Single-Source Shortest Paths

BFS, Dijkstra, Bellman-Ford

The Problem

Given a directed graph $G = (V, E)$ with edge weights $w(u, v)$ and a source s, compute for every vertex v

$$\text{dist}(s, v) = \text{the minimum total weight of a path from } s \text{ to } v.$$

The Problem

Given a directed graph $G = (V, E)$ with edge weights $w(u, v)$ and a source s, compute for every vertex v

$$\text{dist}(s, v) = \text{the minimum total weight of a path from } s \text{ to } v.$$

Which algorithm we use depends on the weights:

weights	algorithm	time
all equal (unweighted)	BFS	$O(n + m)$
non-negative	Dijkstra	$O((n + m) \log n)$
arbitrary, may be negative	Bellman-Ford	$O(nm)$

Shortest-Path Trees

Optimal substructure

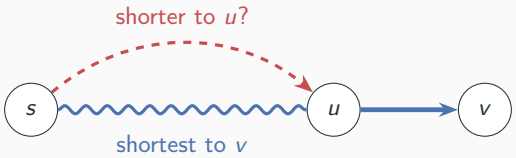
If a shortest path from s to v ends with the edge $u \to v$, then its prefix is a shortest path from s to u.



Shortest-Path Trees

Optimal substructure

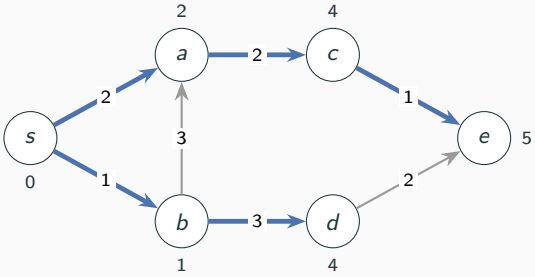
If a shortest path from s to v ends with the edge $u \to v$, then its prefix is a shortest path from s to u.



If the red path existed, red followed by $u \to v$ would beat the shortest path to v. Contradiction.

Shortest-Path Trees

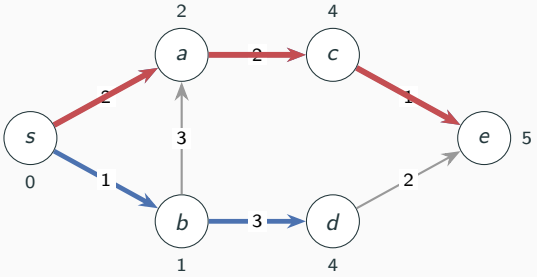
So each vertex only needs its parent: the vertex just before it on a shortest path. The parent edges form a shortest-path tree rooted at s.



Small numbers are distances from s. Blue edges are the tree; $b \to a$ and $d \to e$ are on no shortest path.

Shortest-Path Trees

So each vertex only needs its parent: the vertex just before it on a shortest path. The parent edges form a shortest-path tree rooted at s.



Small numbers are distances from s. Blue edges are the tree; $b \to a$ and $d \to e$ are on no shortest path.

Path to e: follow parents backwards,

$$e \to c \to a \to s,$$

so the shortest path is $s \to a \to c \to e$, of length 5.

BFS: Unweighted Graphs

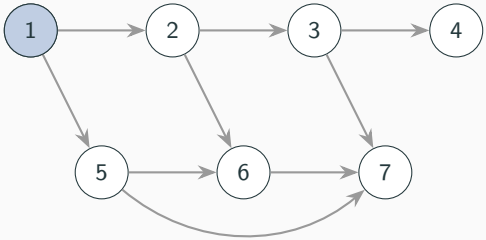
Breadth-First Search

Explore in waves: every vertex at distance d before any vertex at distance $d+1$. A queue (first in, first out) keeps the order.

```
dist[v] ← ∞ for all v;  dist[s] ← 0
Q ← [s]
while Q is not empty:
  u ← dequeue(Q)
  for each neighbour v of u:
    if dist[v] = ∞:          first time we see v
      dist[v] ← dist[u] + 1;  parent[v] ← u
      enqueue(Q, v)
```

BFS in Action

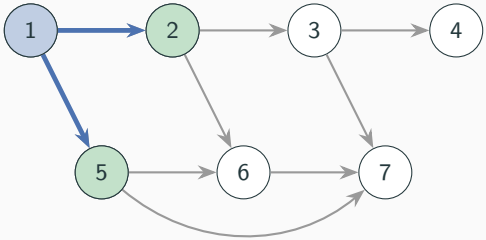
BFS from 1, neighbours in increasing order. Colour = distance.



