Chapter 6: Shortest Paths

Algorithms

Relaxation | Dijkstra | Bellman-Ford | Negative cycles | Floyd-Warshall | A* search | Unweighted paths with BFS | Comparison

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/algorithms/slides/Chapter_06_Shortest_Paths.html

Learning Outcomes

Concepts

Distinguish single-source, all-pairs and single-pair shortest-path problems; explain **relaxation** and the predecessor (π) array.

Dijkstra and Bellman-Ford

Run both algorithms step by step on an example graph, fill in the distance and π tables, and trace the Java code.

Negative edges and cycles

Show why Dijkstra fails on negative edges; detect negative cycles with Bellman-Ford and Floyd-Warshall.

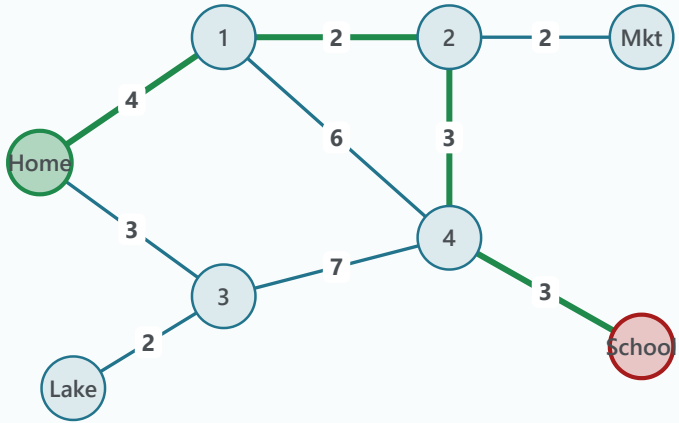
Floyd-Warshall and A*

Update the all-pairs D and Π matrices for k = 1…n; apply A*'s goal-directed search with f = g + h, and compare the algorithms.

INTRODUCTION

A Road Network Is a Graph

- Intersections are **vertices**, roads are **edges**, and road lengths or travel times are **weights**.
- The **length** of a path = the sum of its edge weights.
- A **shortest path** between two vertices is a path of minimum total weight; its length is written δ(s, v).
- Example: from Home to School, Home–1–2–4–School = 4 + 2 + 3 + 3 = **12**; Home–1–4–School has fewer edges but costs 13.



In a weighted graph, the path with the fewest edges **need not** be the shortest. Navigation, network routing (OSPF, RIP), pathfinding in games and logistics all solve this problem.

Kinds of Shortest-Path Problems

Single-source

From s to **every** vertex. Dijkstra, Bellman-Ford, BFS.

All-pairs

Every pair (u, v); the result is an n × n matrix. Floyd-Warshall.

Single-pair

Only s → t; goal-directed search. A*.

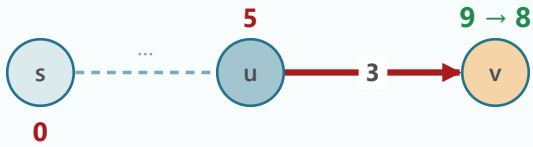
Algorithm	What it finds	Restriction / note
Dijkstra	source to all vertices	edge weights **must be nonnegative**
Bellman-Ford	source to all vertices	handles negative edges, detects negative cycles; slower than Dijkstra
Floyd-Warshall	all pairs	handles negative edges; detects negative cycles on the diagonal
A*	start to goal	speeds up search with a heuristic estimate
BFS	source to all vertices	unweighted (or equal-weight) graphs only

The Basic Operation: Relaxation

Each vertex keeps two values:

- `d[v]` : the shortest-path **estimate** so far (∞ at first, 0 at the source)
- `π[v]` : the vertex before v on that path (its predecessor)

```
RELAX(u, v, w):
  if d[u] + w(u, v) < d[v]:
    d[v] = d[u] + w(u, v)
    π[v] = u
```



d[u] + w = 5 + 3 = 8 < 9 → d[v] = 8, π[v] = u

Every algorithm in this chapter relies on relaxation; they differ in **the order** in which they relax edges and **how many times**. d[v] can only decrease and never drops below the true δ(s, v).

Properties of Shortest Paths

Optimal substructure

Every subpath of a shortest path is itself a shortest path. If s ⇝ u ⇝ v is shortest, so is s ⇝ u. Dynamic programming and greedy methods rest on this.

