

UNIVERSITY
OF TRIESTE

Algorithmic Design

University of Trieste — a.a. 2026/2027

Prof. Antonio Emanuele Cinà
`antonio.cina@units.it`

Department of Mathematics, Informatics and Geosciences
University of Trieste

a.a. 2026/2027

Module B — Data Mining through Algorithmic Methods

Chapter 9:

Weighted graphs and shortest paths at scale

Which observed route should we recommend?

Chapter 7 established the modelling workflow; Chapter 8 introduced graph representations and traversal. We now attach weights to the relationships.

The practical task

Given a source station s and a destination t, find a route with minimum total weight under a declared model.

We will follow one small dataset through graph construction, local improvements, and a shortest-path tree.

Pseudocode describes the algorithms independently of a programming language or queue library. Concrete implementation belongs to the laboratory.

Why a plain graph is not enough

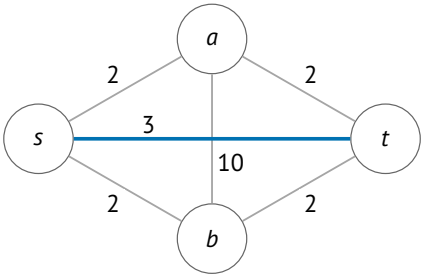
An unweighted graph records whether a relation exists. Many data-mining questions also need to know its *cost*, *strength*, or *reliability*.

Data	Edge weight may represent	Typical question
Road network	minutes, kilometres, toll	cheapest route
Communication network	latency, loss, capacity	fastest or most reliable route
Interaction network	frequency or dissimilarity	strong connections or clusters
Infrastructure plan	construction cost	cheapest connected network

The numerical meaning comes from the application. The algorithm is correct only for the objective encoded by those weights.

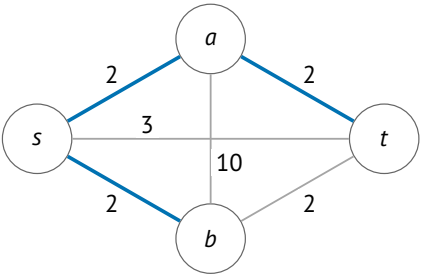
The same weights can support different tasks

Shortest path from s to t



minimise one route: total 3

Minimum spanning tree



connect every vertex: total 6

Shortest path: connect one source to one destination with minimum route weight.

Minimum spanning tree: connect every vertex with minimum total selected edge weight.

An MST need not be a shortest-path tree. This chapter develops shortest paths; Chapter 10 returns to MSTs for clustering and network structure.

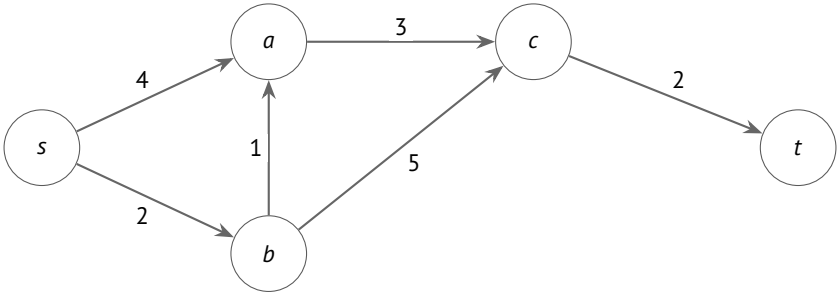
Turn observed durations into edge weights

Each station is a vertex. For one declared time window, create a directed edge for each transition with sufficiently many validated direct journeys. Its weight is their median duration in minutes.

Transition	Observed durations	Median weight
$s \to a$	$4, 4, 5$	4
$s \to b$	$2, 2, 3$	2
$b \to a$	$1, 1, 2$	1
$a \to c$	$3, 3, 4$	3
$b \to c$	$5, 5, 6$	5
$c \to t$	$2, 2, 2$	2

A mean, median or high percentile answers a different operational question.

Read the route graph



Every edge label is a median travel time in minutes.

Arrows encode direction. A reverse connection may be absent or have a different duration.

The placement of vertices does not encode distance or travel time.

Declare what the route graph leaves out

Weighted-graph representation contract

State what vertices and edges mean, whether edges are directed, how observations are aggregated, the unit and interpretation of every weight, the rule for absent edges, and the period of validity.

This example excludes waiting time, future congestion, uncertainty and stations removed during cleaning.

If those quantities matter, change the representation before running the algorithm. Report graph-construction cost separately from query cost.

One weighted graph, several representations

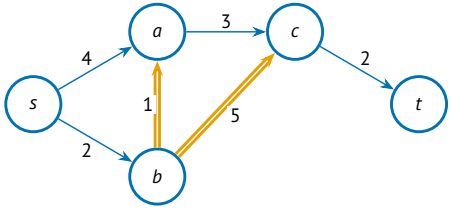
Let $n = |V|$ and $m = |E|$. The stored representation determines which graph operations are cheap.

- A **weight matrix** stores one position for every ordered pair.
- **Weighted adjacency lists** group the outgoing (neighbour, weight) records of each vertex.
- A **weighted edge list** stores (source, target, weight) records directly.

The graph and its mathematical paths do not change. Storage, edge lookup, neighbour enumeration, and complete scans do.

Find outgoing edges in a weight matrix

Weighted route graph



Weight matrix W

destination

source

	s	a	b	c	t
s	0	4	2	∞	∞
a	∞	0	∞	3	∞
b	∞	1	0	5	∞
c	∞	∞	∞	0	2
t	∞	∞	∞	∞	0

Row b: examine 5 positions, return 2 edges

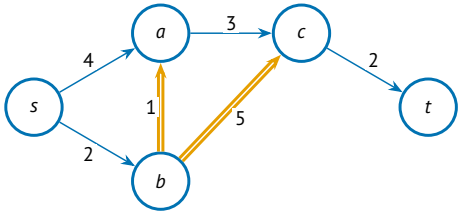
Inspect all $n = 5$ entries in row b; return $(a, 1)$ and $(c, 5)$. The diagonal 0 is an empty path, not a loop.

Edge lookup takes $\Theta(1)$; enumerating one row takes $\Theta(n)$. Storage is $\Theta(n^2)$.

Dashed blue outlines: examined only. Thick orange outlines: examined and returned.

Find outgoing edges in adjacency lists

Weighted route graph



Outgoing adjacency lists

s :	$(a, 4)$	$(b, 2)$
a :	$(c, 3)$	
b :	$(a, 1)$	$(c, 5)$
c :	$(t, 2)$	
t :	[ ]	

List b: examine 2 records, return 2 edges

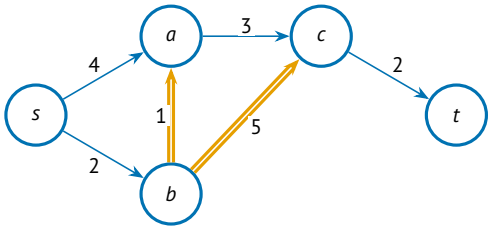
Visit just the two records in Adj[b]: $(a, 1)$ and $(c, 5)$. Other lists are not examined.

Enumeration takes $\Theta(\deg^+(b))$; storage is $\Theta(n + m)$. Testing or deleting a particular edge may require searching the unsorted list.

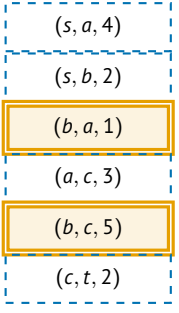
Dashed blue outlines: examined only. Thick orange outlines: examined and returned.

Find outgoing edges in an edge list

Weighted route graph



Edge records (u, v, w)



Test the source of all $m = 6$ records; return the two whose source is b.

One outgoing-edge query can take $\Theta(m)$, but a complete edge pass also takes $\Theta(m)$: a natural fit for Bellman–Ford. Vertex and edge storage uses $\Theta(n + m)$.

Dashed blue outlines: examined only. Thick orange outlines: examined and returned.

Compare the basic operation costs

Let $n = |V|$, $m = |E|$, and let $\deg^+(u)$ be the out-degree of u.

Operation	Matrix	Adjacency lists	Edge list
Space	$\Theta(n^2)$	$\Theta(n+m)$	$\Theta(n+m)$
Read/test (u, v)	$\Theta(1)$	$O(\deg^+(u))$	$O(m)$
Enumerate edges from u	$\Theta(n)$	$\Theta(\deg^+(u))$	$\Theta(m)$
Scan all edge records	$\Theta(n^2)$	$\Theta(n+m)$	$\Theta(m)$

List bounds count inspected records; add constant setup for an empty scan. Lists are unsorted; hash tables or additional indexes change lookup costs and use extra space.

Building the representation also has a cost

Start with n vertices and m already constructed edge records.

Representation	Construction time
Weight matrix	$\Theta(n^2 + m)$
Adjacency lists	$\Theta(n + m)$
Edge list with vertex records	$\Theta(n + m)$

Why the matrix has an extra term

Initialize all n^2 positions, set the diagonal to zero, then insert the m edge weights. Even a sparse graph has a complete matrix.

Aggregation of raw observations is an earlier stage. Report its cost separately from representation construction and shortest-path queries.

Choose the representation for the access pattern

Algorithmic need	Natural representation	Reason
Relax outgoing edges	adjacency lists	touch only actual outgoing edges
Repeat a full edge scan	edge list	one sequential pass over m records
Update every ordered pair	matrix	direct access to $D[i, j]$
Sort edges by weight	edge list	records can be ordered directly

No representation is universally best. The relevant comparison is the cost of the operations used by the selected algorithm on the expected graph density.

Define what a route costs

Let $G = (V, E)$ be directed and $w : E \to \mathbb{R}$ assign a weight to each edge.

Directed walk, path, and weight

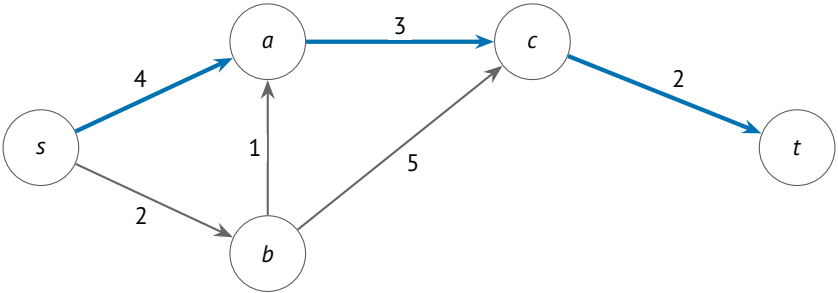
A directed walk $p = \langle v_0, v_1, \ldots, v_k \rangle$ satisfies $(v_{i-1}, v_i) \in E$ for every $i = 1, \ldots, k$. A directed path is a directed walk with no repeated vertex.

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

The empty path from a vertex to itself has weight 0.

For an additive objective, sum the edge weights; do not count only the hops.

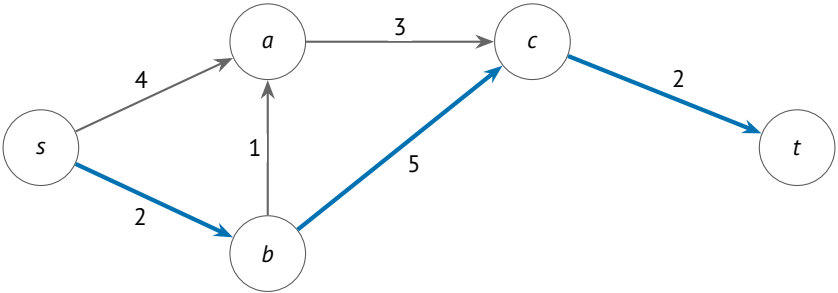
A route through station a



$$s \to a \to c \to t \qquad w(p) = 4 + 3 + 2 = 9.$$

The thick blue edges identify the route being evaluated.

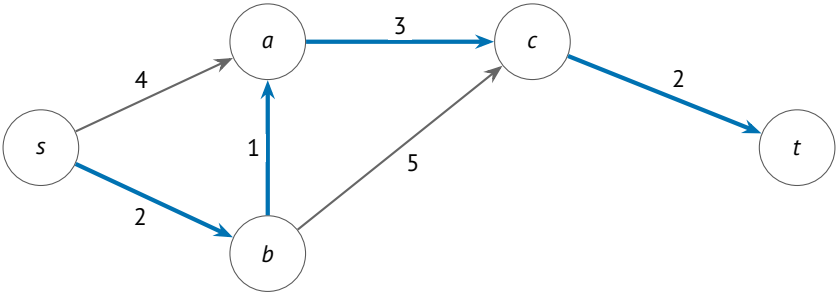
A route through station b



$$s \to b \to c \to t \qquad w(p) = 2 + 5 + 2 = 9.$$

The thick blue edges identify the route being evaluated.

More edges can give a cheaper route



$s \to b \to a \to c \to t \qquad w(p) = 2 + 1 + 3 + 2 = 8.$

This path is shortest in the running graph. Minimum total weight is different from minimum hop count.