Start: $Q = [1]$, $\text{dist}[1] = 0$.

BFS in Action

BFS from 1, neighbours in increasing order. Colour = distance.

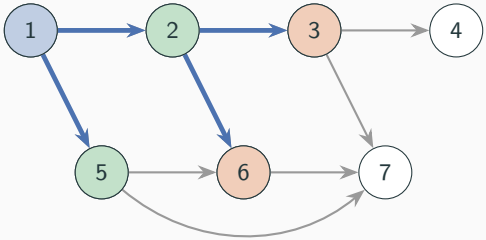


Dequeue 1: discover $2, 5$ at distance 1.

$Q = [2, 5]$

BFS in Action

BFS from 1, neighbours in increasing order. Colour = distance.

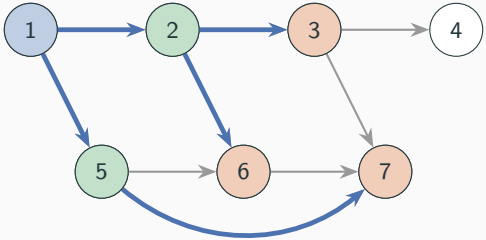


Dequeue 2: discover 3, 6 at distance 2.

$Q = [5, 3, 6]$

BFS in Action

BFS from 1, neighbours in increasing order. Colour = distance.

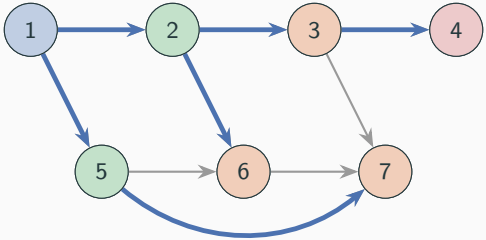


Dequeue 5: 6 is already seen; discover 7 at distance 2.

$Q = [3, 6, 7]$

BFS in Action

BFS from 1, neighbours in increasing order. Colour = distance.

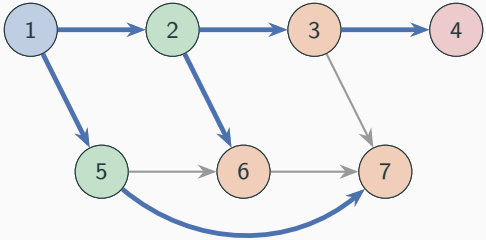


Dequeue 3: discover 4 at distance 3; 7 already seen.

$Q = [6, 7, 4]$

BFS in Action

BFS from 1, neighbours in increasing order. Colour = distance.



Dequeue 6, 7, 4: nothing new. Q is empty.

v	1	2	3	4	5	6	7
dist	0	1	2	3	1	2	2

Blue edges: the shortest-path tree. Edges $3 \to 7$, $5 \to 6$, $6 \to 7$ reach vertices already seen.

Reconstructing a Path

After BFS from 1:

v	1	2	3	4	5	6	7
parent	–	1	2	3	1	2	5

Path to 4: $4 \leftarrow 3 \leftarrow 2 \leftarrow 1$, so $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$.

```
function reconstruct_path(parent, source, target)
    path = [target]
    while path[1] != source
        parent[path[1]] == 0 && return Int[] # target unreachable
        pushfirst!(path, parent[path[1]])
    end
    return path
end
```

Why BFS Is Correct

Claim

For every $k \geq 0$: every vertex at distance k from s gets $\text{dist}[v] = k$, and all of them enter the queue after every vertex at distance $k - 1$ and before any vertex at distance $k + 1$.

Why BFS Is Correct

Claim

For every $k \geq 0$: every vertex at distance k from s gets $\text{dist}[v] = k$, and all of them enter the queue after every vertex at distance $k-1$ and before any vertex at distance $k+1$.

Induction on k. For $k = 0$ only s is enqueued, with label 0.

Let v be at distance $k+1$, and let u be the vertex before v on a shortest path, so u is at distance k.

- No vertex at distance $< k$ has an edge to v (else $\text{dist}(s, v) \leq k$), so v is still unseen when the distance-k vertices start leaving the queue.
- When u leaves the queue, v is discovered, if not already, by a vertex at distance k, so $\text{dist}[v] = k+1$.