Triangle inequality

For every edge (u, v), $\delta(s, v) \le \delta(s, u) + w(u, v)$. Relaxation fixes edges that violate it.

Negative cycle

If a cycle of negative total weight is reachable from the source, each lap makes the path shorter: the shortest path is **undefined** ($-\infty$).

Simple paths

Without negative cycles a shortest path has no cycle, so it has at most **V − 1** edges. That is where Bellman-Ford's V − 1 passes come from.

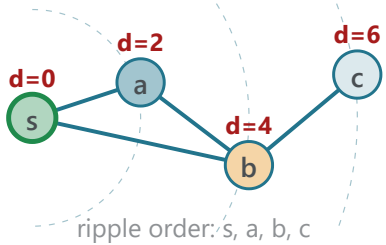
To rebuild a path, follow π from the target back to the source and list the vertices in reverse. Together, all π edges form a **shortest-path tree**.

Dijkstra's Algorithm

- Finds shortest paths from a **source** vertex to every other vertex (single-source).
- Devised by **Edsger W. Dijkstra** in 1956 and published in 1959.
- Correct only on graphs whose edge weights are **nonnegative**.
- **Greedy**: at each step it picks the not-yet-settled vertex with the **smallest** d value and settles it.
- The neighbors of the chosen vertex are relaxed.

The ripple analogy

A ripple from a stone dropped at the source reaches the vertices in order of distance: the nearest first, then the next… Dijkstra **settles** vertices in exactly this order. The set of settled vertices is called **S**.



Dijkstra: The Steps

Step 1: Choose the start vertex and set its distance to **0**; every other vertex gets **∞**.

Initialization: d[s] = 0, d[v] = ∞, π[v] = undefined, S = ∅.

Dijkstra: The Steps

Step 1: Choose the start vertex and set its distance to **0**; every other vertex gets **∞**.

Initialization: d[s] = 0, d[v] = ∞, π[v] = undefined, S = ∅.

Step 2: From the chosen vertex, compute distances to its not-yet-processed **neighbors**.

Candidate distance for neighbor v: d[u] + w(u, v).

Dijkstra: The Steps

Step 1: Choose the start vertex and set its distance to **0**; every other vertex gets **∞**.

Initialization: d[s] = 0, d[v] = ∞, π[v] = undefined, S = ∅.

Step 2: From the chosen vertex, compute distances to its not-yet-processed **neighbors**.

Candidate distance for neighbor v: d[u] + w(u, v).

Step 3: If a shorter path is found, **update** the neighbor's distance (relaxation).

If d[u] + w(u, v) < d[v], d[v] and π[v] change.

Dijkstra: The Steps

Step 1: Choose the start vertex and set its distance to **0**; every other vertex gets **∞**.

Initialization: d[s] = 0, d[v] = ∞, π[v] = undefined, S = ∅.

Step 2: From the chosen vertex, compute distances to its not-yet-processed **neighbors**.

Candidate distance for neighbor v: d[u] + w(u, v).

Step 3: If a shorter path is found, **update** the neighbor's distance (relaxation).

If d[u] + w(u, v) < d[v], d[v] and π[v] change.

Step 4: Mark the processed vertex (add it to S) and **select** the unmarked vertex with the smallest distance.

The selection uses an array or a priority queue.

Dijkstra: The Steps

Step 1: Choose the start vertex and set its distance to **0**; every other vertex gets **∞**.

Initialization: d[s] = 0, d[v] = ∞, π[v] = undefined, S = ∅.

Step 2: From the chosen vertex, compute distances to its not-yet-processed **neighbors**.

Candidate distance for neighbor v: d[u] + w(u, v).

Step 3: If a shorter path is found, **update** the neighbor's distance (relaxation).

If d[u] + w(u, v) < d[v], d[v] and π[v] change.