Three possible meanings of distance

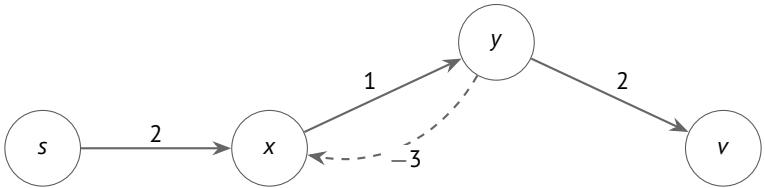
Shortest-path distance

$$\delta(u,v) = \begin{cases} +\infty, & \text{if } v \text{ is unreachable from } u, \\ -\infty, & \text{if } u\text{-to-}v \text{ walk weights are unbounded below}, \\ \min_{p:u\rightsquigarrow v} w(p), & \text{otherwise, over all } u\text{-to-}v \text{ walks}. \end{cases}$$

If the minimum is finite, remove repeated cycles of non-negative total weight to obtain a simple path of the same minimum weight.

A path attaining that minimum is a shortest path.

A negative cycle can remove the finite optimum

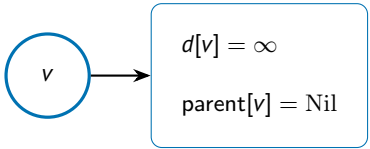


$$s \to x \to y \to v: \quad 2 + 1 + 2 = 5.$$

The cycle $x \to y \to x$ costs $1 - 3 = -2$. Repeating it k times gives an s-to-v walk of weight $5 - 2k$.

Thus $\delta(s, v) = -\infty$: the cycle is reachable from s and can reach v. This is not merely a very negative finite cost.

What each vertex stores



Initial fields before setting the source label to 0

$d[v]$ is the weight of the best route discovered so far. It may decrease.

$\text{parent}[v]$ records the predecessor when an improvement occurs.

$\delta(s, v)$ is the true optimum in the fixed graph. It does not change.

Read the drawings

Inside a vertex: its estimate $d[v]$. Beside an edge: its fixed weight $w(u, v)$. These are different quantities.

In code, $d[v]$ is written `distance[v]`; `distance` and `parent` are arrays indexed by v. With negative cycles, a discovered route can be a walk with repeated vertices.

Relaxation asks one local question

Suppose we know a route from s to u of weight $d[u]$, and (u, v) has weight $w(u, v)$.

1. Extend that route by one edge: the candidate has weight $d[u] + w(u, v)$.
2. Compare it with the best route to v currently known, whose weight is $d[v]$.
3. Keep the cheaper value and remember the predecessor that produced it.

Relax (u, v)

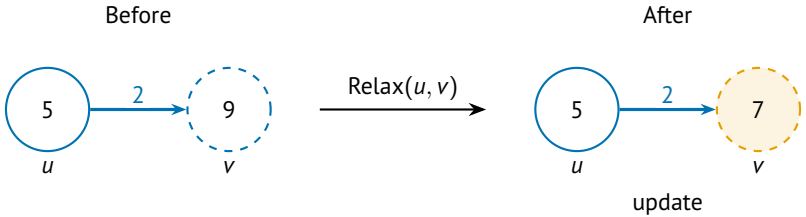
$$d_{\text{new}}[v] = \min\{d[v],\ d[u] + w(u, v)\}.$$

If the candidate wins, set $\text{parent}[v] = u$. Only the fields of v may change.

Relaxation improves one discovered route; it does not by itself certify a shortest path.

Relaxation finds a better route

The numbers inside the circles are estimates. Test the candidate $5 + 2 = 7$.

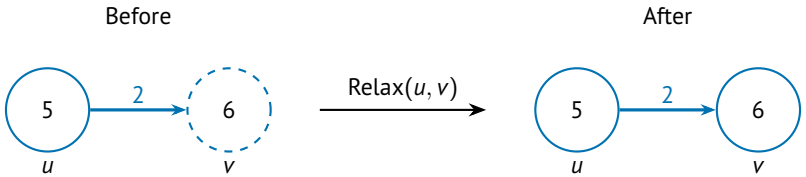


$$9 > 7 : \qquad d[v] \leftarrow 7, \qquad \text{parent}[v] \leftarrow u.$$

Only v changes. Its new estimate describes the route through u; it is not yet certified optimal.

Relaxation leaves the route unchanged

The numbers inside the circles are estimates. Test the candidate $5 + 2 = 7$.



$6 \leq 7$: $\quad d[v] = 6, \quad$ parent$[v]$ stays unchanged.

A candidate must be strictly smaller. Equal-cost alternatives also leave the current parent unchanged.

Initialize once, then relax edges

```
Initialize-single-source(G, s)
    create arrays distance and parent indexed by V
    for each vertex v in V
        distance[v] <- infinity
        parent[v] <- Nil
    distance[s] <- 0
    return distance, parent

Relax(u, v, w, distance, parent)
    if distance[u] != infinity and
       distance[v] > distance[u] + w(u,v)
        distance[v] <- distance[u] + w(u,v)
        parent[v] <- u
```

Initialization creates and returns both arrays. `Relax` receives those same arrays and updates them in place; equal candidates change neither field.

Every finite estimate has a route behind it

Initially, $d[s] = 0$ represents the empty path. Other vertices have estimate ∞.

If a finite $d[u]$ represents a discovered route, extending it by (u, v) gives a route of weight $d[u] + w(u, v)$.

$$\delta(s, v) \leq d[v].$$

What relaxation guarantees

It can improve an estimate, but it does not invent a value unsupported by a walk in the graph. It is an upper bound on the optimum, not a proof that the optimum has been reached.

Predecessors let us reconstruct the answer

The current parent fields define predecessor edges

$$E_\pi = \{(\text{parent}[v], v) : \text{parent}[v] \neq \text{Nil}\}.$$

This subgraph changes whenever a relaxation changes a parent.

After successful single-source termination

Follow parent pointers backward from a reachable destination, then reverse the sequence. Reachable predecessor edges form a shortest-path tree rooted at s.

Unreachable vertices retain parent `Nil`; if drawn, they are isolated. Before successful termination, do not assume the predecessor subgraph is already a shortest-path tree.

Dijkstra's algorithm

Input

A directed graph $G = (V, E)$ with weights w and one source s.

Goal and output

Find minimum total weights from s to all reachable vertices. Return distances d and parents for reconstructing routes.

Applicability

Every edge weight must be non-negative. Unreachable vertices retain distance ∞.

Strategy: finalise the smallest tentative distance, then relax the outgoing edges. This fits the running travel-time model.

Dijkstra finalises the smallest estimate

Assumption: every edge weight is non-negative.

1. Initialize $d[s] = 0$, all other estimates to ∞, and $Q = V$.
2. Extract the vertex with the smallest current estimate. If it is infinite, stop.
3. Otherwise settle that vertex and relax its outgoing edges to vertices still in Q.
4. Repeat. The returned parents reconstruct shortest paths from s.

One algorithm, different selection structures

An array scans for the minimum. A heap maintains priorities when relaxation lowers an estimate. Both must select using the current values of d.

Dijkstra in pseudocode

```
Dijkstra(G, w, s)
    distance, parent <- Initialize-single-source(G, s)
    Q <- set containing every vertex in V

    while Q is not empty
        u <- Extract-min(Q, distance)
        if distance[u] = infinity
            break                  # remaining vertices are unreachable
        for each v in Adj[u]
            if v is in Q
                Relax(u, v, w, distance, parent)
    return distance, parent
```

If an infinite minimum is extracted, stop without scanning its outgoing edges. It is not a finite settled vertex.

A Boolean flag supports constant-time membership. A heap implementation must maintain its keys when estimates decrease.

Reading a Dijkstra execution

Blue double outline Settled: the distance is final.

Orange dashed outline Unsettled with a finite estimate; still in Q.

Plain white outline Unsettled with estimate ∞; also in Q.

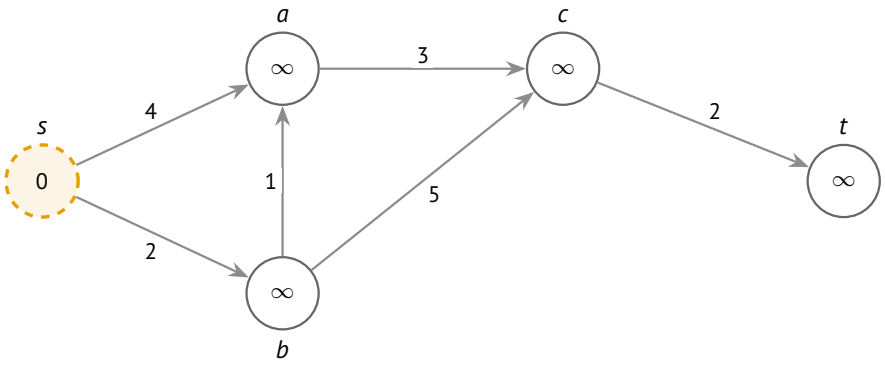
Edges and labels

Thick blue arrows are current predecessor edges. Dashed vermillion marks the edge being tested, including when it becomes a predecessor. Other arrows show the fixed graph edges and weights.

Unlike the BFS diagrams, these styles describe settled status and tentative keys. Dijkstra does not use the BFS White/Grey/Black fields.

We first extract a vertex, then scan *all* its outgoing edges before the next extraction.

Start with only the source known

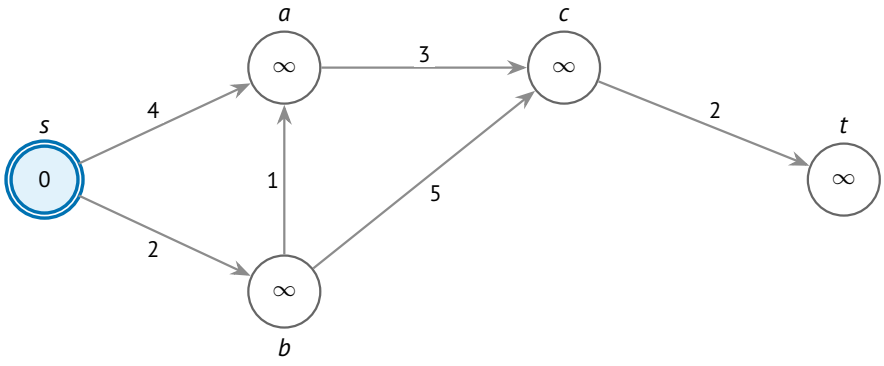


$$Q = \{s{:}0, a{:}\infty, b{:}\infty, c{:}\infty, t{:}\infty\}; S = \varnothing.$$

All parents are Nil. Extract s, the smallest key.

Double outline: settled. Dashed orange: finite key in Q. Plain: key ∞. Thick arrows: current predecessor edges.

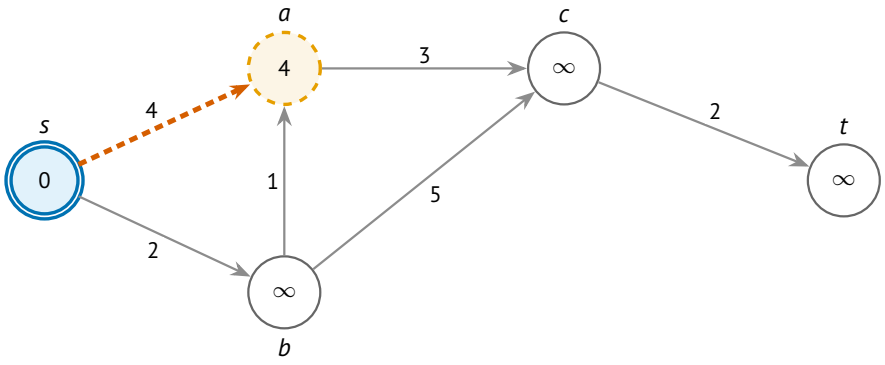
Extract s and make its distance final



$$Q = \{a : \infty, b : \infty, c : \infty, t : \infty\}, \qquad S = \{s\}.$$

The minimum key is $d[s] = 0$. Move s to S before scanning its outgoing edges. All other vertices remain in Q with infinite keys.

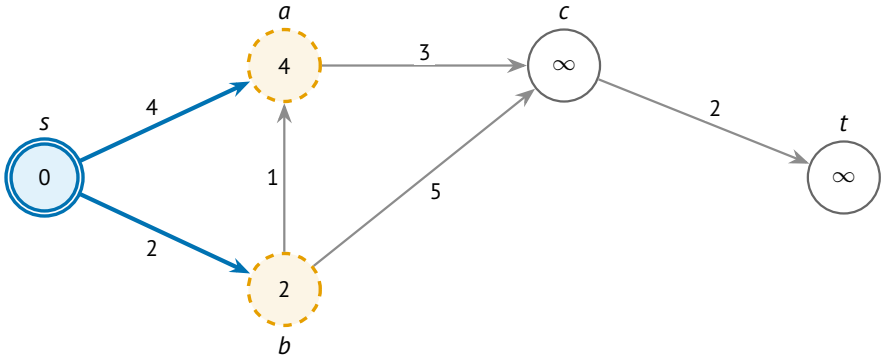
From s, first relax the edge to a



$$Q = \{a : 4, b : \infty, c : \infty, t : \infty\}, \qquad S = \{s\}.$$

Test $\infty > 0 + 4$. Set $d[a] = 4$, parent$[a] = s$ and decrease its queue key. The scan of s is not finished: do not extract a yet.