BFS: Running Time and Code

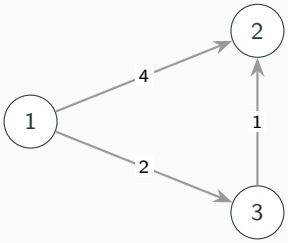
Each vertex is enqueued once and each list scanned once: $O(n + m)$.

```
function bfs(n, adj, source)
    dist = fill(Inf, n)
    parent = zeros(Int, n)
    dist[source] = 0
    queue, head = Int[source], 1 # queue[head:end] is the live queue
    while head <= length(queue)
        u = queue[head]; head += 1
        for v in adj[u]
            if dist[v] == Inf
                dist[v] = dist[u] + 1
                parent[v] = u
                push!(queue, v)
            end
        end
    end
    return dist, parent
end
```

Dijkstra: Non-Negative Weights

When Edges Have Weights

BFS minimises the number of edges, not their total weight.



The direct edge $1 \to 2$ costs 4, but $1 \to 3 \to 2$ costs only 3.

Tense Edges and Relaxation

Keep a tentative distance dist[v], always the length of some path from s to v (or ∞).

An edge $u \to v$ is tense if

$$\text{dist}[u] + w(u, v) < \text{dist}[v] :$$

going through u is a shorter route to v than the best one known.

Tense Edges and Relaxation

Keep a tentative distance dist[v], always the length of some path from s to v (or ∞).

An edge $u \to v$ is tense if

$$\text{dist}[u] + w(u, v) < \text{dist}[v] :$$

going through u is a shorter route to v than the best one known.

Relaxing it fixes that:

$$\text{dist}[v] \leftarrow \text{dist}[u] + w(u, v), \qquad \text{parent}[v] \leftarrow u.$$

Dijkstra and Bellman-Ford both just relax edges. They differ in which edges and in what order.

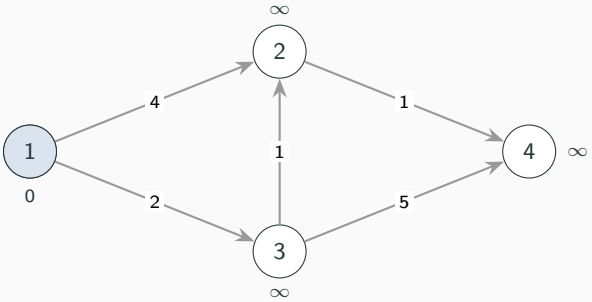
Dijkstra's Algorithm

Repeatedly finalise the unfinished vertex with the smallest tentative distance, then relax its outgoing edges.

$\text{dist}[v] \leftarrow \infty$ for all v; $\text{dist}[s] \leftarrow 0$
$R \leftarrow V$ *unfinished vertices*
while R is not empty:
  $u \leftarrow$ the vertex of R with minimum $\text{dist}[u]$
  remove u from R
  for each edge $u \to v$:
    if $\text{dist}[u] + w(u, v) < \text{dist}[v]$: *tense*
      $\text{dist}[v] \leftarrow \text{dist}[u] + w(u, v)$; $\text{parent}[v] \leftarrow u$

BFS is the special case where every weight is 1.

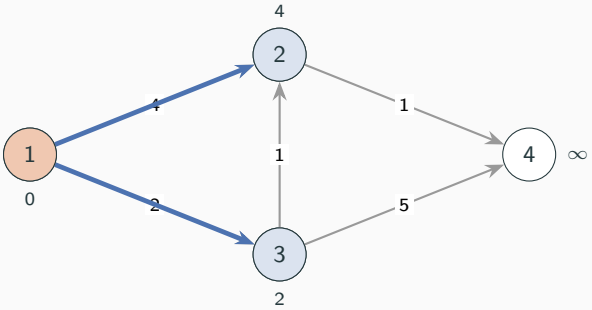
Dijkstra in Action



Initialise: $\text{dist}[1] = 0$, all others ∞.