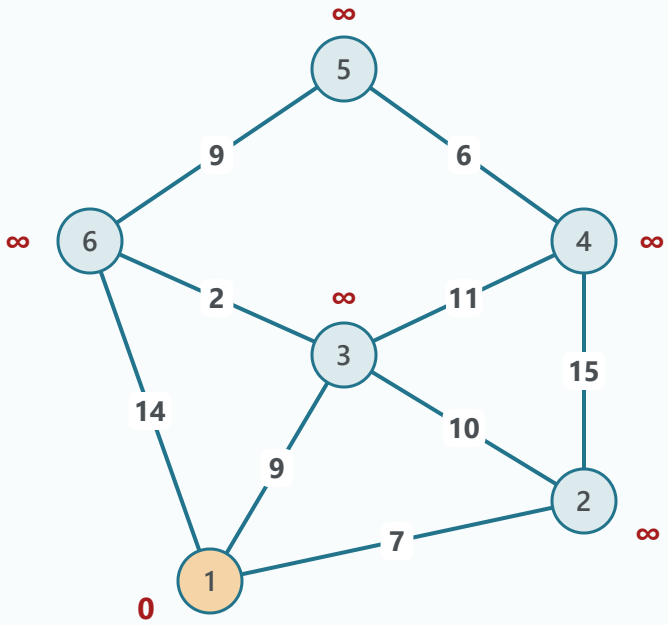
Step 4: Mark the processed vertex (add it to S) and **select** the unmarked vertex with the smallest distance.

The selection uses an array or a priority queue.

Step 5: Repeat steps 2–4 until every vertex is processed.

Unreachable vertices keep distance ∞.

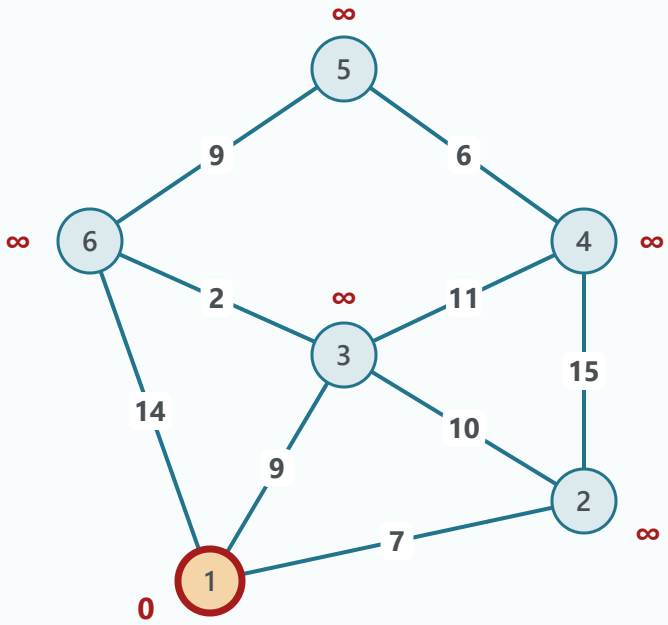
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	0	–
2	∞	–
3	∞	–
4	∞	–
5	∞	–
6	∞	–

Initialization: source **1**, d[1] = 0, all others ∞. No vertex is settled.

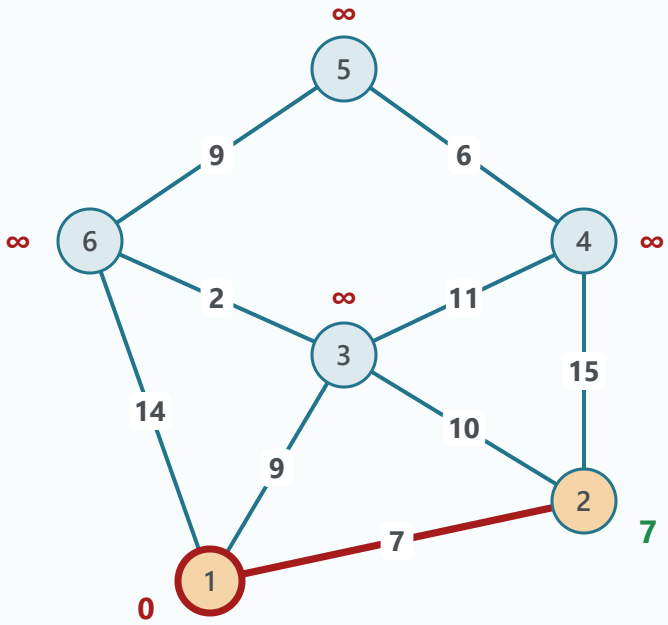
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	∞	–
3	∞	–
4	∞	–
5	∞	–
6	∞	–

Smallest d among unsettled vertices: **1** (d = 0); it is settled.