From s, next relax the edge to b

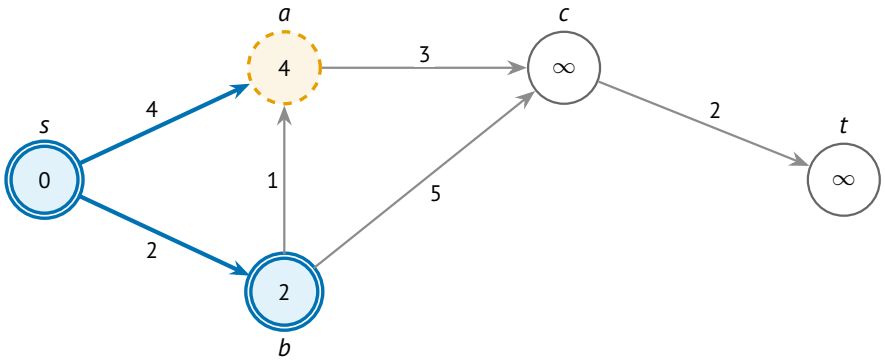


$Q = \{b{:}2, a{:}4, c{:}\infty, t{:}\infty\}$; $S = \{s\}$.

Test $\infty > 0 + 2$: set $d[b] = 2$ and its parent to s. The scan of s is complete. Next extract b, whose key is smaller than that of a.

Double outline: settled. Dashed orange: finite key in Q. Plain: key ∞. Thick arrows: current predecessor edges.

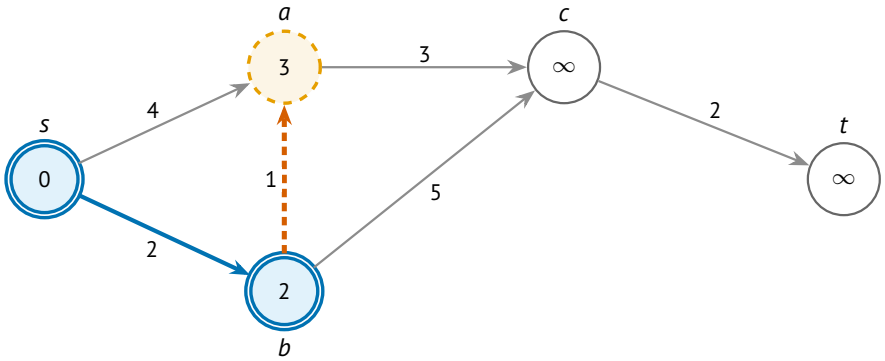
Extract b before scanning its edges



$$Q = \{a : 4, c : \infty, t : \infty\}, \qquad S = \{s, b\}.$$

Key 2 is smaller than 4, so settle b. Its outgoing edges are scanned in the order $b \to a$, then $b \to c$.

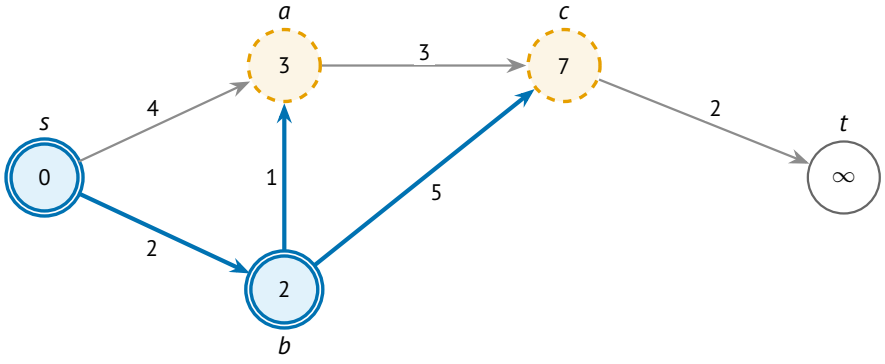
From b, improve the route to a



$$Q = \{a : 3, c : \infty, t : \infty\}, \qquad S = \{s, b\}.$$

Test $4 > 2 + 1$. Change $d[a]$ from 4 to 3 and its parent from s to b. Update the key of a; next scan $b \to c$.

From b, next relax the edge to c

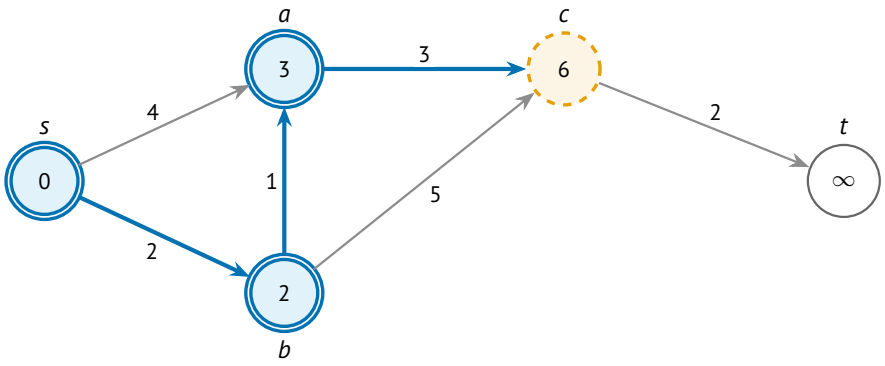


$$Q = \{a{:}3, c{:}7, t{:}\infty\}; S = \{s, b\}.$$

Test $\infty > 2 + 5$: set $d[c] = 7$ and its parent to b. The scan of b is complete. Vertex a now has the smallest key, 3.

Double outline: settled. Dashed orange: finite key in Q. Plain: key ∞. Thick arrows: current predecessor edges.

Settle a and improve the route to c

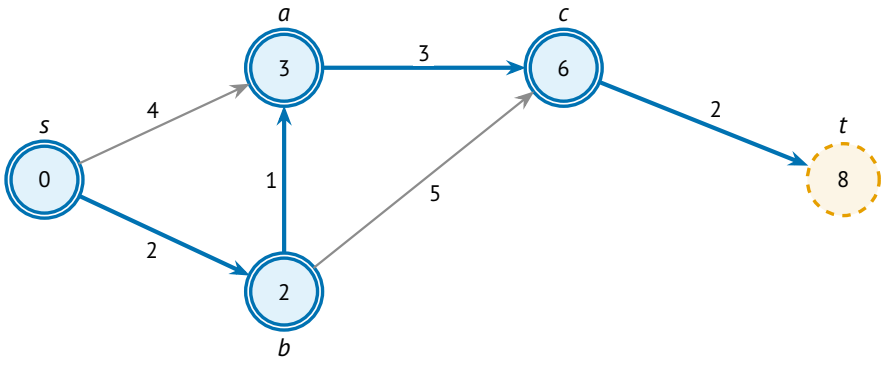


$$Q = \{c{:}6, t{:}\infty\};\ S = \{s, b, a\}.$$

$3 + 3 = 6 < 7$: set $d[c] = 6$ and change its parent from b to a.

Double outline: settled. Dashed orange: finite key in Q. Plain: key ∞. Thick arrows: current predecessor edges.

Settle c and reach the destination

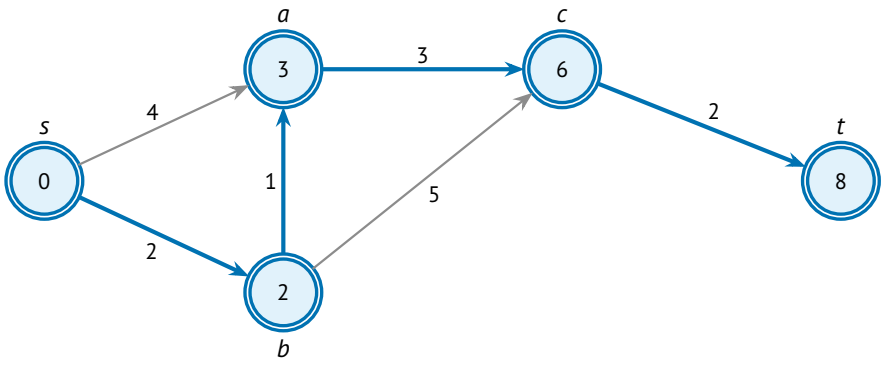


$$Q = \{t{:}8\}; S = \{s, b, a, c\}.$$

Relax $c \to t$: set $d[t] = 6 + 2 = 8$ and parent$[t] = c$.

Double outline: settled. Dashed orange: finite key in Q. Plain: key ∞. Thick arrows: current predecessor edges.

Settle t and finish the computation

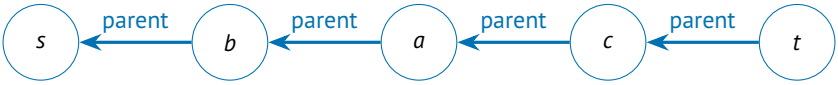


$$Q = \varnothing;\ S = V.$$

All distances are final. The thick arrows form the shortest-path tree rooted at s.

Double outline: settled. Dashed orange: finite key in Q. Plain: key ∞. Thick arrows: current predecessor edges.

Follow parents backward, return the route forward



$$t, c, a, b, s \quad \longrightarrow \quad s, b, a, c, t.$$

$$w(p) = 2 + 1 + 3 + 2 = 8 = d[t].$$

These arrows are parent pointers, directed backward. Graph edges used by the returned path run forward.

Dijkstra running time

```
Dijkstra(G, w, s)
    distance, parent <- Initialize-single-source(G, s)
    Q <- set containing every vertex in V  # Theta(n)
    while Q is not empty                   # at most n times
        u <- Extract-min(Q, distance)      # array or heap
        if distance[u] = infinity
            break
        for each v in Adj[u]               # graph access
            if v is in Q                   # Theta(1) flag
                Relax(u, v, w, distance, parent)
    return distance, parent
```

Initialization costs $\Theta(n)$. An abstract relaxation costs $\Theta(1)$; a heap implementation must also maintain its keys after successful improvements.

Two independent implementation choices

Let $n = |V|$ and $m = |E|$. Each iteration performs two different searches.

Store the graph

Find the outgoing edges of the selected vertex u.

Choose weighted adjacency lists or a weight matrix.

Store the unsettled set Q

Find the vertex with the smallest current distance.

Choose an unsorted array or an indexed binary heap.

An adjacency list is not a priority queue. The implementation chooses one structure from each column.

Discuss: If we replace the array for Q by a heap, does scanning a matrix row become cheaper?

Count the outgoing-edge inspections

After extracting u, inspect its outgoing edges. In the worst case, all vertices are reachable.

Weighted adjacency lists

Read $\deg^+(u)$ edge records for vertex u:

$$\sum_{u \in V} \deg^+(u) = m.$$

Only existing edges are inspected. The edge inspections cost $\Theta(m)$ in total, in addition to vertex overhead.

Weight matrix

Read all n entries of row u, even if most are ∞:

$$n \text{ rows} \times n \text{ entries} = \Theta(n^2).$$

A heap cannot remove these row scans.

An array pays for minimum selection

An unsorted array does not maintain priority order. A successful relaxation changes the distance directly; there is no heap to repair.

Extract the minimum by scanning

When all vertices are reachable, the successive unsettled sets have sizes

$$n,\ n-1,\ \ldots,\ 1.$$

Thus the selection cost is

$$\Theta(n + (n-1) + \cdots + 1) = \Theta(n^2).$$

A Boolean settled flag makes the membership test constant time. A fixed-length array may instead scan all n positions on every extraction, with the same worst-case total.

A heap pays for extraction and key repair

An indexed binary heap stores one entry per unsettled vertex, keyed by its current distance.

Operation	Number	Cost per operation
Build initial heap	1	$\Theta(n)$
Extract minimum	at most n	$O(\log n)$
Repair a decreased key	successful relaxations only	$O(\log n)$

Why indexed?

A vertex-to-position index locates the entry of v without a linear search. After its distance decreases, move that entry upward until heap order is restored.

An unsuccessful relaxation leaves the priority unchanged and requires no repair.

Why there can be more repairs than vertices

In the running five-vertex graph, the successful changes are:

Settled vertex	Examined edge	Improved estimate
s	$s \to a$	$d[a] : \infty \to 4$
s	$s \to b$	$d[b] : \infty \to 2$
b	$b \to a$	$d[a] : 4 \to 3$
b	$b \to c$	$d[c] : \infty \to 7$
a	$a \to c$	$d[c] : 7 \to 6$
c	$c \to t$	$d[t] : \infty \to 8$

There are six successful relaxations but only five vertices. Both a and c improve twice before settlement.

Discuss: What can each successful improvement be charged to?