- extracted now
- finalised
- tentative

Dijkstra in Action



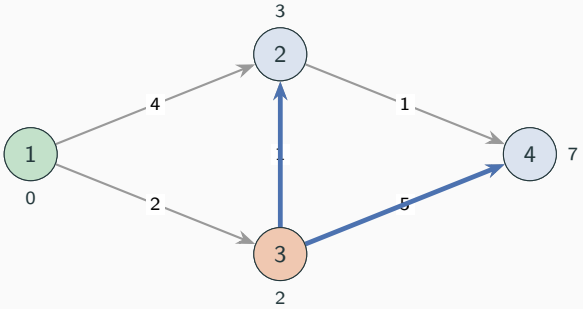
Extract 1 (0). Relax $1 \to 2$: 4. Relax $1 \to 3$: 2.

extracted now

finalised

tentative

Dijkstra in Action



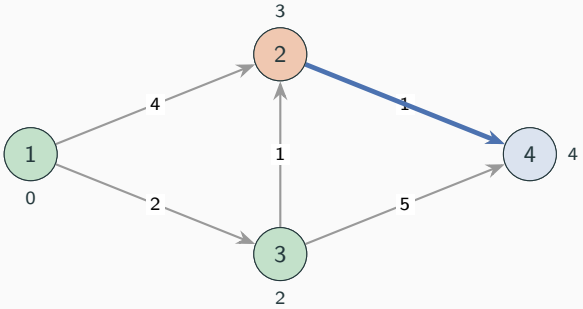
Extract 3 (2). Relax $3 \to 2$:
$\min(4, 2+1) = 3$. Relax $3 \to 4$: 7.

extracted now

finalised

tentative

Dijkstra in Action



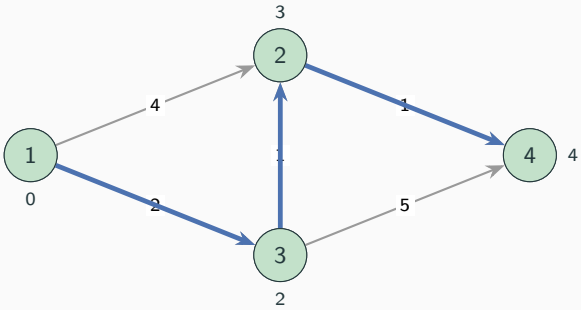
Extract 2 (3). Relax $2 \to 4$:
$\min(7, 3 + 1) = 4$.

extracted now

finalised

tentative

Dijkstra in Action



Extract 4 (4). Done. Blue edges: the shortest-path tree.

extracted now

finalised

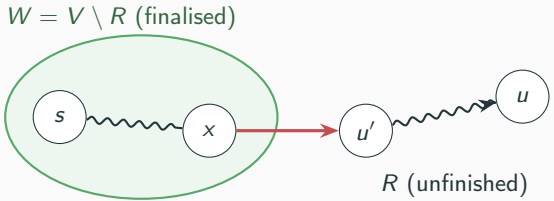
tentative

Correctness

Invariant

When u is removed from R, $\text{dist}[u] = \text{dist}(s, u)$.

Induction on the removal order. s goes first with $\text{dist}[s] = 0$. Now let u be removed next with $\text{dist}[u] = d$, and suppose some path P from s to u has length $d' < d$.



P starts in W and ends in R, so it has a first edge $x \to u'$ with $x \in W$ and $u' \in R$.

Correctness

x is finalised, so by induction $\text{dist}[x] = \text{dist}(s, x)$, and edge $x \to u'$ was relaxed when x was removed. Hence

$$\text{dist}[u'] \;\leq\; \text{dist}[x] + w(x, u') \;\leq\; \text{length of } P \text{ up to } u' \;\leq\; d' \;<\; d,$$

where the third step uses that the rest of P has non-negative weight.

Correctness

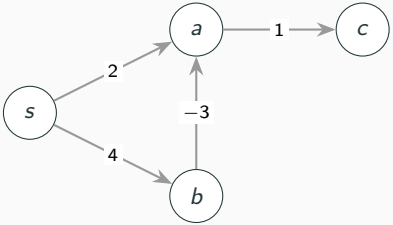
x is finalised, so by induction $\text{dist}[x] = \text{dist}(s, x)$, and edge $x \to u'$ was relaxed when x was removed. Hence

$$\text{dist}[u'] \;\leq\; \text{dist}[x] + w(x, u') \;\leq\; \text{length of } P \text{ up to } u' \;\leq\; d' \;<\; d,$$

where the third step uses that the rest of P has non-negative weight.