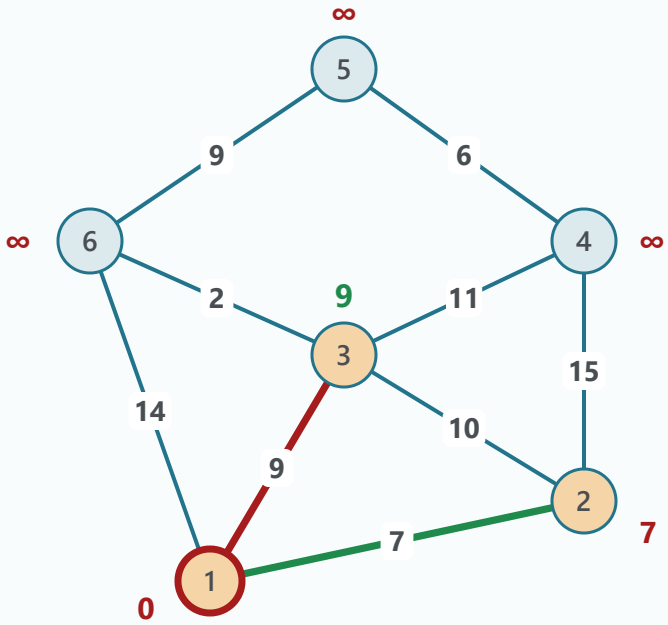
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	**7**	**1**
3	∞	–
4	∞	–
5	∞	–
6	∞	–

Edge (1, 2): d[1] + 7 = 7 < ∞ → **d[2] = 7**, π[2] = 1.

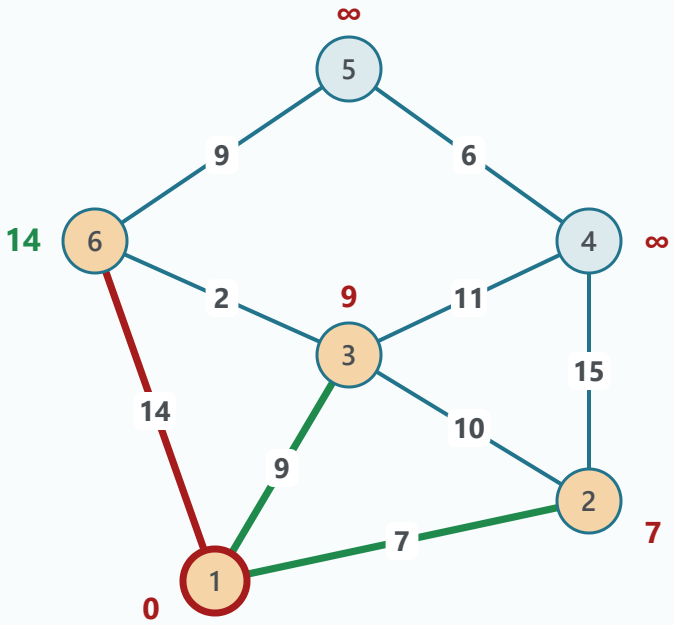
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	7	1
3	**9**	**1**
4	∞	–
5	∞	–
6	∞	–

Edge (1, 3): d[1] + 9 = 9 < ∞ → **d[3] = 9**, π[3] = 1.

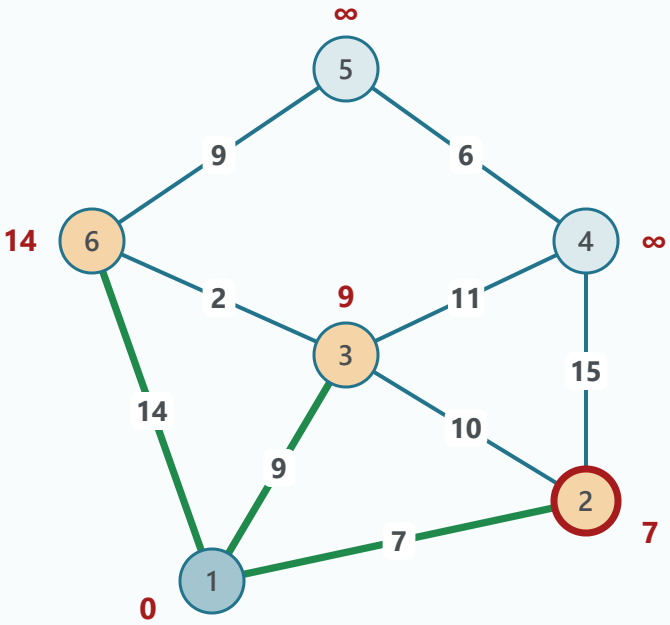
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	7	1
3	9	1
4	∞	–
5	∞	–
6	**14**	**1**