Where the two heap terms come from

Let h count successful relaxations. Each is caused by an examined edge, and each edge is examined at most once, so $h \leq m$.

$$\underbrace{O(n \log n)}_{\text{minimum extractions}} + \underbrace{O(h \log n)}_{\text{successful key repairs}} \subseteq O((n+m)\log n).$$

Adjacency lists and an indexed heap

Include initialization and edge inspections:

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

The constant-time comparison inside `Relax` and the heap repair are different costs. Maintaining the queue implements the same mathematical algorithm.

Auxiliary state is $\Theta(n)$ beyond the graph.

Combine graph storage and queue costs

Upper bounds after the graph has been built:

Graph storage	Unsorted-array Q	Indexed-heap Q
Adjacency lists	$O(n^2 + m)$	$O((n + m) \log n)$
Weight matrix	$O(n^2)$	$O(n^2 + (n + m) \log n)$

Sparse and dense inputs

For a simple graph, $m = O(n^2)$. Array-based selection takes $O(n^2)$ time; this is tight when all vertices are reachable.

Lists with a heap exploit sparsity. A matrix with an array is a simple dense-graph choice. A matrix with a heap retains quadratic row scans and has an $O(n^2 \log n)$ dense-graph bound.

Separate input work and reachable work

Let R be the vertices reachable from s, $r = |R|$, and m_R the number of edges leaving them.

What the execution actually visits

It makes r finite extractions. If $r < n$, one further extraction returns an infinite key and triggers the stop.

Lists inspect m_R edges; a matrix inspects nr entries. The worst case has $r = n$.

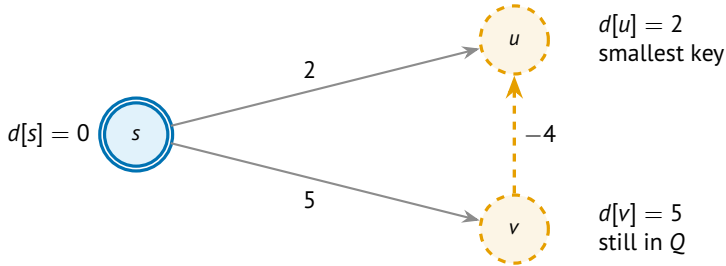
What happens before the query

Reading an explicit $n \times n$ matrix costs $\Theta(n^2)$. Building lists from clean edge records costs $\Theta(n + m)$, including empty lists.

Cleaning, aggregating and validating raw observations are separate data-preparation costs.

Why Dijkstra fails with negative edges

After processing s, the tentative distances are $d[u] = 2$ and $d[v] = 5$.



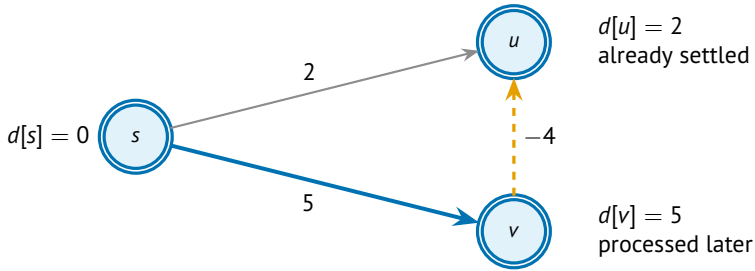
Dijkstra selects the smallest estimate

It extracts u before v and settles u with distance 2.

Double outlines mean settled; dashed outlines mean still in Q. The dashed arrow has weight -4.

A later edge reveals a cheaper path

When v is processed, its edge to u gives a candidate cost $5 + (-4) = 1$.



The earlier decision cannot be trusted

The true shortest path is $s \to v \to u$, with cost $1 < 2$. But u is already outside Q: the queue-membership test in the pseudocode prevents relaxing an edge to a settled destination.

The algorithm returns $d[u] = 2$, although $\delta(s, u) = 1$.

Negative edges require a different guarantee

No negative cycle is needed

The example has no cycle at all. A negative edge can make a later route cheaper than an estimate already finalised. Dijkstra is therefore not guaranteed to work on graphs with negative edges.

Adding a constant does not repair the problem

Adding c to every edge adds kc to a path containing k edges. Paths with different numbers of edges receive different offsets, so their order can change.

Bellman–Ford keeps estimates revisable instead of settling vertices permanently. It allows negative edges and detects a negative cycle reachable from the source.

Why can a useful model contain negative edges?

Negative weights need not mean “negative travel time.” They arise when an edge records a signed change rather than a physical length.

- A rebate, credit, or recovered energy can reduce an accumulated cost.
- Multiplicative gains can be transformed with $w(u, v) = -\log r(u, v)$. A gain $r(u, v) > 1$ produces a negative edge.
- In a currency graph, a negative cycle after this transformation represents a product of exchange rates greater than one: an arbitrage opportunity in the idealised model.

Dijkstra cannot finalise a frontier when a later negative edge may undo the apparent lead. We need an algorithm that keeps estimates revisable.

Bellman–Ford

Input

A directed weighted graph $G = (V, E)$ and one source s; an edge list supports repeated complete scans.

Goal and output

Return shortest distances and parents when no negative cycle is reachable from s; otherwise report that such a cycle exists.

Applicability

Negative edges are allowed. A negative cycle outside the source-reachable part is not reported.

Strategy: keep estimates revisable and scan all edges $n - 1$ times, where $n = |V|$. Use one further scan for cycle detection.

Bellman–Ford keeps estimates revisable

Bellman–Ford removes greedy finalisation. It repeatedly scans *every edge* and applies the same relaxation rule.

What a pass guarantees

Improved information can advance at least one edge farther along a path per complete pass. With in-place updates, several edges may propagate in the same pass, depending on scan order.

Negative edges are allowed. A further scan detects a negative cycle reachable from the source.

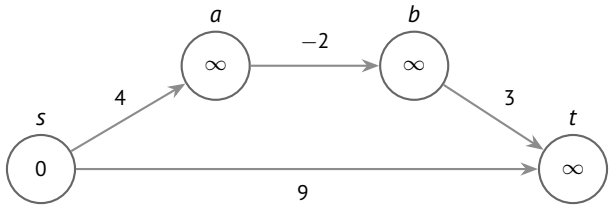
Bellman–Ford in pseudocode

```
Bellman-Ford(G, w, s)
    distance, parent <- Initialize-single-source(G, s)
    for pass <- 1 to |V|-1
        for each edge (u,v) in E
            Relax(u, v, w, distance, parent)

    for each edge (u,v) in E
        if distance[u] != infinity and
           distance[v] > distance[u] + w(u,v)
            return Failure-negative-cycle
    return distance, parent
```

The final guard excludes unreachable tails. Failure reports a negative cycle reachable from s; a cycle elsewhere is irrelevant to this query.

Start with the source

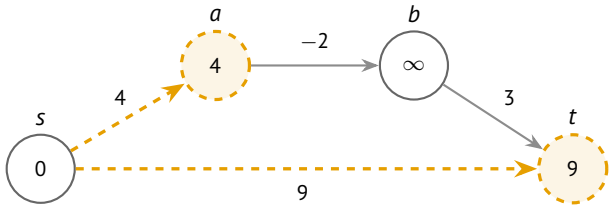


What changes?

Only $d[s] = 0$ is finite. Scan $(b, t), (a, b), (s, a), (s, t)$ in this order in every pass. No vertex is settled.

Dashed orange arrows show successful tests; orange outlined vertices changed. Thick blue arrows show other current parents. Labels inside vertices are distance estimates.

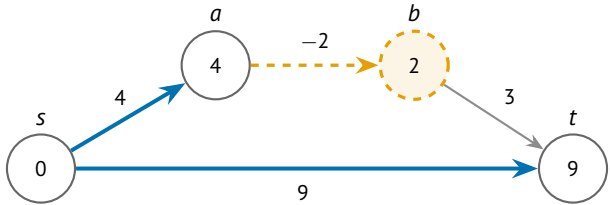
Pass 1: discover two direct routes



What changes?

The first two tails have infinite estimates: no change. Then $d[a] : \infty \to 4$ and $d[t] : \infty \to 9$; both parents become s. The direct route to t is only tentative.

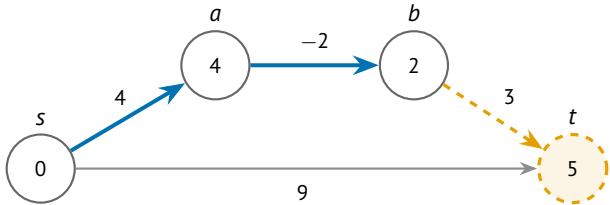
Pass 2: reach b through the negative edge



What changes?

$a \to b$ gives $d[b] : \infty \to 4 - 2 = 2$ and $\text{parent}[b] = a$. The edge $b \to t$ was tested earlier, so $d[t]$ stays 9 until the next pass. The other tests change nothing.

Pass 3: improve the route to t



What changes?

$b \to t$ now gives $d[t] : 9 \to 2 + 3 = 5$; its parent changes from s to b. All remaining tests fail to improve an estimate. The route $s \to a \to b \to t$ costs 5, rather than 9.

The final scan tests without updating

For the small example, $d = (0, 4, 2, 5)$ in order (s, a, b, t).

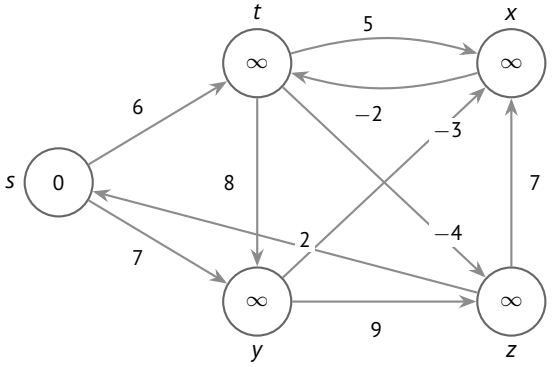
Edge	Target	Candidate	Improves?
$b \to t$	5	$2 + 3 = 5$	No
$a \to b$	2	$4 - 2 = 2$	No
$s \to a$	4	$0 + 4 = 4$	No
$s \to t$	5	$0 + 9 = 9$	No

Return distances and parents

The extra scan changes no array entry. This graph is acyclic: its negative edge cannot belong to a negative cycle.

The reversed route order made progress take three passes. A different scan order can propagate several edges within one pass.

A larger example with the same relaxation rule

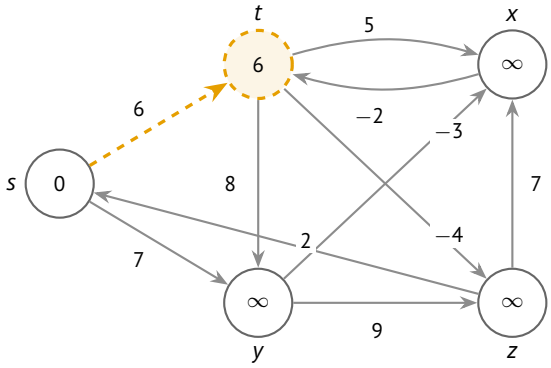


Fixed scan order in every pass:

$$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y).$$

Five vertices require four main passes. Only s is finite initially. Dashed orange marks successful tests; orange vertices changed; thick blue marks other current parents. No vertex is settled.

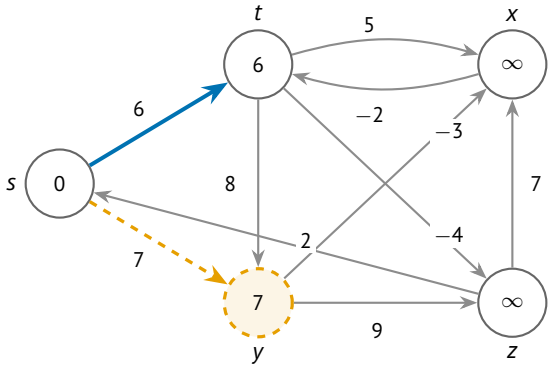
Pass 1: relax $s \to t$



What changes?

Edges 1–8 have infinite tails, so they do nothing. Edge 9 discovers t: $d[t] : \infty \to 0 + 6 = 6$; set its parent to s. This value is not final.

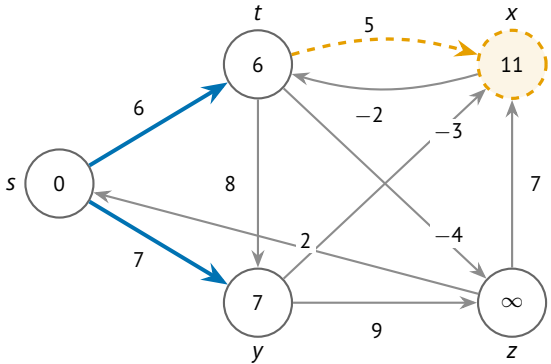
Pass 1: relax $s \to y$



What changes?

Edge 10 discovers y: $d[y] : \infty \to 0 + 7 = 7$; set its parent to s. End of pass 1: $(0, 6, \infty, 7, \infty)$. The outgoing edges of t, y wait until the next pass.