But Dijkstra removed u, not u', so $d = \text{dist}[u] \leq \text{dist}[u']$. Contradiction. $\square$

Negative Edges Break It



- Dijkstra finalises s (0), a (2), c (3), then b (4).
- Only then does it see $b \to a$: the true distance to a is $4 - 3 = 1$.
- c is already final with 3, but $\text{dist}(s, c) = 2$.

The proof failed exactly where we used "the rest of P is non-negative".

Running Time

Linear scan for the minimum. n rounds, each scanning R in $O(n)$, plus every edge relaxed once:

$$O(n^2 + m).$$

Good for dense graphs.

Running Time

Linear scan for the minimum. n rounds, each scanning R in $O(n)$, plus every edge relaxed once:

$$O(n^2 + m).$$

Good for dense graphs.

Binary min-heap. Push $(\text{dist}[v], v)$ on every relaxation, and skip stale entries when popped. At most $m + 1$ pushes and pops, each $O(\log m) = O(\log n)$:

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

Good for sparse graphs.

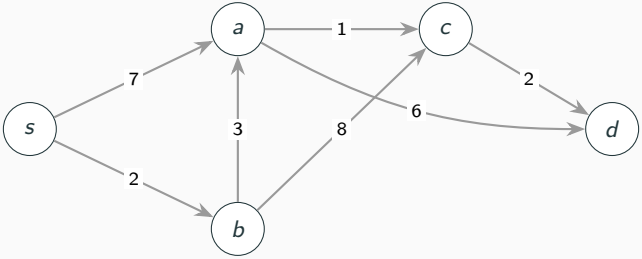
Dijkstra in Julia

```
using DataStructures: PriorityQueue, enqueue!, dequeue_pair!

function dijkstra(n, adj, source) # adj[u]: list of (v, weight) pairs
    dist = fill(Inf, n)
    parent = zeros(Int, n)
    dist[source] = 0.0
    pq = PriorityQueue{Int, Float64}()
    enqueue!(pq, source, 0.0)
    while !isempty(pq)
        u, d = dequeue_pair!(pq)
        d > dist[u] && continue # stale entry
        for (v, w) in adj[u]
            if dist[u] + w < dist[v] # tense edge: relax it
                dist[v] = dist[u] + w
                parent[v] = u
                pq[v] = dist[v] # insert, or lower its key
            end
        end
    end
    return dist, parent
end
```

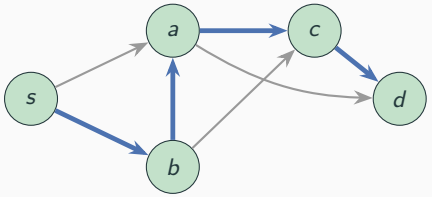
Exercise

Run Dijkstra from s. Give the order in which vertices are finalised, the final distances, and the shortest-path tree.



Solution

extract	s	a	b	c	d
start	0	∞	∞	∞	∞
s (0)		7	2		
b (2)		5		10	
a (5)				6	11
c (6)					8
d (8)					



Order s, b, a, c, d. Distances: $s\colon 0$, $a\colon 5$, $b\colon 2$, $c\colon 6$, $d\colon 8$. Blue edges: the shortest-path tree.

Bellman-Ford: Negative Weights

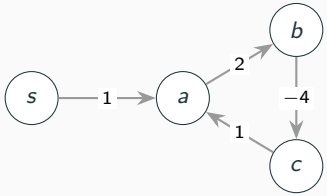
Negative Edges and Negative Cycles

Negative weights are natural: rebates, energy gained, currency exchange with logarithms.

Negative Edges and Negative Cycles

Negative weights are natural: rebates, energy gained, currency exchange with logarithms.

But a negative cycle reachable from s makes distances meaningless: walk around it again and the total keeps falling.

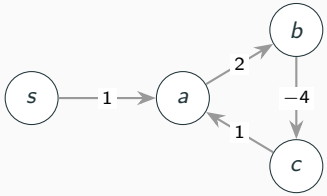


cycle weight $2 - 4 + 1 = -1$

Negative Edges and Negative Cycles

Negative weights are natural: rebates, energy gained, currency exchange with logarithms.

But a negative cycle reachable from s makes distances meaningless: walk around it again and the total keeps falling.



cycle weight $2 - 4 + 1 = -1$

Goal. If no negative cycle is reachable from s, compute all distances. Otherwise, report the negative cycle.

The Key Fact

With no negative cycles, some shortest path to each v is simple: removing a cycle never makes a path longer.

A simple path visits at most n vertices, so it uses at most $n - 1$ edges.