Edge (1, 6): d[1] + 14 = 14 < ∞ → **d[6] = 14**, π[6] = 1.

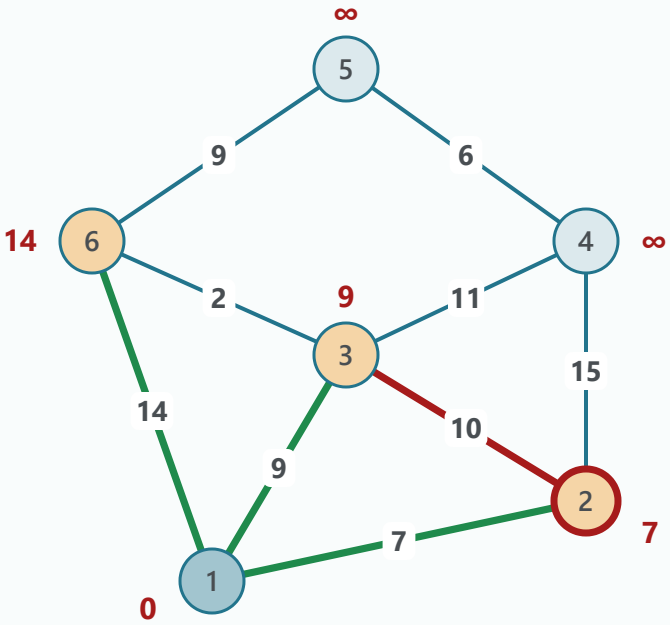
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	**7**	1
3	9	1
4	∞	–
5	∞	–
6	14	1

Smallest d among unsettled vertices: **2** (d = 7); it is settled. Settled neighbors (1) are skipped.

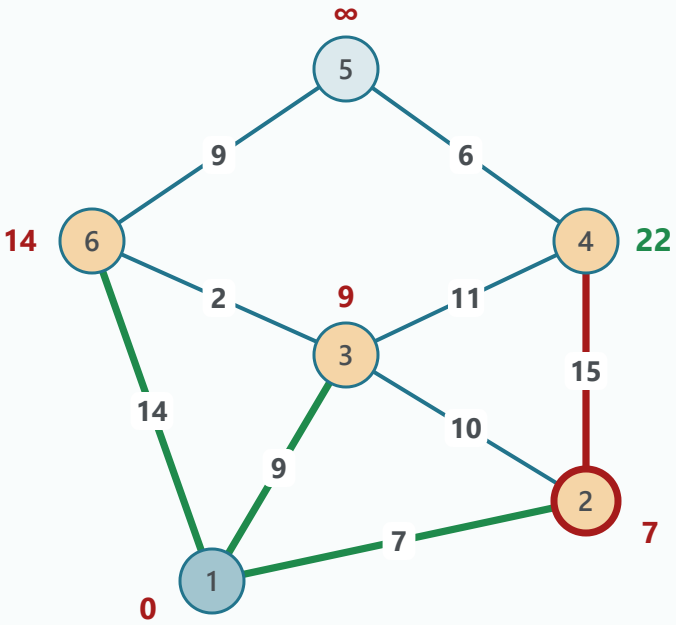
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	**7**	1
3	9	1
4	∞	–
5	∞	–
6	14	1

Edge (2, 3): d[2] + 10 = 17 ≥ 9 → no change.