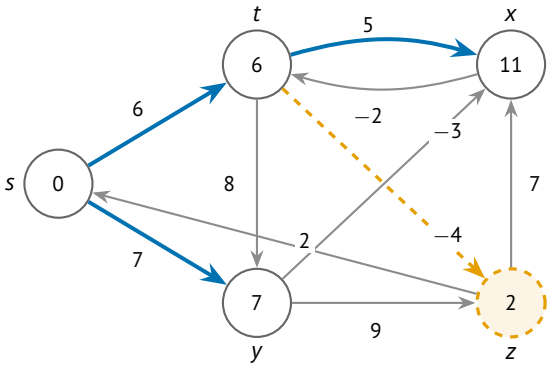
Pass 2: relax $t \to x$



What changes?

Edge 1 uses $d[t] = 6$: $d[x] : \infty \to 6 + 5 = 11$; set its parent to t. This first route to x can still improve later in this pass.

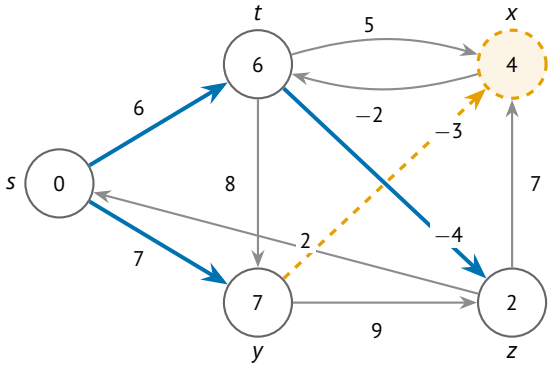
Pass 2: relax $t \to z$



What changes?

Edge 2 offers $6 + 8 = 14$ for y, so keep 7. Edge 3 gives $d[z] : \infty \to 6 - 4 = 2$; set its parent to t.

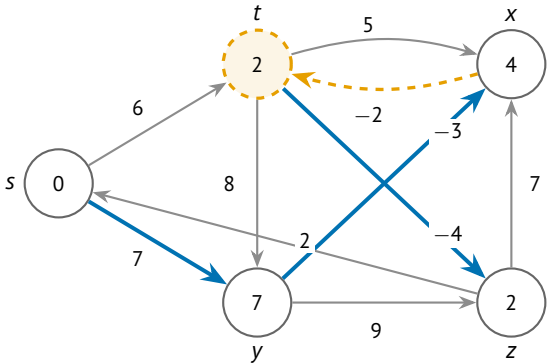
Pass 2: relax $y \to x$



What changes?

Edge 4 offers $11 - 2 = 9$ for t, so keep 6. Edge 5 gives $d[x] : 11 \to 7 - 3 = 4$; its parent changes from t to y. Edges 6–10 change nothing. End of pass 2: $(0, 6, 4, 7, 2)$.

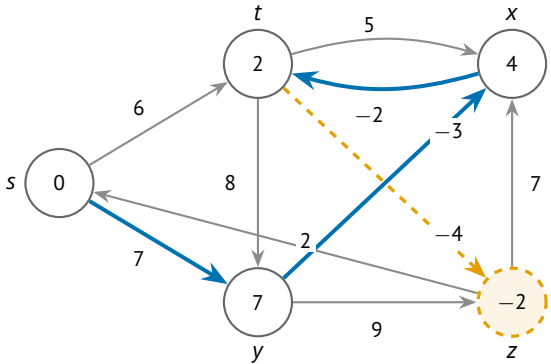
Pass 3: relax $x \to t$



What changes?

Edges 1–3 change nothing. Edge 4 gives $d[t] : 6 \to 4 - 2 = 2$; its parent changes from s to x. Edges 5–10 change nothing. Since $t \to z$ was already scanned, $d[z]$ stays 2 in this pass.

Pass 4: relax $t \to z$



What changes?

Edges 1–2 change nothing. Edge 3 gives $d[z] : 2 \to 2 - 4 = -2$; its parent remains t. Edges 4–10 change nothing. End of pass 4: $(0, 2, 4, 7, -2)$.

The final scan finds no improvement

Edge	Target	Candidate
$t \to x$	4	7
$t \to y$	7	10
$t \to z$	-2	-2
$x \to t$	2	2
$y \to x$	4	4
$y \to z$	-2	16
$z \to x$	4	5
$z \to s$	0	0
$s \to t$	2	6
$s \to y$	7	7

For every edge, the candidate is at least the current target estimate. The final scan only tests; it performs no updates.

Return distances and parents

The final parent route is

$$s \to y \to x \to t \to z,$$

with successive costs $7, 4, 2, -2$.

A negative edge is allowed. No reachable negative cycle is reported.

Why n minus 1 passes suffice

Theorem: Bellman–Ford correctness

If no negative-weight cycle is reachable from s, the algorithm returns $d[v] = \delta(s, v)$ for every vertex. Otherwise its final edge scan reports failure.

For the distance claim, assume no negative cycle is reachable from s.

Pass invariant

After pass i, every vertex having a shortest path with at most i edges has its correct distance. Some estimates may become correct earlier.

The source is correct initially. A shortest path can be chosen simple, so it has at most $n - 1$ edges. An arbitrary walk may repeat vertices and have more edges.

Thus $n - 1$ passes suffice for reachable vertices; unreachable vertices retain ∞. The next slide proves the inductive step.

Use one extra scan to test for a reachable cycle

After $n-1$ passes, test whether an edge with $d[u] \neq \infty$ still satisfies

$$d[v] > d[u] + w(u,v).$$

- If an edge is improvable, a negative cycle is reachable from s; otherwise all finite shortest distances would already be correct.
- Conversely, a reachable negative cycle keeps an estimate improvable. All cycle vertices have finite estimates after $n-1$ passes. If no cycle edge were improvable, summing $d[v] \leq d[u] + w(u,v)$ around it would give $0 \leq$ its negative total weight, a contradiction.

A cycle unreachable from s is not reported. Failure is a cycle warning, not a classification of every destination as having distance $-\infty$.

Count Bellman–Ford passes

```
distance, parent <-
    Initialize-single-source(G, s) # Theta(n), once
for pass <- 1 to |V|-1             # n-1 complete passes
    for each edge (u,v) in E       # m attempts per pass
        Relax(u, v, w,
              distance, parent)    # Theta(1)
for each edge (u,v) in E           # at most m final tests
    if distance[u] != infinity and
       distance[v] > distance[u] + w(u,v)
        return Failure-negative-cycle
return distance, parent
```

Exact work in the main loops

There are $(n-1)m$ relaxation attempts. The final scan examines every edge unless it returns failure earlier.

An edge list makes each complete pass a direct scan of the stored records.

Bellman–Ford time and space

For the displayed fixed-pass algorithm, initialization and the complete passes give

$$T(n, m) = \Theta(n + nm).$$

For $m \geq 1$, this is also $\Theta(nm)$. The n term remains necessary for an edgeless graph.

Estimates and parents use $\Theta(n)$ auxiliary space beyond the graph.

Optional early termination

A variant can stop after an entire pass makes no change. It can save work on some inputs, but does not improve the worst-case bound. This flag is not part of the displayed code.

Report graph-construction cost separately from these query costs.

Match computation to the query workload

Workload	Required answer
Single pair	One route from s to t.
Single source	Distances from s to every vertex.
Single destination	Reverse all directed edges and start at the destination.
All pairs	$\delta(i, j)$ for every ordered pair, often reused by later queries.

An all-pairs workload is not one larger source query: it asks for one single-source answer from every possible source and stores n^2 distances.

The all-pairs shortest-path problem

Input and required output

Given $G = (V, E)$ with $n = |V|$, compute

$$D[s, v] = \delta(s, v) \qquad \text{for every ordered pair } (s, v) \in V \times V.$$

Use $D[s, v] = \infty$ if unreachable, and $-\infty$ if a reachable negative cycle can also reach v.

	v_1	v_2	$\cdots$	v_n
s_1	$\delta(s_1, v_1)$	$\delta(s_1, v_2)$	$\cdots$	$\delta(s_1, v_n)$
s_2	$\delta(s_2, v_1)$	$\delta(s_2, v_2)$	$\cdots$	$\delta(s_2, v_n)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
s_n	$\delta(s_n, v_1)$	$\delta(s_n, v_2)$	$\cdots$	$\delta(s_n, v_n)$

One row is a single-source result. The complete answer has n rows and already requires $\Theta(n^2)$ output space.

Run Dijkstra from every source

```
Repeated-Dijkstra(G, w)
    for each source s in V
        distance, source_parent <- Dijkstra(G, w, s)
        D[s,*] <- distance
        parent[s,*] <- source_parent
    return D, parent
```

Each row of D answers one single-source query. Row `parent[s,*]` retains the predecessors needed to reconstruct routes from that source.

Assumption and storage

All edge weights must be non-negative. Storing distances and one parent row per source takes $\Theta(n^2)$ space.

Read the cost of repeated Dijkstra

With adjacency lists and an indexed binary heap, one run costs $O((n+m)\log n)$. There are n sources:

$$n \cdot O((n+m)\log n) = O\big(n(n+m)\log n\big).$$

Method	Sparse graph, $m = \Theta(n)$	Dense graph, $m = \Theta(n^2)$
Repeated heap Dijkstra	$O(n^2 \log n)$	$O(n^3 \log n)$
Repeated array Dijkstra	$O(n^3)$	$O(n^3)$
Floyd–Warshall	$\Theta(n^3)$	$\Theta(n^3)$

"Better" depends on the graph

Floyd–Warshall improves the displayed heap bound on dense graphs and also permits negative edges. It is not universally faster: on a sparse non-negative graph, repeated heap Dijkstra has the better asymptotic bound.

Why introduce a dedicated all-pairs algorithm?

Workloads that need the matrix

- Compare the accessibility of every pair of locations.
- Precompute graph-geodesic distances for clustering or similarity.
- Evaluate distance-based centrality or facility locations.

What Floyd–Warshall changes

- It updates all pairs together instead of solving rows independently.
- It runs in $\Theta(n^3)$ time and uses $\Theta(n^2)$ matrix space.
- It allows negative edges, provided affected negative cycles are handled.

The central idea

Reuse already solved subpaths: for each possible intermediate vertex k, test whether going from i to k and then from k to j improves $D[i, j]$.

Floyd–Warshall

Input

A weighted graph represented by an $n \times n$ initial matrix W, with a fixed vertex order.

Goal and output

Compute a distance matrix for every ordered pair. The displayed algorithm returns distances, not predecessor paths.

Applicability

Negative edges are allowed. Interpret entries as finite shortest distances only when no affecting negative cycle exists.

Strategy: progressively allow vertices as intermediate stops. Useful when many pairwise queries will reuse the matrix.

Floyd–Warshall reuses shorter subproblems

For a dense graph or an explicitly requested distance matrix, compute all pairs by dynamic programming.

Question at outer iteration k

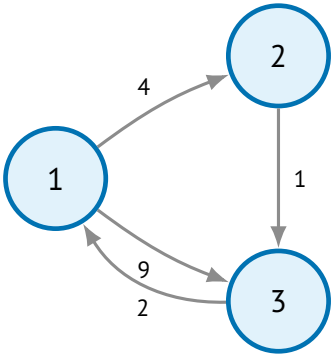
Can the best known route from i to j improve by going through vertex k?

$$D[i,j] \leftarrow \min\{D[i,j], D[i,k] + D[k,j]\}.$$

Admit intermediate vertices in a fixed order: 1, then 2, then 3 in the next example. The endpoints need not lie in the admitted set.

Fix the graph and matrix order

directed weighted graph



$W =$

initial matrix

	1	2	3
1	0	4	9
2	∞	0	1
3	2	∞	0

Rows are origins; columns are destinations, both ordered $1, 2, 3$.

A direct edge contributes its weight, a missing edge contributes ∞, and each empty path contributes diagonal value 0.

Initialize the matrix without discarding a negative loop

For this loop-free example, $W[i,i] = 0$, and $W[i,j]$ is the edge weight or ∞ when no direct edge exists.

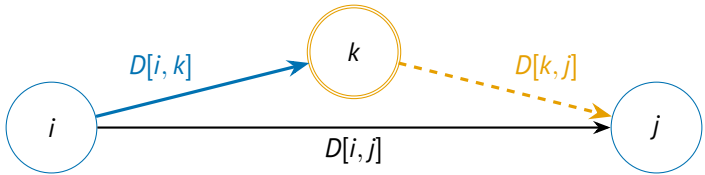
If self-loops are permitted

Use $W[i,i] = \min\{0, w(i,i)\}$ when a self-loop exists. A negative self-loop is already a negative cycle and must not be overwritten by zero.

The empty path offers cost 0, but a negative self-loop offers a smaller cost. Keep both possibilities in the initialization.

What one matrix update reads

Fix k, the intermediate vertex. Then choose an origin i and a destination j. The target is row i, column j of the matrix.



Compare two alternatives

Keep the current value $D[i,j]$, or use the candidate $D[i,k] + D[k,j]$. An operand may already describe several edges. If either operand is ∞, this candidate does not create a finite route.

