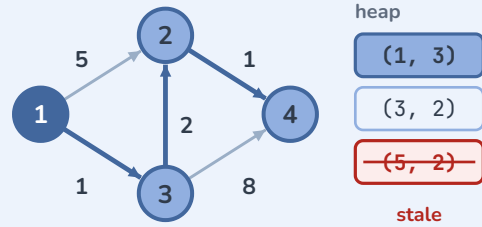


Shortest paths

Chapter 3.3 · Overall level: Intermediate

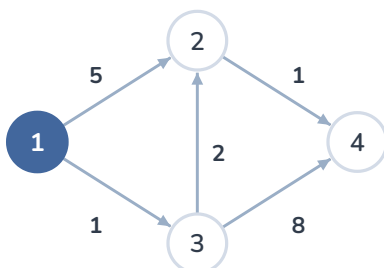


A map app does not look for the route with the fewest roads; it looks for the quickest one. *When every road has its own travel time, how do we find the cheapest way from one place to every other?*

Breadth-first search counted edges. This chapter counts weights. It introduces Dijkstra's algorithm for one starting point, the priority queue that powers it, Floyd–Warshall for all pairs at once on small graphs, and a modelling trick that turns extra rules into extra states.

3.3.1 Weighted edges and relaxation

In a **weighted graph** every edge has a cost w , such as minutes or money, and the cost of a route is the sum of its edge weights. The running example is a directed graph with edges $1 \rightarrow 2$: 5, $1 \rightarrow 3$: 1, $3 \rightarrow 2$: 2, $2 \rightarrow 4$: 1 and $3 \rightarrow 4$: 8. All weights in this chapter are nonnegative.



Fewest edges (what BFS finds)

$1 \rightarrow 2$: one edge, cost 5

Cheapest (what we want)

$1 \rightarrow 3 \rightarrow 2$: two edges, cost $1 + 2 = 3$

More edges can be cheaper when edges have different weights.

Figure 3.3.1 — BFS would report vertex 2 at one edge, but the cheapest route to it uses two edges.

BFS fails because it trusts the first discovery. The fix is to keep a **tentative label** $\text{dist}[v]$, the cost of the cheapest route to v found so far (∞ when none is known), and to improve it whenever an edge offers something better.

Improving a label along one edge is called **relaxing** the edge $u \rightarrow v$ with weight w :

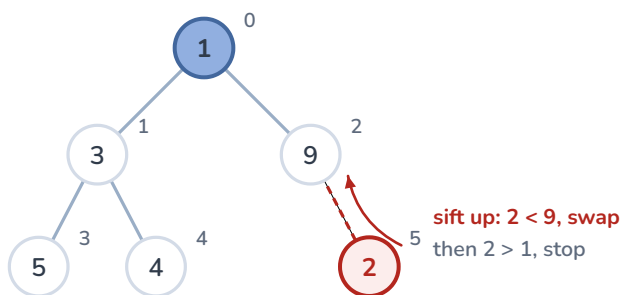
```
if dist[u] + w < dist[v]:
    dist[v] ← dist[u] + w
```

In the example, relaxing $1 \rightarrow 2$ gives $\text{dist}[2] = 5$; relaxing $1 \rightarrow 3$ and then $3 \rightarrow 2$ improves it to $1 + 2 = 3$. Every shortest-path algorithm in this chapter is a strategy for choosing which edges to relax, and in which order, so that the labels become exact without wasted work.

3.3.2 A min-priority queue

Dijkstra's algorithm needs, again and again, the vertex with the smallest tentative label. Scanning all labels each time would cost $O(n)$ per step. A **min-priority queue** does better: it stores items with priorities, *push* adds an item, and *pop* removes an item of smallest priority. Both take $O(\log k)$ time for k stored items.

The usual implementation is a **binary heap**: an array read as a tree in which every parent is no larger than its children, so the minimum sits at index 0. To push, append the item and *sift it up*, swapping with its parent while it is smaller. To pop, take index 0, move the last item to the top, and *sift it down*, swapping with the smaller child while that child is smaller. The tree has about $\log_2 k$ levels, which gives the $O(\log k)$ cost.



the same heap as an array

0	1	2	3	4	5
1	3	9	5	4	2

parent of i : $\lfloor (i - 1) / 2 \rfloor$

children of i : $2i + 1$ and $2i + 2$

the minimum is always at index 0

Figure 3.3.2 — Push on a binary heap. Only the path from the new item towards the root changes; popping uses the opposite path from the root down.