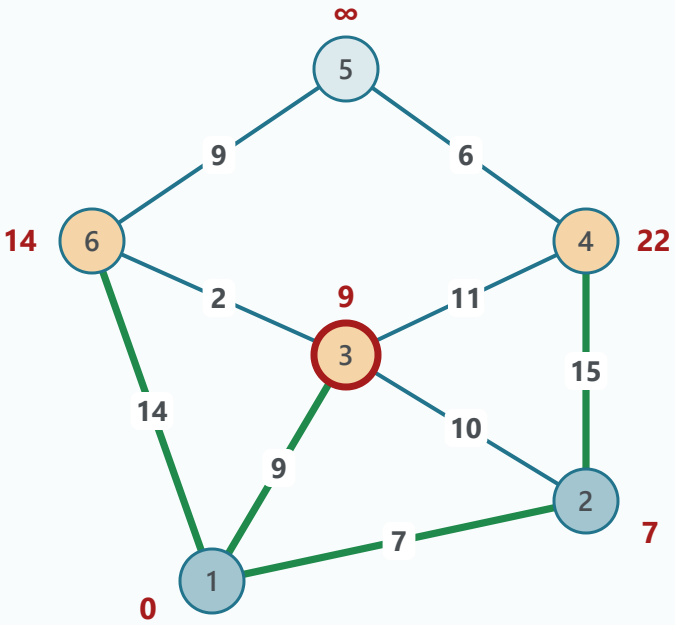
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	**7**	1
3	9	1
4	**22**	**2**
5	∞	–
6	14	1

Edge (2, 4): d[2] + 15 = 22 < ∞ → **d[4] = 22**, π[4] = 2.

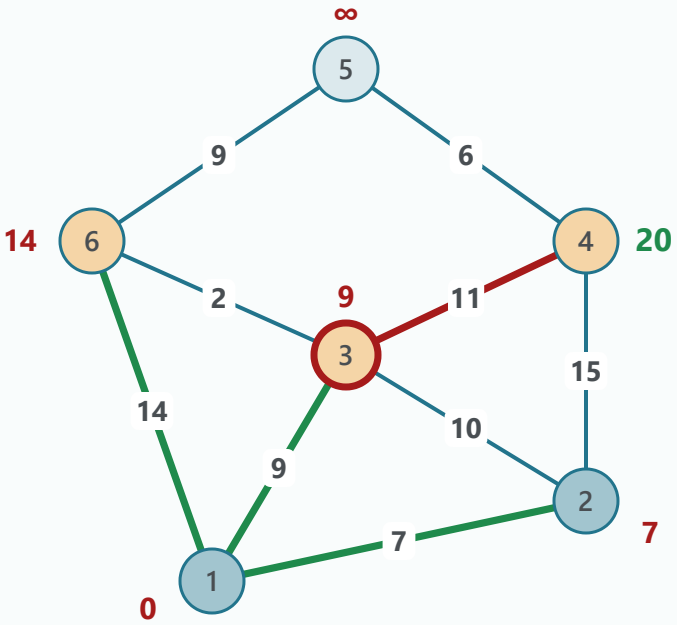
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	22	2
5	∞	–
6	14	1

Smallest d among unsettled vertices: **3** (d = 9); it is settled. Settled neighbors (1, 2) are skipped.

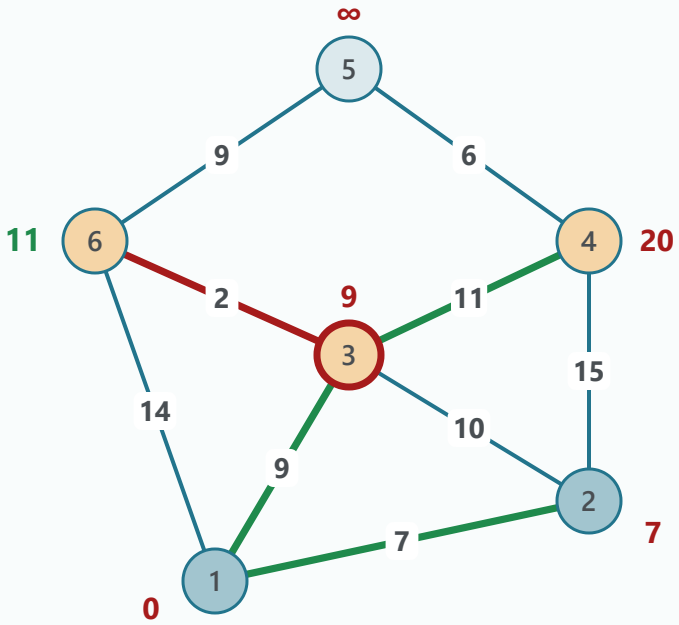
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	**20**	**3**
5	∞	–
6	14	1

Edge (3, 4): d[3] + 11 = 20 < 22 → **d[4] = 20**, π[4] = 3.