Read $D[i,k]$ (solid blue) and $D[k,j]$ (dashed orange); write only the target $D[i,j]$ if the sum is smaller.

Floyd–Warshall in pseudocode

```
Floyd-Warshall(W)
    n <- number of rows of W
    D <- W
    for k <- 1 to n
        for i <- 1 to n
            for j <- 1 to n
                D[i,j] <- min(D[i,j], D[i,k] + D[k,j])
    return D
```

Here W is square and indexed from 1 to n. Use $+\infty + x = +\infty$ for finite x (and $+\infty + \infty = +\infty$).

The order of the outer loop is essential: k controls which intermediate vertices have been admitted.

What the three loops process

Keep the intermediate vertex fixed

For $k = 1$, test every origin i and every destination j. Only after all nine pairs have been processed do we move to $k = 2$.

Intermediate	Origin	Destination pairs, in order
$k = 1$	$i = 1$	$(1,1) \rightarrow (1,2) \rightarrow (1,3)$
$k = 1$	$i = 2$	$(2,1) \rightarrow (2,2) \rightarrow (2,3)$
$k = 1$	$i = 3$	$(3,1) \rightarrow (3,2) \rightarrow (3,3)$

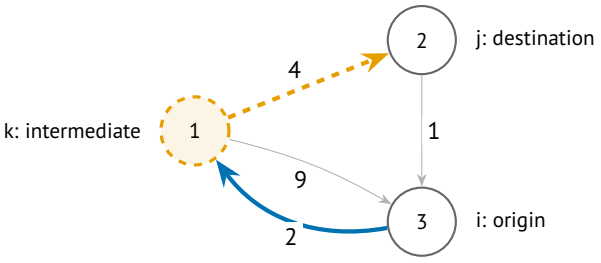
For example, $(i,j) = (2,3)$ gives $\min\{1, \infty + 9\} = 1$: no change. The next slides focus on $(3,2)$, the only improving pair for $k = 1$.

We update route costs in the matrix; we do not visit or settle graph vertices as in Dijkstra.

Through 1: compare routes from 3 to 2

Current operation: $k = 1$ (intermediate), $i = 3$ (origin), $j = 2$ (destination).

Same graph; selected subpaths



Before this update

columns: destination j

rows: origin i	1	2	3
1	0	4	9
2	∞	0	1
3	2	∞	0

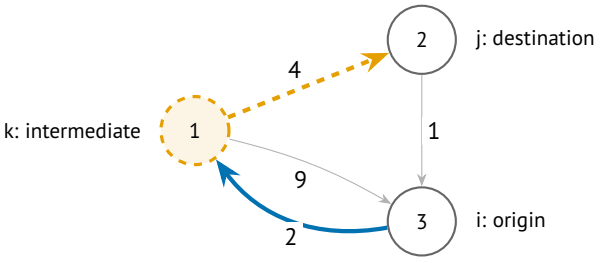
$\underbrace{D[3,2] = \infty}_{\text{double outline: current route}}$ $\underbrace{D[3,1] = 2}_{\text{solid: first subpath}}$ $\underbrace{D[1,2] = 4}_{\text{dashed: second subpath}}$.

Trace $3 \to 1 \to 2$ in the graph. Read the two matching cells, add their values, and compare the sum with the target cell.

Update the distance from 3 to 2

Current operation: $k = 1$ (intermediate), $i = 3$ (origin), $j = 2$ (destination).

Same graph; selected subpaths



After this update

columns: destination j

rows: origin i	1	2	3
1	0	4	9
2	∞	0	1
3	2	6	0

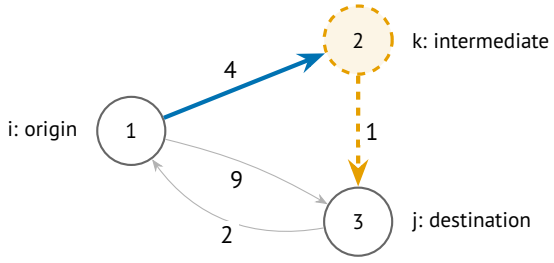
$$\underbrace{D[3,2]}_{\text{double outline: target}} \leftarrow \min\{\infty,\ \underbrace{2}_{D[3,1]} + \underbrace{4}_{D[1,2]}\} = 6.$$

Only the target cell changes; the original graph and its edge weights stay the same. This is the only improving pair in this stage; the other eight pairs are tested too.

Through 2: compare routes from 1 to 3

Current operation: $k = 2$ (intermediate), $i = 1$ (origin), $j = 3$ (destination).

Same graph; selected subpaths



Before this update

columns: destination j

rows: origin i	1	2	3
1	0	4	9
2	∞	0	1
3	2	6	0

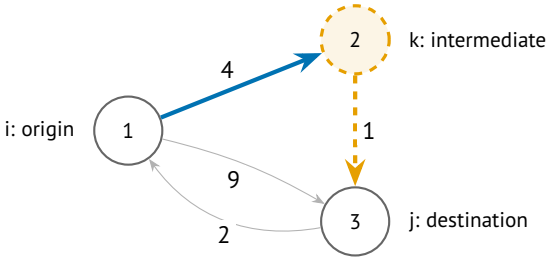
$\underbrace{D[1,3] = 9}_{\text{double outline: current route}}$ $\underbrace{D[1,2] = 4}_{\text{solid: first subpath}}$ $\underbrace{D[2,3] = 1}_{\text{dashed: second subpath}}$.

Trace $1 \to 2 \to 3$ in the graph. Read the two matching cells, add their values, and compare the sum with the target cell.

Update the distance from 1 to 3

Current operation: $k = 2$ (intermediate), $i = 1$ (origin), $j = 3$ (destination).

Same graph; selected subpaths



After this update

columns: destination j

rows: origin i	1	2	3
1	0	4	5
2	∞	0	1
3	2	6	0

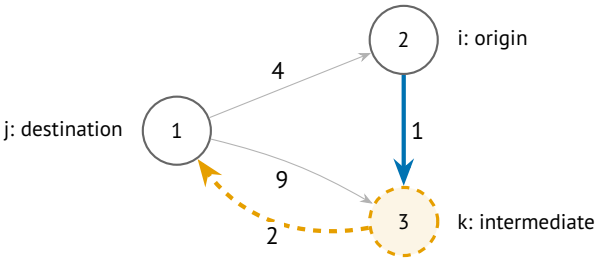
$$\underbrace{D[1,3]}_{\text{double outline: target}} \leftarrow \min\{9,\ \underbrace{4}_{D[1,2]} + \underbrace{1}_{D[2,3]}\} = 5.$$

Only the target cell changes; the original graph and its edge weights stay the same. This is the only improving pair in this stage; the other eight pairs are tested too.

Through 3: compare routes from 2 to 1

Current operation: $k = 3$ (intermediate), $i = 2$ (origin), $j = 1$ (destination).

Same graph; selected subpaths



Before this update

columns: destination j

rows: origin i	1	2	3
1	0	4	5
2	∞	0	1
3	2	6	0

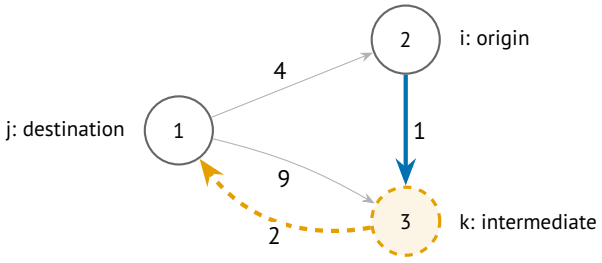
$\underbrace{D[2,1] = \infty}_{\text{double outline: current route}}$ $\underbrace{D[2,3] = 1}_{\text{solid: first subpath}}$ $\underbrace{D[3,1] = 2}_{\text{dashed: second subpath}}$.

Trace $2 \to 3 \to 1$ in the graph. Read the two matching cells, add their values, and compare the sum with the target cell.

Update the distance from 2 to 1

Current operation: $k = 3$ (intermediate), $i = 2$ (origin), $j = 1$ (destination).

Same graph; selected subpaths



After this update

columns: destination j

rows: origin i	1	2	3
1	0	4	5
2	3	0	1
3	2	6	0

$$\underbrace{D[2,1]}_{\text{double outline: target}} \leftarrow \min\{\infty,\ \underbrace{1}_{D[2,3]} + \underbrace{2}_{D[3,1]}\} = 3.$$

Only the target cell changes; the original graph and its edge weights stay the same. This is the only improving pair in this stage; the other eight pairs are tested too.

Read the final route costs from the matrix

After all three intermediate vertices have been admitted,

$$D = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ 2 & 3 & 0 & 1 \\ 3 & 2 & 6 & 0 \end{array}.$$

The original edge and the best route are different objects

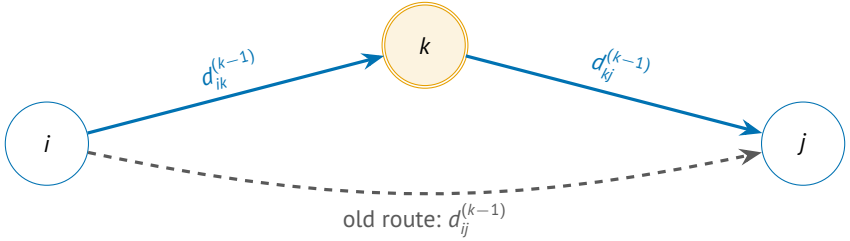
The graph still contains the direct edge $1 \to 3$ of weight 9. The matrix now stores $D[1,3] = 5$, the cost of $1 \to 2 \to 3$.

Likewise, $D[3,2] = 6$ describes $3 \to 1 \to 2$, even though no direct edge $3 \to 2$ exists.

Discuss: Read $D[2,1]$. Which route in the original graph explains this value?

Compare a route through k

Let $d_{ij}^{(k)}$ be the minimum path weight with intermediate vertices in $\{1, \ldots, k\}$. Endpoints i, j are unrestricted; $d_{ij}^{(0)} = W[i, j]$.



$$d_{ij}^{(k)} = \min\left\{ d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right\}.$$

Each arrow represents a subpath, which may contain several edges. Compare the old route with the candidate through k.

Assume no negative cycle. After stage n, $d_{ij}^{(n)} = \delta(i, j)$. The code stores stages in one matrix D, updated in place.

Why the matrix recurrence works

Assume that the graph has no negative-weight cycle.

After outer iteration k

$D[i, j]$ is the minimum weight of an i-to-j path whose intermediate vertices belong to $\{1, \ldots, k\}$.

- A best path avoids k: keep the previous value.
- Or it uses k: split it into an i-to-k part and a k-to-j part.

Choose the cheaper option. With no permitted intermediate vertices, the initialization supplies direct edges and empty paths.

After $k = n$, all vertices are permitted, so the matrix gives all-pairs distances when finite optima exist.

Count the Floyd–Warshall updates

```
Floyd-Warshall(W)
    n <- number of rows of W      # Theta(1)
    D <- W                        # Theta(n^2) to copy
    for k <- 1 to n               # n stages
        for i <- 1 to n           # n origins per stage
            for j <- 1 to n       # n destinations per origin
                D[i,j] <- min(D[i,j], D[i,k] + D[k,j])
                                  # Theta(1) per update
    return D
```

There are exactly n^3 updates, including tests that do not improve an entry.

$$\Theta(n^2) + \Theta(n^3) = \Theta(n^3).$$

A matrix gives constant-time access to each operand of the update.

All-pairs output has a substantial storage cost

The distance matrix occupies $\Theta(n^2)$ space. An optional predecessor or next-hop matrix supports path reconstruction with the same asymptotic space bound; the displayed code returns distances only.

Illustrative raw matrix budget

At $n = 10{,}000$, $n^2 = 10^8$ entries: storing 8 bytes per distance uses 0.8 GB, before other data or metadata. The triple loop performs 10^{12} updates.

Choose all-pairs computation because the workload needs it, not merely because a matrix is convenient.

A negative diagonal changes the interpretation

After the computation, a negative diagonal entry identifies a negative cycle.

Do not report every returned number as a distance

For every ordered pair (i, j) whose walk can pass through a negative cycle, $\delta(i, j) = -\infty$. A finite matrix entry is not a finite optimum for that pair.

The displayed pseudocode returns a matrix; it does not itself replace all affected entries by $-\infty$.

The cycle must be reachable from i, and j must be reachable from the cycle.

Choose a method for the weights and workload

Method	Applicability and workload	Time bound
Dijkstra, array	Non-negative; one source; dense	$O(n^2 + m)$
Dijkstra, heap	Non-negative; one source; sparse	$O((n + m) \log n)$
Bellman–Ford	Negative edges; one source; reachable cycle detection	$O(n + nm)$ total
Repeated Dijkstra	Non-negative; every source	$O(n(n + m) \log n)$
Floyd–Warshall	Negative edges; all-pairs matrix	$\Theta(n^3)$

Also consider update frequency, path reconstruction, stored output and the number of sources actually queried. Negative cycles require explicit handling.