Priority queues in four languages

Language	Min-priority queue of (distance, vertex)	Watch out
Python	<code>heapq.heappush(h, (d, v)), heapq.heappop(h)</code>	tuples compare by distance first
C++	<code>priority_queue<pair<long long, int>, vector<...>, greater<>></code>	the default is a max -heap
Java	<code>PriorityQueue<Long[]> with Long.compare(a[0], b[0])</code>	<code>poll()</code> removes the minimum
JavaScript	no built-in heap: write a small binary heap	compare the distance, not the pair

JavaScript has no priority queue, so its program carries its own. Here is the heap class from the chapter's JavaScript program; Section 3.3.5 shows the rest. Items are `[distance, vertex]` pairs, and only the distance `a[i][0]` is compared. The index arithmetic is the one from Figure 3.3.2.

JavaScript (Node), part 1 of main.js: the heap

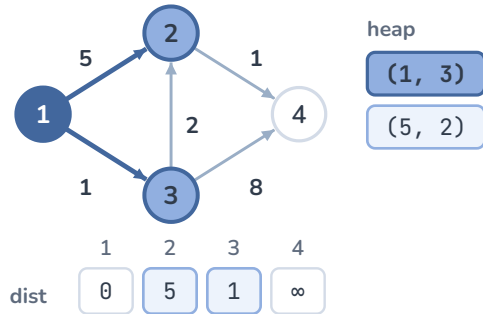
```
class MinHeap {                                // binary heap of [distance, vertex] pairs
  constructor() { this.a = []; }
  get size() { return this.a.length; }
  push(item) {
    const a = this.a;
    a.push(item);
    let i = a.length - 1;
    while (i > 0) {                             // sift up while smaller than the parent
      const p = (i - 1) >> 1;
      if (a[p][0] <= a[i][0]) break;
      [a[p], a[i]] = [a[i], a[p]];
      i = p;
    }
  }
  pop() {
    const a = this.a, top = a[0], last = a.pop();
    if (a.length > 0) {
      a[0] = last;
      let i = 0;
      while (true) {                             // sift down towards the smaller child
        const l = 2 * i + 1, r = l + 1;
        let s = i;
        if (l < a.length && a[l][0] < a[s][0]) s = l;
        if (r < a.length && a[r][0] < a[s][0]) s = r;
        if (s === i) break;
        [a[s], a[i]] = [a[i], a[s]];
        i = s;
      }
    }
    return top;
  }
}
```

Two details deserve attention. In `pop`, the last item is removed first; if that empties the heap, the removed item was the top itself and nothing has to be moved. And sifting down must compare with the *smaller* child: swapping with the larger one would put a larger value above a smaller one and break the heap order.

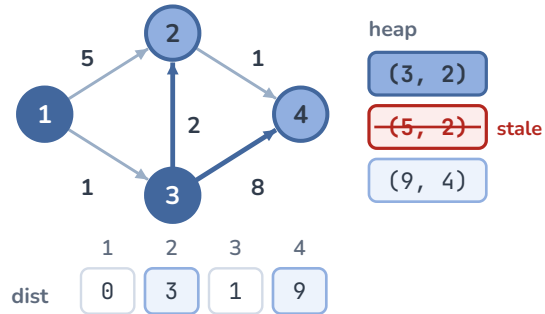
3.3.3 Dijkstra's algorithm

Start with $\text{dist}[\text{source}] = 0$, every other label ∞ , and the pair $(0, \text{source})$ in the heap. Repeatedly pop the pair with the smallest distance. The vertex is now **settled**: its label is final. Relax all edges leaving it, and push a new pair for every label that improves.