The Key Fact

With no negative cycles, some shortest path to each v is simple: removing a cycle never makes a path longer.

A simple path visits at most n vertices, so it uses at most $n - 1$ edges.

Bellman-Ford

Relax every edge, and repeat that $n - 1$ times.

Round k extends correct shortest paths by one more edge.

The Algorithm

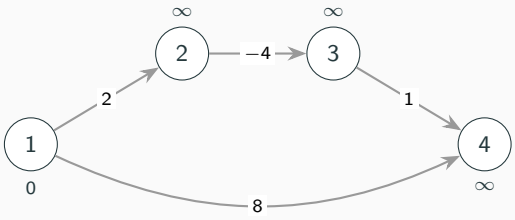
$\text{dist}[v] \leftarrow \infty$ for all v; $\text{dist}[s] \leftarrow 0$
repeat $n-1$ times:
  for each edge $u \to v$:
    if $\text{dist}[u] + w(u,v) < \text{dist}[v]$:
      $\text{dist}[v] \leftarrow \text{dist}[u] + w(u,v)$; $\text{parent}[v] \leftarrow u$

for each edge $u \to v$: *one more pass*
  if $\text{dist}[u] + w(u,v) < \text{dist}[v]$:
    report "negative cycle reachable from s"

$n-1$ rounds of m relaxations each: $O(nm)$.

Bellman-Ford in Action

Edges are scanned in the order $3 \to 4$, $2 \to 3$, $1 \to 2$, $1 \to 4$ in every round.

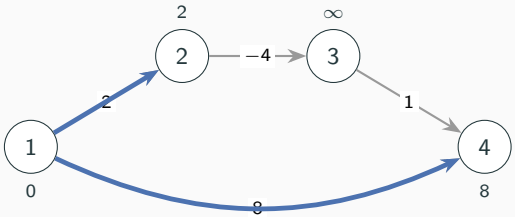


after	1	2	3	4
start	0	∞	∞	∞

$n = 4$, so 3 rounds.

Bellman-Ford in Action

Edges are scanned in the order $3 \to 4$, $2 \to 3$, $1 \to 2$, $1 \to 4$ in every round.

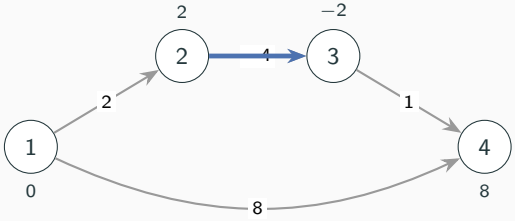


after	1	2	3	4
start	0	∞	∞	∞
round 1	0	2	∞	8

Round 1: $3 \to 4$ and $2 \to 3$ start from ∞; then $1 \to 2$ and $1 \to 4$ fire.

Bellman-Ford in Action

Edges are scanned in the order $3 \to 4$, $2 \to 3$, $1 \to 2$, $1 \to 4$ in every round.

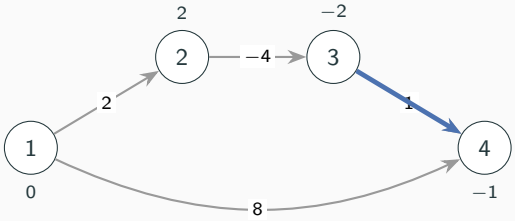


after	1	2	3	4
start	0	∞	∞	∞
round 1	0	2	∞	8
round 2	0	2	-2	8

Round 2: $2 \to 3$ fires, $2 + (-4) = -2$.

Bellman-Ford in Action

Edges are scanned in the order $3 \to 4$, $2 \to 3$, $1 \to 2$, $1 \to 4$ in every round.



after	1	2	3	4
start	0	∞	∞	∞
round 1	0	2	∞	8
round 2	0	2	-2	8
round 3	0	2	-2	-1

Round 3: $3 \to 4$ fires, $-2 + 1 = -1 < 8$. The path $1 \to 2 \to 3 \to 4$ beats the direct edge.

Edge Order Matters, Rounds Don't

With the order $1 \to 2$, $2 \to 3$, $3 \to 4$, $1 \to 4$, a single round already gives the final answer $0, 2, -2, -1$.

- A lucky order can finish early: stop as soon as a round changes nothing.
- An unlucky order needs all $n - 1$ rounds, which is what the analysis must cover.