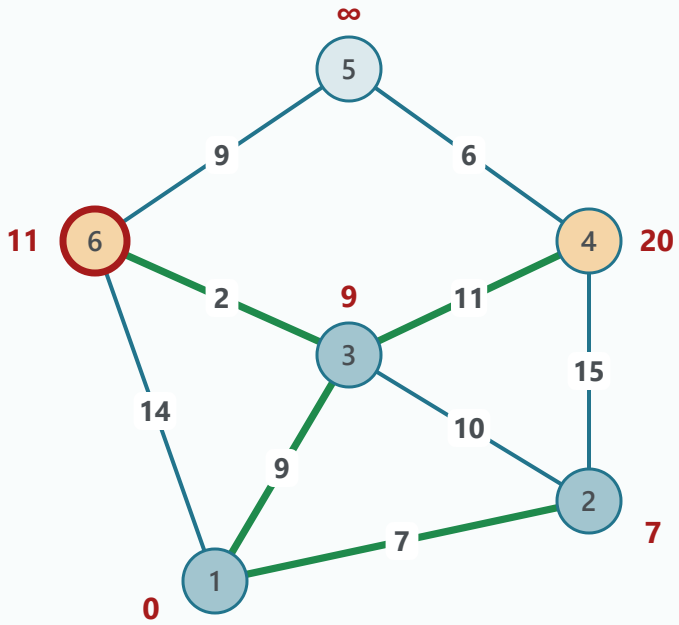
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	20	3
5	∞	–
6	**11**	**3**

Edge (3, 6): d[3] + 2 = 11 < 14 → **d[6] = 11**, π[6] = 3.

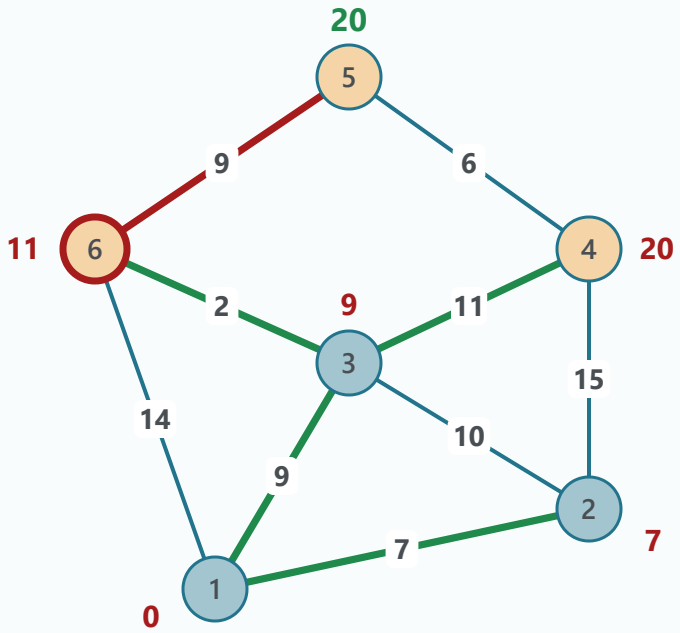
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	20	3
5	∞	–
6	**11**	3

Smallest d among unsettled vertices: **6** (d = 11); it is settled. Settled neighbors (1, 3) are skipped.