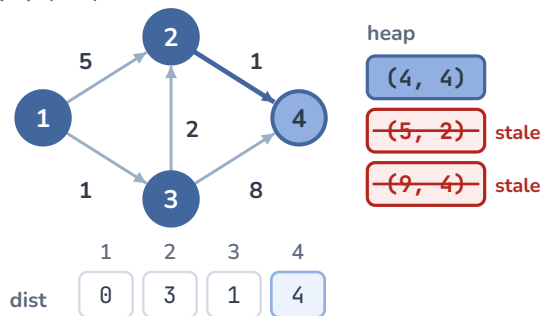
pop (0, 1): settle 1



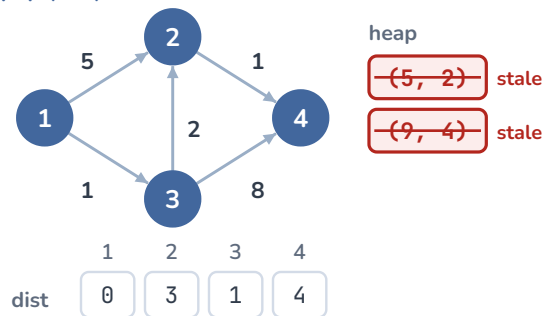
pop (1, 3): settle 3



pop (3, 2): settle 2



pop (4, 4): settle 4



● settled ● tentative thick arrows: edges just relaxed · shaded dist: label just improved · heap smallest first

Figure 3.3.3 — A shorter route may improve a tentative label before that vertex is settled. Vertex positions stay fixed; the stale entry $(5, 2)$ is crossed out but left in the heap.

Look at vertex 2. After settling 1 it has label 5 and the heap holds $(5, 2)$. Settling 3 relaxes $3 \rightarrow 2$ and improves the label to 3, so $(3, 2)$ is pushed. A heap cannot cheaply find and change the old pair, so $(5, 2)$ simply stays behind. It is now **stale**: it describes an outdated route.

Skipping stale entries

When a pair (d, u) is popped and $d \neq \text{dist}[u]$, a better route to u was found after this pair was pushed, and u has already been settled with the smaller value. The pair is ignored. In the example, $(5, 2)$ and $(9, 4)$ are popped last and both skipped, leaving the final distances $[0, 3, 1, 4]$. This “lazy deletion” keeps the code short; the heap may hold several pairs for one vertex, but at most one per successful relaxation, so never more than $m + 1$.

3.3.4 Why the smallest label is final

Claim: when a pair (d, u) with $d = \text{dist}[u]$ is popped, d is the true distance to u . Suppose instead some route P to u is cheaper than d . P starts at the settled source; let y be the first vertex on P that is not yet settled and x the vertex before it. When x was settled, the edge $x \rightarrow y$ was relaxed, so $\text{dist}[y]$ is at most the cost of P up to y . Because weights are **nonnegative**, the rest of P cannot make it cheaper, so $\text{dist}[y] \leq \text{cost}(P) < d$, and a pair with priority $\text{dist}[y]$ is in the heap. Then the heap would have returned that pair before (d, u) , a contradiction. (If y is u itself, $\text{dist}[u] < d$ contradicts $d = \text{dist}[u]$ directly.)

The argument uses nonnegative weights at exactly one point, and without them it breaks. With edges $1 \rightarrow 2$: 2, $1 \rightarrow 3$: 3 and $3 \rightarrow 2$: -2, vertex 2 is popped with label 2 and treated as finished, but the route through 3 costs 1. A version that never revisits a settled vertex prints 2. The lazy version re-pushes vertex 2 and happens to repair this tiny case, but on other graphs it can need an enormous number of repairs. Either way, the usual guarantee does not apply with a negative edge, even when the graph has no negative cycle.

Pseudocode

Dijkstra with a binary heap: $O(n + m \log m)$ time, $O(n + m)$ memory

```

dist[1..n] ← ∞; dist[s] ← 0; heap ← [(0, s)]
while the heap is not empty:
    (d, u) ← pop the pair with the smallest d
    if d ≠ dist[u]: continue           (stale entry)
    for each edge u → v with weight w:
        if d + w < dist[v]:
            dist[v] ← d + w
            push (dist[v], v)

```

3.3.5 Code in four languages

Each program solves the first practice problem, **Directed travel times**. The first line holds n and m , then m lines “ $u\ v\ w$ ” describe directed edges $u \rightarrow v$ with weight w . The program prints the distances from vertex 1 to vertices 1, ..., n on one line, with -1 for unreachable vertices.

Sample input	Sample output
<pre>4 5 1 2 5 1 3 1 3 2 2 2 4 1 3 4 8</pre>	<pre>0 3 1 4</pre>

The sample is the running example, so the output matches Figure 3.3.3. The programs follow the pseudocode line for line.

