm with unit tests and small datasets.