Guided design problem: distribution cost

The same company now minimises the cost of sending one parcel from A to every location.

Assumptions

Every route has a fixed, strictly positive cost, independent of the other shipments. Journey costs add along the route.

Return one location–cost pair for every location, using ∞ when unreachable. As an extension, reconstruct a minimum-cost route.

Discuss: How do we formalise journey cost? Is this a single-source, all-pairs or minimum-spanning-tree problem?

Read the cost matrix

	A	B	C	D	E
A	0	8	3	∞	∞
B	∞	0	∞	4	∞
C	∞	2	0	9	∞
D	∞	∞	∞	0	∞
E	∞	∞	∞	∞	0

Rows are origins; columns are destinations. Finite off-diagonal entries are direct route costs. Diagonal zeros represent empty journeys; ∞ means no direct route.

Discuss: Which entries become directed edges? Do diagonal zeros create loops? Must E remain in the output?

Minimise total cost from one source

Input and feasible journeys

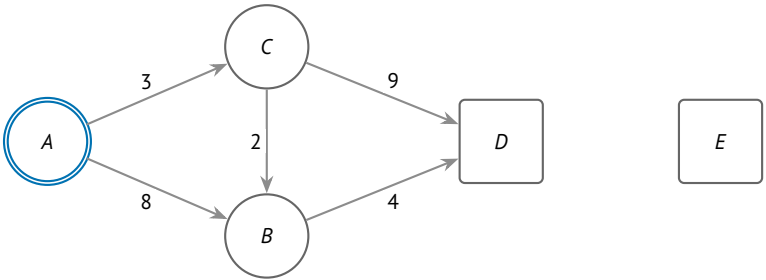
An $n \times n$ cost matrix C, source A, and directed paths beginning at A.

$$\text{cost}(p) = \sum_{(u,v) \in p} C[u, v].$$

Output

For every location, the minimum path cost from A, or ∞ if unreachable. This is a single-source problem, not all pairs or a minimum spanning tree.

Attach costs to the permitted routes



Create (u, v) with weight $C[u, v]$ for each finite off-diagonal entry. Keep all five vertices, including E.

Circles: warehouses. Rounded rectangles: points of sale. Edge labels: cost per parcel in euros.

Discuss: Would BFS work? Compare $A \to B$ with $A \to C \to B$: which uses fewer stages, and which costs less?

Fewer stages can cost more

Route	Stages	Cost
$A \to B$	1	8
$A \to C \to B$	2	$3 + 2 = 5$

The objective has changed

BFS would choose one stage, but the cheaper route costs 5. We must retain tentative total costs and allow a better route to replace an earlier one.

Discuss: Which algorithm fits positive costs and one source? What extra work would Bellman–Ford or Floyd–Warshall do?

Reuse Dijkstra for the cost objective

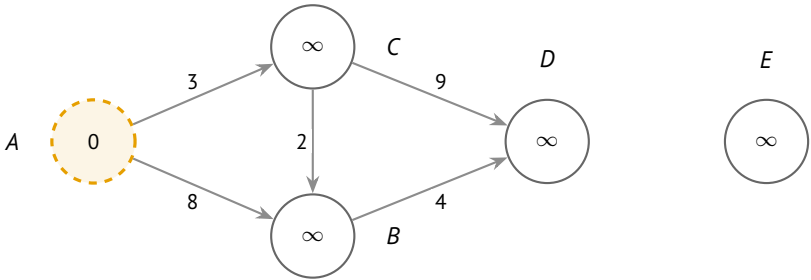
The weights are positive, hence non-negative. Dijkstra can safely settle the smallest finite tentative cost.

Adapt to the supplied matrix

Keep the matrix and use the array-based implementation, or build adjacency lists and use an indexed heap. Both implement the same Dijkstra procedure; the representation determines its cost.

Bellman–Ford is correct but its negative-edge support is unnecessary. Floyd–Warshall solves the larger all-pairs problem.

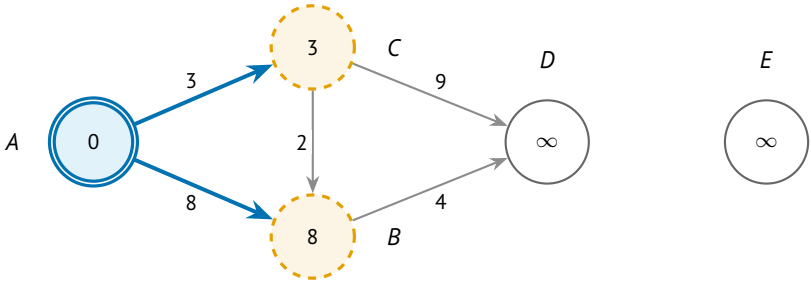
Initialise the tentative costs



$d[A] = 0$; every other distance is ∞. All parents are Nil; no vertex is settled.

Orange dashed: finite tentative cost. Double blue: settled. White: infinite estimate. Thick blue arrows: current predecessors.

Settle A and record its direct routes

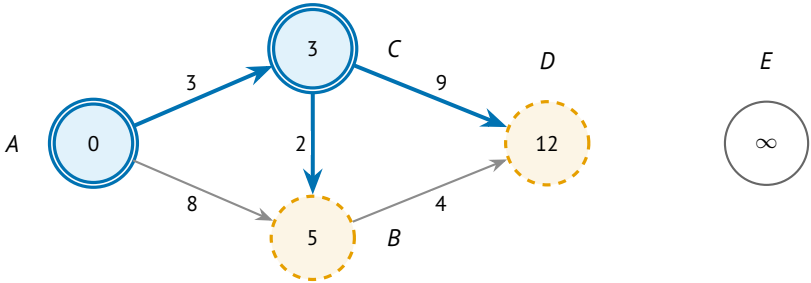


Scan row A: $d[B] = 8$, $d[C] = 3$ and both parents become A.

Which location comes next?

Choose the smallest finite unsettled estimate. Can the current cost 8 for B still improve?

Settle C and replace the route to B

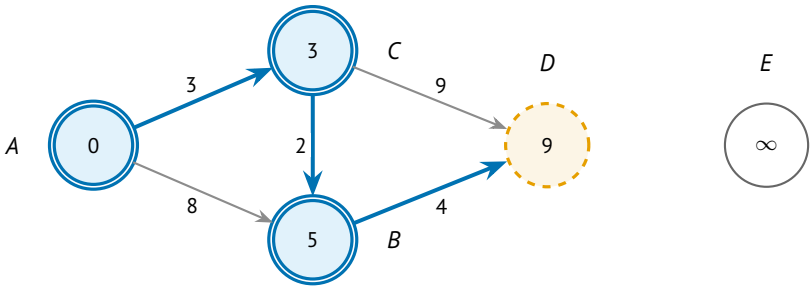


$$d[B] : 8 \longrightarrow 3 + 2 = 5, \qquad d[D] : \infty \longrightarrow 3 + 9 = 12.$$

Set parent$[B] = C$ and parent$[D] = C$. The old predecessor $A \to B$ is replaced.

Discuss: Which vertex is settled next? Can its outgoing edge improve the current cost 12 for D?

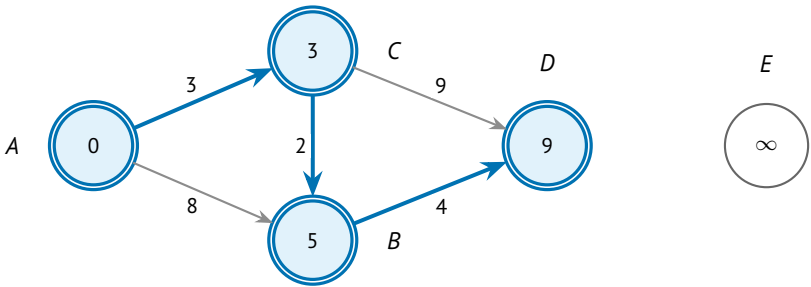
Settle B and improve D



$$d[D] : 12 \longrightarrow 5 + 4 = 9, \qquad \text{parent}[D] \leftarrow B.$$

The new route is $A \to C \to B \to D$. First discovery did not finalise either B or D.

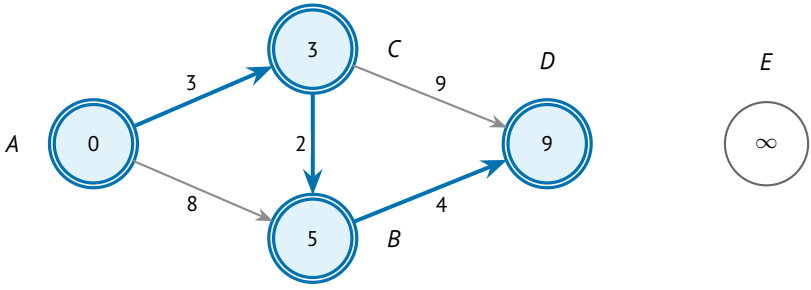
Settle D and stop before processing E



Settlement order: A, C, B, D. Row D offers no further route.

The only unsettled vertex is E, with estimate ∞. Dijkstra extracts an infinite estimate and stops before scanning its outgoing edges.

Return the minimum costs for every location



$$[(A, 0), (B, 5), (C, 3), (D, 9), (E, \infty)]$$

The final predecessor edges are $A \to C$, $C \to B$ and $B \to D$. The unreachable location is still present in the result.

Discuss: How can the parent array recover the route to D without storing a separate route for every destination?

Interpret the matrix as a graph

Create one vertex per location and an edge (u, v) for every finite off-diagonal entry $C[u, v]$, with weight $C[u, v]$.

Model, then reuse an algorithm

`Directed-weighted-graph-from-matrix` expresses this modelling step. It may retain the matrix or construct adjacency lists. It is not another shortest-path algorithm.

All costs are nonnegative, so call the Dijkstra procedure already developed. Return its parent array to retain routes compactly.

Compute and format the minimum costs

```
Minimum-distribution-costs(C, source)
    G, weight <- Directed-weighted-graph-from-matrix(C)
    distance, parent <- Dijkstra(G, weight, source)
    costs <- list of (v, distance[v]) pairs in matrix order
    return costs, parent
```

The returned list includes every location. In particular, E remains present with cost ∞: no special correction is required.

Reconstruct a route when requested

```
Minimum-cost-route(parent, source, destination)
    return Reconstruct-path(parent, source, destination)
```

Follow parent links backwards from the destination, then reverse the sequence. If it does not reach the source, report no route.

$$D \leftarrow B \leftarrow C \leftarrow A.$$

Store the shared parent tree once. Construct only the routes that are requested.

Reconstruct and check the routes

$$D \leftarrow B \leftarrow C \leftarrow A \quad \Longrightarrow \quad A \rightarrow C \rightarrow B \rightarrow D.$$

Destination	Route	Cost
A	A	0
B	$A \rightarrow C \rightarrow B$	$3 + 2 = 5$
C	$A \rightarrow C$	3
D	$A \rightarrow C \rightarrow B \rightarrow D$	$3 + 2 + 4 = 9$
E	no route	∞

Discuss: Follow the parents backwards, then add the route costs. Do they match the distances, and how do they differ from BFS?

Account for modelling, search and output

For an explicit $n \times n$ input matrix, reading and identifying its finite entries costs $\Theta(n^2)$.

Keep the matrix and use an array for the unsettled set

Dijkstra takes $\Theta(n^2)$ in the worst case, with $\Theta(n)$ auxiliary space. Formatting the list of costs takes $\Theta(n)$.

The complete solution therefore takes $\Theta(n^2)$ time. The input matrix itself occupies $\Theta(n^2)$ space.

Reconstructing one requested route takes time proportional to its number of vertices. No route reconstruction is paid for until it is requested.

Keep representation and query costs separate

With adjacency lists and an indexed binary heap, the standard Dijkstra bound is $O((n + m) \log n)$.

But this input is a matrix

Converting all matrix entries to adjacency lists first costs $\Theta(n^2)$. The complete bound is $O(n^2 + (n + m) \log n)$. A sparse result does not remove the cost of reading the supplied matrix.

The array-based solution fits the given input directly. Materialising a separate full route to every destination can require quadratic output space; parent pointers store the shared information in $\Theta(n)$.

Same network, different objective

	Minimum stages	Minimum cost
Edge contribution	One stage	Its weight
Next vertex	Oldest queued	Smallest tentative cost
Distance becomes final	First discovery	Settlement
Route to D	$A \to B \to D$	$A \to C \to B \to D$
Answer for D	2 stages	9 euros

The design decision

Formalise the objective before choosing the algorithm. The same vertices and arrows can support different computational questions.

Guided laboratory: reliable data delivery

A nightly batch travels from collection server S to analytics cluster T. If any transfer fails, the training job must restart.

Required result

For every service, return the greatest probability of successful delivery on the first attempt and a route attaining it. An unreachable service gets probability 0 and an empty route.

The input consists of transfer logs and a source service.

Discuss: What should an edge represent? What quantity describes a complete route? Would fewer transfers necessarily be better?