Python 3

```
import sys
import heapq

def main():
    data = sys.stdin.buffer.read().split()
    n, m = int(data[0]), int(data[1])
    adj = [[] for _ in range(n + 1)]
    for k in range(m):
        u, v, w = int(data[2 + 3 * k]), int(data[3 + 3 * k]), int(data[4 + 3 * k])
        adj[u].append((v, w))          # directed edge u -> v
    INF = float("inf")
    dist = [INF] * (n + 1)
    dist[1] = 0
    heap = [(0, 1)]                    # (tentative distance, vertex)
    while heap:
        d, u = heapq.heappop(heap)      # smallest distance first
        if d != dist[u]:
            continue                    # stale entry: u improved later
        for v, w in adj[u]:
            if d + w < dist[v]:          # relax the edge u -> v
                dist[v] = d + w
                heapq.heappush(heap, (dist[v], v))
    print(" ".join(str(x) if x != INF else "-1" for x in dist[1:]))

main()
```

C++17

```

#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    vector<vector<pair<int, int>>> adj(n + 1);    // (to, weight)
    for (int k = 0; k < m; k++) {
        int u, v, w;
        cin >> u >> v >> w;
        adj[u].push_back({v, w});            // directed edge u -> v
    }
    const long long INF = LLONG_MAX;          // never add to INF
    vector<long long> dist(n + 1, INF);
    // greater<> turns the default max-heap into a min-heap
    priority_queue<pair<long long, int>, vector<pair<long long, int>>,
        greater<>> heap;
    dist[1] = 0;
    heap.push({0, 1});
    while (!heap.empty()) {
        auto [d, u] = heap.top();
        heap.pop();
        if (d != dist[u]) continue;          // stale entry
        for (auto [v, w] : adj[u]) {
            if (d + w < dist[v]) {            // relax the edge u -> v
                dist[v] = d + w;
                heap.push({dist[v], v});
            }
        }
    }
    for (int v = 1; v <= n; v++)
        cout << (dist[v] == INF ? -1 : dist[v]) << (v < n ? ' ' : '\n');
}

```

Unreachable vertices, wide sums and undirected graphs

A vertex that no route reaches keeps its label ∞ ; print it as -1 if the problem asks for that. Choose the sentinel carefully. The programs relax only from popped vertices, whose labels are finite, so they never compute $\infty + w$. Code that adds a weight to an “infinity” such as `INT_MAX` overflows and turns it into a large negative number that looks like a wonderful route.

Distances grow beyond the edge weights: a path of 199,999 edges of weight 10^6 costs about 2×10^{11} , which does not fit in a 32-bit integer. Use `long long` in C++ and `long` in Java; JavaScript Numbers are exact below about 9×10^{15} .

For an undirected graph, insert each edge in both directions, exactly as in Chapter 3.1; the algorithm itself does not change. Every edge causes at most one push, so there are at most $m + 1$ pushes and pops, each $O(\log m)$, plus $O(n)$ to set up the labels.

Java 17

```

import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        StreamTokenizer in = new StreamTokenizer(
            new BufferedReader(new InputStreamReader(System.in)));
        in.nextToken(); int n = (int) in.nval;
        in.nextToken(); int m = (int) in.nval;
        List<List<int[]>> adj = new ArrayList<>();
        for (int v = 0; v <= n; v++) adj.add(new ArrayList<>());
        for (int k = 0; k < m; k++) {
            in.nextToken(); int u = (int) in.nval;
            in.nextToken(); int v = (int) in.nval;
            in.nextToken(); int w = (int) in.nval;
            adj.get(u).add(new int[] {v, w});           // directed edge u -> v
        }
        final long INF = Long.MAX_VALUE;               // never add to INF
        long[] dist = new long[n + 1];
        Arrays.fill(dist, INF);
        // entries {distance, vertex}; the smallest distance comes out first
        PriorityQueue<long[]> heap =
            new PriorityQueue<>((a, b) -> Long.compare(a[0], b[0]));
        dist[1] = 0;
        heap.add(new long[] {0, 1});
        while (!heap.isEmpty()) {
            long[] top = heap.poll();
            long d = top[0];
            int u = (int) top[1];
            if (d != dist[u]) continue;                 // stale entry
            for (int[] e : adj.get(u)) {
                if (d + e[1] < dist[e[0]]) {            // relax the edge u -> v
                    dist[e[0]] = d + e[1];
                    heap.add(new long[] {dist[e[0]], e[0]});
                }
            }
        }
        StringBuilder out = new StringBuilder();
        for (int v = 1; v <= n; v++)
            out.append(dist[v] == INF ? -1 : dist[v]).append(v < n ? " " : "\n");
        System.out.print(out);
    }
}