Edge Order Matters, Rounds Don't

With the order $1 \to 2$, $2 \to 3$, $3 \to 4$, $1 \to 4$, a single round already gives the final answer $0, 2, -2, -1$.

- A lucky order can finish early: stop as soon as a round changes nothing.
- An unlucky order needs all $n - 1$ rounds, which is what the analysis must cover.

The proof only uses one fact about a round: every edge is relaxed at least once in it.

Correctness

Claim

After round k, $\text{dist}[v] \leq$ the weight of the shortest path from s to v that uses at most k edges.

Induction on k. Round 0: $\text{dist}[s] = 0$, everything else ∞.

Round k: take the best path P to v with at most k edges, and let $u \to v$ be its last edge. The rest of P uses at most $k-1$ edges, so after round $k-1$, $\text{dist}[u] \leq w(P) - w(u,v)$. Relaxing $u \to v$ in round k gives

$$\text{dist}[v] \;\leq\; \text{dist}[u] + w(u,v) \;\leq\; w(P).$$

Correctness

Claim

After round k, $\text{dist}[v] \leq$ the weight of the shortest path from s to v that uses at most k edges.

Induction on k. Round 0: $\text{dist}[s] = 0$, everything else ∞.

Round k: take the best path P to v with at most k edges, and let $u \to v$ be its last edge. The rest of P uses at most $k-1$ edges, so after round $k-1$, $\text{dist}[u] \leq w(P) - w(u,v)$. Relaxing $u \to v$ in round k gives

$$\text{dist}[v] \;\leq\; \text{dist}[u] + w(u,v) \;\leq\; w(P).$$

So after $n-1$ rounds: $\text{dist}[v] \leq \text{dist}(s,v)$ by the key fact, and $\text{dist}[v] \geq \text{dist}(s,v)$ since it is always the weight of some path. They are equal. $\square$

Detecting Negative Cycles

Claim

After $n-1$ rounds, some edge is still tense $\iff$ a negative cycle is reachable from s.

(tense $\Rightarrow$ cycle) Suppose no negative cycle is reachable. Then the distances are exact, and exact distances satisfy $\text{dist}(s, v) \leq \text{dist}(s, u) + w(u, v)$, so nothing is tense.

Detecting Negative Cycles

Claim

After $n-1$ rounds, some edge is still tense $\iff$ a negative cycle is reachable from s.

(tense $\Rightarrow$ cycle) Suppose no negative cycle is reachable. Then the distances are exact, and exact distances satisfy $\text{dist}(s,v) \leq \text{dist}(s,u) + w(u,v)$, so nothing is tense.

(cycle $\Rightarrow$ tense) Let $v_1 \to \cdots \to v_k \to v_1$ be a reachable negative cycle (all its dist values are finite). If no edge were tense, then $\text{dist}[v_{i+1}] \leq \text{dist}[v_i] + w(v_i, v_{i+1})$ around the whole cycle. Adding these up, the dist terms cancel:

$$0 \;\leq\; \sum_{\text{cycle}} w(v_i, v_{i+1}) \;<\; 0.$$

Contradiction. $\square$

Bellman-Ford in Julia

```
function bellman_ford(n, adj, source)
    # adj[u] is a list of (v, weight) pairs
    dist = fill(Inf, n)
    parent = zeros(Int, n)
    dist[source] = 0.0
    for _ in 1:n-1
        updated = false
        for u in 1:n
            dist[u] == Inf && continue
            for (v, w) in adj[u]
                if dist[u] + w < dist[v]
                    dist[v] = dist[u] + w
                    parent[v] = u
                    updated = true
                end
            end
        end
        updated || break # already final
    end
    # one more pass
    for u in 1:n
        for (v, w) in adj[u]
            if dist[u] + w < dist[v]
                error("negative cycle")
            end
        end
    end
    return dist, parent
end
```

The early exit keeps the worst case at $O(nm)$ but often stops much sooner.

Summary

algorithm	weights	idea	time
BFS	unweighted	waves, a queue	$O(n+m)$
Dijkstra	≥ 0	finalise the closest unfinished vertex	$O((n+m)\log n)$
Bellman-Ford	any	relax every edge, $n-1$ times	$O(nm)$

- All three build a shortest-path tree through parent pointers.
- Dijkstra and Bellman-Ford are both sequences of relaxations of tense edges.
- Use the fastest algorithm the weights allow.