Data inspection

Transfer	Successes	Attempts	Estimated reliability
$S \to A$	980	1000	0.98
$S \to B$	940	1000	0.94
$S \to T$	840	1000	0.84
$A \to C$	940	1000	0.94
$A \to T$	880	1000	0.88
$B \to C$	990	1000	0.99
$C \to T$	980	1000	0.98

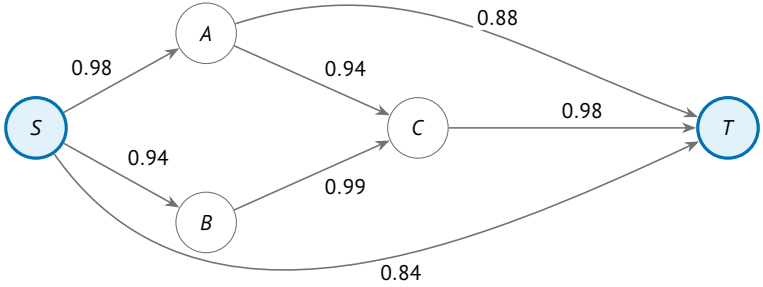
Turn transfer counts into edge estimates

Transfer	Successes	Attempts	Estimated reliability
$S \to A$	980	1000	0.98
$S \to B$	940	1000	0.94
$S \to T$	840	1000	0.84
$A \to C$	940	1000	0.94
$A \to T$	880	1000	0.88
$B \to C$	990	1000	0.99
$C \to T$	980	1000	0.98

$$r(u, v) = \frac{\text{successful transfers from } u \text{ to } v}{\text{attempted transfers from } u \text{ to } v}.$$

Estimate weights from the declared observation period, then run the graph algorithm.

Build a directed reliability graph



Each service is a vertex. An arrow is an available direct transfer; its label is a success probability.

Direction matters. No direct edge does not imply no multi-step route.

Discuss: Which assumptions let us combine the three transfers on $S \to B \to C \to T$?

A successful delivery requires every transfer

Assume stable estimates and independent transfer-success events.

$$R(S \to B \to C \to T) = 0.94 \cdot 0.99 \cdot 0.98 = 0.911988.$$

Why multiply?

The batch arrives only when all three transfers succeed. Under independence, the probability of their joint success is the product.

For $p = \langle v_0, \ldots, v_k \rangle$,

$$R(p) = \prod_{i=0}^{k-1} r(v_i, v_{i+1}).$$

Maximise this product over routes from S to each destination.

Test simple strategies against complete routes

Route to T	Transfers	Reliability
$S \to T$	1	0.84
$S \to A \to C \to T$	3	$0.98 \cdot 0.94 \cdot 0.98 = 0.902776$
$S \to B \to C \to T$	3	$0.94 \cdot 0.99 \cdot 0.98 = 0.911988$

BFS minimizes transfers, so it chooses the wrong objective. Choosing the largest first-edge probability also fails: $0.98 > 0.94$, but the best complete route starts through B.

Ideas?

Test simple strategies against complete routes

Route to T	Transfers	Reliability
$S \to T$	1	0.84
$S \to A \to C \to T$	3	$0.98 \cdot 0.94 \cdot 0.98 = 0.902776$
$S \to B \to C \to T$	3	$0.94 \cdot 0.99 \cdot 0.98 = 0.911988$

Dijkstra? Adding probabilities does not describe joint success.

Discuss: How could we turn products into sums without changing the best route?

Transform the objective without changing its optimum

Use natural logarithms and assign $w(u, v) = -\ln r(u, v)$.

$$\sum_{e \in p} w(e) = -\ln\left(\prod_{e \in p} r(e)\right) = -\ln R(p).$$

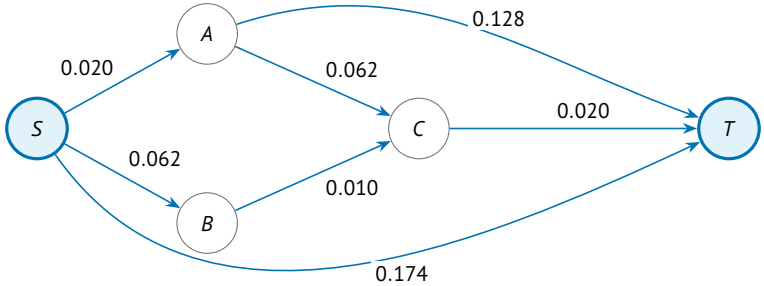
Maximise reliability by minimising cost

Because $-\ln x$ decreases as x increases, the route with maximum reliability has minimum transformed cost.

For $0 < r \leq 1$, $-\ln r \geq 0$: Dijkstra applies. An edge with $r = 1$ has cost 0; an edge with $r = 0$ is unusable and is omitted or assigned ∞.

Keep every service in the vertex set even when it has no usable edges.

Run Dijkstra on the transformed graph

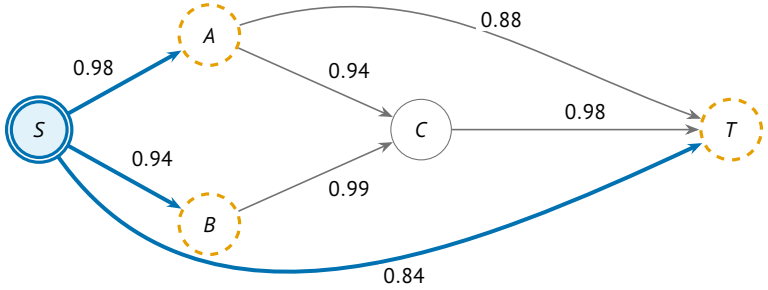


The topology is unchanged; labels now show $-\ln r$, rounded to three decimals. Compute with unrounded values.

Here $d[v] = \texttt{distance[v]}$. The tentative reliability $R(v) = e^{-d[v]}$ becomes optimal when v is settled.

Predict: After settling S, which service has the smallest transformed distance?

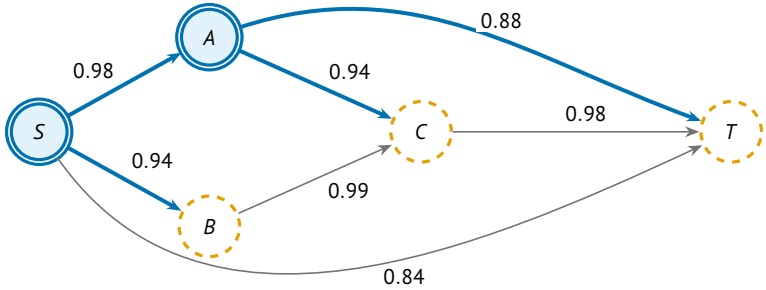
Settle S and compare its outgoing routes



$d[A] = -\ln .98 \approx .0202$, $d[B] = -\ln .94 \approx .0619$, $d[T] = -\ln .84 \approx .1744$.

A is next. The drawing retains probability labels; Dijkstra compares transformed distances. Double blue: settled. Dashed orange: finite tentative distance. White: infinite. Thick blue arrows: parents.

Settle A and create two tentative routes

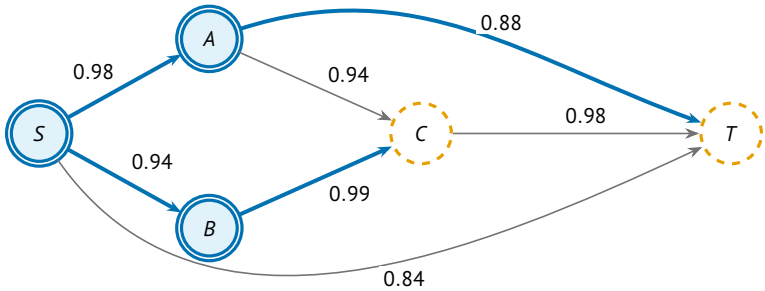


$R(C) = .98 \cdot .94 = .9212$, so $d[C] \approx .0821$.

$R(T) = .98 \cdot .88 = .8624$ improves .84; set both parents to A.

Predict: $d[B] \approx .0619 < d[C] \approx .0821$. Can processing B replace the current route to C?

Settle B and improve the route to C

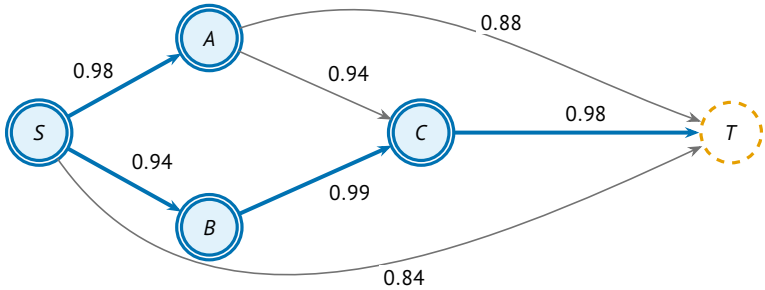


$R(C) : .9212 \longrightarrow .94 \cdot .99 = .9306$.

Thus $d[C]$ decreases from about .0821 to .0719, and parent$[C]$ changes from A to B.

C is next; T still has tentative reliability .8624.

Settle C and improve the route to T



$R(T) : .8624 \longrightarrow .9306 \cdot .98 = .911988.$

$d[T]$ decreases to $-\ln(.911988) \approx .0921$; set parent$[T] = C$.

The next extraction settles T.

Return probabilities and reconstruct routes

Service	Maximum reliability	One optimal route
S	1	$[S]$
A	.98	$S \to A$
B	.94	$S \to B$
C	.9306	$S \to B \to C$
T	.911988	$S \to B \to C \to T$

$$T \leftarrow C \leftarrow B \leftarrow S.$$

Reverse the parent chain when a route is requested. Retaining the parent tree avoids duplicating shared route prefixes.

Separate estimation, transformation and search

```
Most-reliable-routes(records, source)
    estimate each direct reliability r(u,v) from the logs
    build the directed graph G of usable transfers
    assign every edge the cost weight(u,v) <- -log(r(u,v))
    distance, parent <- Dijkstra(G, weight, source)
    convert each finite distance d[v] into reliability exp(-d[v])
    return the reliabilities and the parent tree
```

This is high-level design pseudocode. Here $d[v]$ denotes `distance[v]` and log is natural. Map ∞ to reliability 0; reconstruct routes on demand.

Account for the graph, search and requested output

For n services and m usable transfers, graph construction takes $\Theta(n+m)$ after estimation. With adjacency lists and an indexed binary heap:

$$T_{\text{Dijkstra}} = O((n+m)\log n), \qquad S_{\text{aux}} = \Theta(n).$$

Returning every route can cost more

Writing separate routes costs the sum of their lengths, which may be $\Theta(n^2)$. Returning probabilities and a parent tree uses linear state; pay for a route when it is requested.

Adding logarithms avoids underflow during repeated multiplication. Converting a very small final probability back with e^{-d} can still underflow.

Check what the probability model assumes

The algorithm optimises the supplied weights. It does not validate the empirical model.

- A short observation window may give unreliable estimates.
- Old estimates may not describe current operating conditions.
- Shared providers or data centres can make failures correlated, invalidating the independent-product model.

Discuss

If two transfers fail whenever the same data centre is unavailable, can their marginal success probabilities still be multiplied? What additional modelling information would we need?

Guided laboratory: currency exchange

A monitoring system receives one synchronized quotation snapshot. A rate $q(u, v)$ means that one unit of u buys $q(u, v)$ units of v, after proportional fees.

Input and requested output

A quotation snapshot, source currency, and positive initial amount A_0. Return the best amount and a route to every reachable currency when finite. Otherwise report a reachable profitable cycle and currencies it can affect.

Discuss: What should the graph store? How do successive conversions combine? Is a finite best answer always guaranteed?

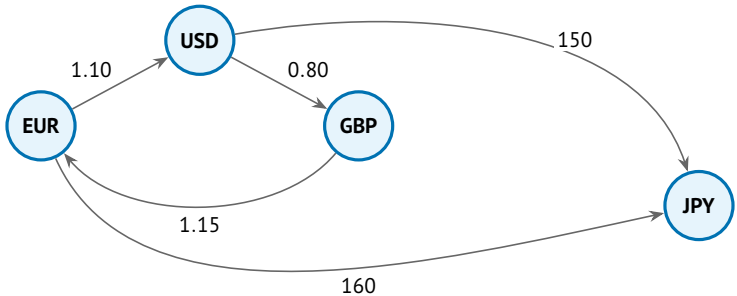
Use effective rates from one snapshot

From	To	Effective rate
EUR	USD	1.10
USD	GBP	0.80
GBP	EUR	1.15
EUR	JPY	160
USD	JPY	150

If the quoted rate is $\widehat{q}$ and the proportional fee is f, use $q = \widehat{q}(1 - f)$. These rates already include that fee.

Direction matters. Combining quotes from different times could create a cycle that never existed in one available market.

Build the graph



For example,

$$100\,\mathrm{EUR} \cdot 1.10 \frac{\mathrm{USD}}{\mathrm{EUR}} \cdot .80 \frac{\mathrm{GBP}}{\mathrm{USD}} = 88\,\mathrm{GBP}.$$