```

JavaScript (Node), part 2 of main.js (part 1 is the heap class in Section 3.3.2)

```
const data = require("fs").readFileSync(0, "utf8").trim().split(/\s+/).map(Number);
const n = data[0], m = data[1];
const adj = Array.from({ length: n + 1 }, () => []);
for (let k = 0; k < m; k++) {
    adj[data[2 + 3 * k]].push([data[3 + 3 * k], data[4 + 3 * k]]); // u -> v, w
}
const dist = new Array(n + 1).fill(Infinity);
dist[1] = 0;
const heap = new MinHeap();
heap.push([0, 1]);
while (heap.size > 0) {
    const [d, u] = heap.pop();
    if (d !== dist[u]) continue; // stale entry
    for (const [v, w] of adj[u]) {
        if (d + w < dist[v]) { // relax the edge u -> v
            dist[v] = d + w;
            heap.push([dist[v], v]);
        }
    }
}
console.log(dist.slice(1).map((x) => (x === Infinity ? -1 : x)).join(" "));
```

3.3.6 All pairs on small graphs: Floyd–Warshall

Some problems ask for distances between many pairs of vertices. Running Dijkstra from every vertex works, but for small dense graphs a three-line method is simpler. Keep a matrix $d[i][j]$: 0 on the diagonal, the weight of the cheapest direct edge (parallel edges: keep the minimum), and ∞ elsewhere. Then allow the vertices, one at a time, to serve as stopovers:

```
for k from 1 to n:                (k must be the outermost loop)
    for i from 1 to n:
        for j from 1 to n:
            d[i][j] ← min(d[i][j], d[i][k] + d[k][j])    (skip if either is ∞)
```

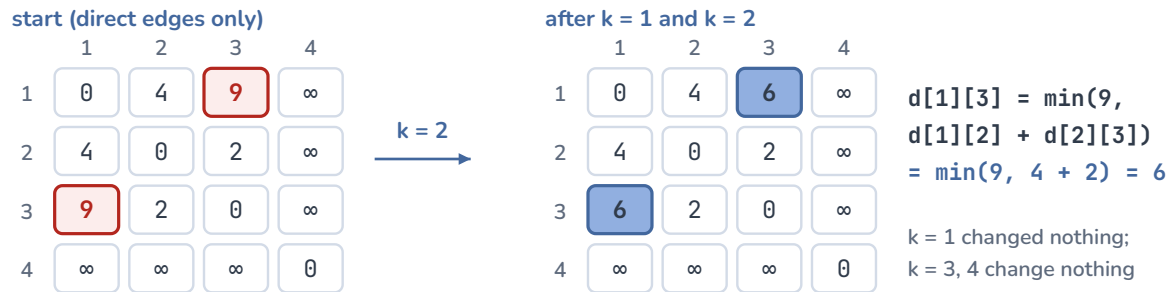


Figure 3.3.4 — Allowing vertex 2 as a stopover shortens $1 \rightarrow 3$ from 9 to 6. Unreachable pairs stay at ∞ and are reported as -1 .

Invariant. After round k , $d[i][j]$ is the cheapest route from i to j whose intermediate vertices all lie in $\{1, \dots, k\}$. Round k keeps it true: a cheapest such route either avoids k as a stopover, and was already counted, or passes through k once, splitting into a route $i \rightarrow k$ and a route $k \rightarrow j$ that use only $\{1, \dots, k-1\}$ inside, and both were exact after round $k-1$. After round n every vertex is allowed, so the matrix holds the true distances. Putting k anywhere but outermost breaks this invariant.

The cost is n^3 updates and n^2 memory: for $n = 300$ that is 27 million simple steps, after which every question is a table lookup. For 200,000 vertices it is hopeless, and Dijkstra from the needed sources is the tool.

3.3.7 One discount: an extra state in the graph

Now a new rule: on the way from 1 to n you may halve the weight of *one* edge you travel, paying $\lfloor w/2 \rfloor$ instead of w . Plain labels per vertex are not enough, because arriving at a vertex with the discount still available is worth more than arriving without it. Two situations need two labels.

So make the situation part of the vertex. The **state** $(v, 0)$ means “at v , discount unused” and $(v, 1)$ means “at v , discount used”. Each edge $u \rightarrow v$ with weight w becomes three transitions: $(u, 0) \rightarrow (v, 0)$ and $(u, 1) \rightarrow (v, 1)$ with cost w , and the **bridge** $(u, 0) \rightarrow (v, 1)$ with cost $\lfloor w/2 \rfloor$. No transition leads from 1 back to 0.

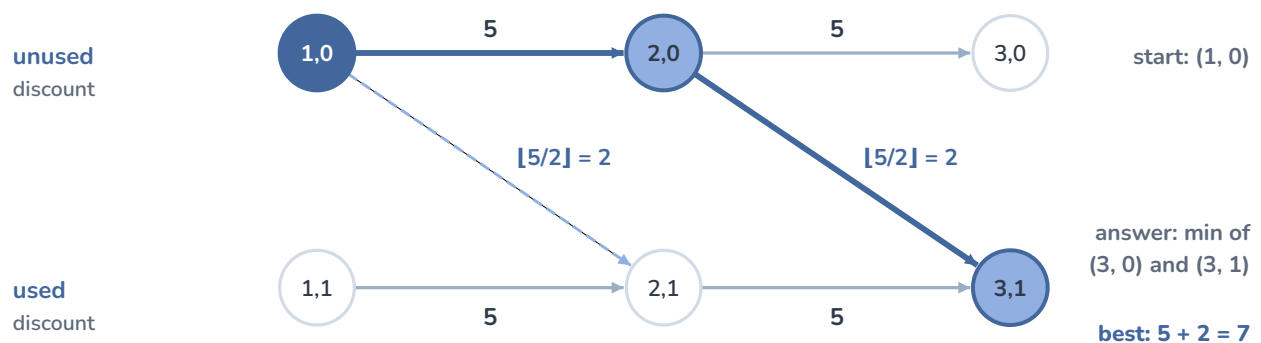


Figure 3.3.5 — The state records whether the discount has already been used; paths cannot move back to unused.

Run Dijkstra from $(1, 0)$ on this graph of $2n$ states and $3m$ transitions; the answer is the smaller of the labels of $(n, 0)$ and $(n, 1)$, since the discount is optional. Why it is right: every route that discounts at most one edge corresponds to exactly one path in the layered graph with the same cost (it switches layers at the discounted edge, or never), and every path corresponds back to such a route, because the layer can change only once. All transition costs are nonnegative, so Dijkstra applies. In the figure, $1 \rightarrow 2$ at full price and then the discounted $2 \rightarrow 3$ costs $5 + 2 = 7$.

The same trick handles many “extra permission” rules: a limited number of free tickets (layers $0, 1, \dots, k$), carrying a key, or the time modulo something. The price is a graph multiplied by the number of situations, so keep that number small.

3.3.8 Choosing a method

Graph and question	Method	Time
Unweighted, or all weights equal; one source	BFS (Chapter 3.1)	$O(n + m)$
Nonnegative weights; one source	Dijkstra with a heap	$O(n + m \log m)$
Nonnegative weights; all pairs, n up to a few hundred	Floyd–Warshall	$O(n^3)$
Extra rule such as one discount	Dijkstra on states (v, extra)	states $\times$ edges $\times$ log
Some negative weights	Bellman–Ford (further reading)	$O(nm)$

3.3.9 Common mistakes

- **Wrong heap order.** C++ `priority_queue` pops the *largest* item unless given `greater<>`. With the wrong order Dijkstra still terminates but settles vertices too early and prints wrong distances.
- **Overflow at the infinity sentinel.** Never compute $\infty + w$. Relax only from popped vertices, or test for ∞ first, and keep distances in 64-bit integers.
- **Forgetting the used/unused state.** One label per vertex cannot tell “here, discount still available” from “here, discount spent”.
- **Not skipping stale entries.** Reprocessing an outdated pair repeats work, and code that relaxes with the popped d must be sure it is current.
- **Dijkstra with negative edges, or Floyd–Warshall with k inside.** Both break the invariant that makes the method correct.

3.3.10 Where these methods stop

Negative weights break Dijkstra's argument, as Section 3.3.4 showed. The Bellman–Ford algorithm, which relaxes every edge in up to $n - 1$ rounds, handles them in $O(nm)$ time and is optional further reading. With a **negative cycle** reachable on the way, “shortest” stops making sense, because going round the cycle again always lowers the cost: with $2 \rightarrow 3$: -3 and $3 \rightarrow 2$: 1 , each lap $2 \rightarrow 3 \rightarrow 2$ changes the total by -2 , forever. Floyd–Warshall needs n^2 memory, impossible for 200,000 vertices. And the state trick multiplies the graph: k discounts mean $k + 1$ layers.

To print a route and not just its cost, store $\text{parent}[v] \leftarrow u$ whenever a relaxation improves $\text{dist}[v]$, then follow the parents back from the target.

Exercises

Difficulty: ● quick check, ●● needs an extra idea. Try each one before reading the answers.

1. ● May Dijkstra's algorithm be used with a negative edge if the graph has no negative cycle?
2. ● Why does the program skip a popped pair (d, u) when $d \neq \text{dist}[u]$?
3. ● Which method would you use: (a) fewest moves in a maze grid; (b) cheapest route from one city among 200,000 cities; (c) many questions about cheapest routes between any two of 150 cities?
4. ●● Run Dijkstra from 1 on the directed edges $1 \rightarrow 2$: 4, $1 \rightarrow 3$: 1, $3 \rightarrow 2$: 2, $2 \rightarrow 5$: 1, $3 \rightarrow 4$: 5, $4 \rightarrow 5$: 1, $2 \rightarrow 4$: 1. Give the order in which vertices are settled, the final distances, and the stale pairs.
5. ●● Three vertices have undirected edges $1-2$: 3, $2-3$: 4 and $1-3$: 10. Write the Floyd–Warshall matrix at the start and after round $k = 2$.
6. ●● Directed edges $1 \rightarrow 2$: 10, $2 \rightarrow 3$: 3, $1 \rightarrow 3$: 14; one edge may be halved (rounded down). What is the cheapest cost from 1 to 3, and which states does the best path use?

Answers and explanations

Read these after attempting the exercises.

1. Its usual correctness guarantee does not apply. The proof that a popped label is final needs nonnegative weights; with a negative edge a settled vertex can still improve. Use Bellman–Ford instead.
2. The pair describes an outdated tentative route: a cheaper one was found later and the vertex has already been settled with it. Processing the pair again would only repeat work.
3. (a) BFS, since every move costs the same; (b) Dijkstra with a heap; (c) Floyd–Warshall once, then answer each question by a table lookup ($150^3 \approx 3.4$ million steps).
4. Settled in the order **1, 3, 2, 4, 5** (4 and 5 both have label 4; either may come first). Distances: **0, 3, 1, 4, 4**.
4. Stale pairs: (4, 2), left behind when $3 \rightarrow 2$ improved vertex 2 to 3, and (6, 4), left behind when $2 \rightarrow 4$ improved vertex 4 to 4.
5. Start: rows 1, 2, 3 are [0, 3, 10], [3, 0, 4], [10, 4, 0]. Round $k = 1$ changes nothing. Round $k = 2$ sets $d[1][3] = d[3][1] = \min(10, 3 + 4) = 7$.
6. Without a discount the best is 13 ($1 \rightarrow 2 \rightarrow 3$). Halving $1 \rightarrow 2$ gives $5 + 3 = 8$; halving $1 \rightarrow 3$ gives 7. The answer is **7**, using the bridge $(1, 0) \rightarrow (3, 1)$ directly.

Practice this topic

Four coding problems for this chapter are in the **Practice** tab. Open any of them in any order; the label shows difficulty within this topic.

Directed travel times

Apply

Find shortest distances from vertex 1 to all vertices of a directed nonnegative-weight graph. Print -1 for unreachable vertices.

Cheapest undirected route

Connect

Find the minimum cost from vertex 1 to vertex n in an undirected nonnegative-weight graph. Print -1 if unreachable.

All-pairs travel desk

Reason

Answer minimum-cost queries between pairs of vertices in a small undirected nonnegative-weight graph. Print -1 for an unreachable pair.

One discounted edge

Stretch

Find the cheapest directed route from 1 to n if you may discount at most one traversed edge of weight w to $\text{floor}(w/2)$. The discount is optional. Print -1 if no route exists.