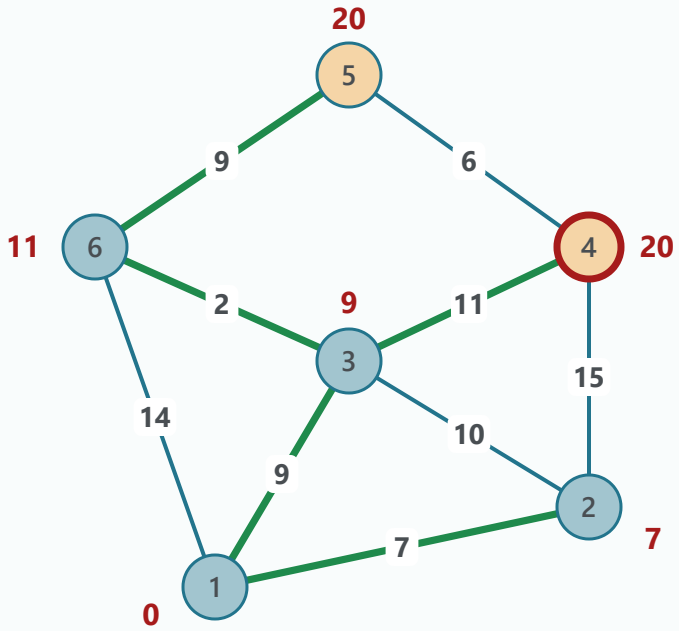
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	20	3
5	**20**	**6**
6	**11**	3

Edge (6, 5): d[6] + 9 = 20 < ∞ → **d[5] = 20**, π[5] = 6.

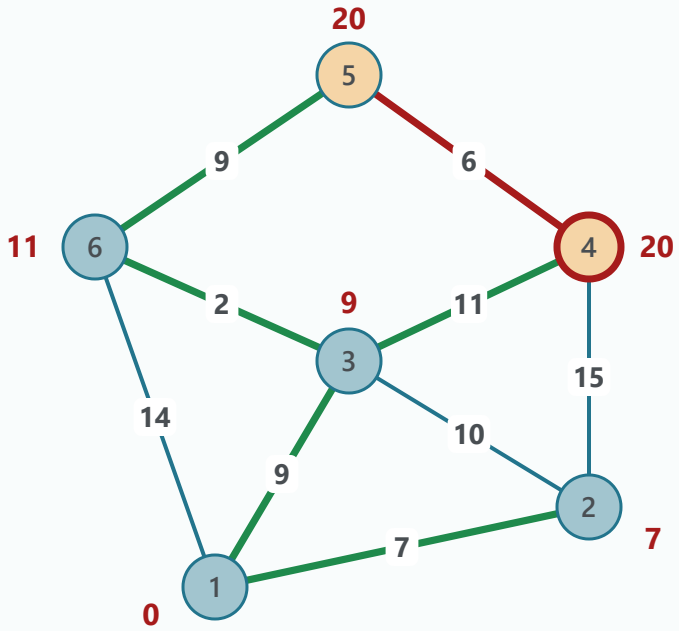
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	**20**	3
5	20	6
6	**11**	3

Smallest d among unsettled vertices: **4** (d = 20); it is settled. (4 and 5 tie at 20; the lower-numbered 4 is taken.) Settled neighbors (2, 3) are skipped.

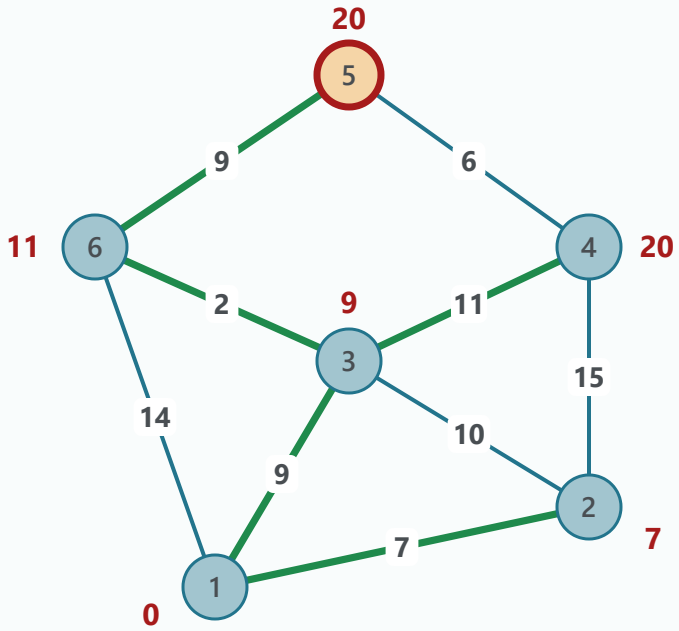
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	**20**	3
5	20	6
6	**11**	3

Edge (4, 5): d[4] + 6 = 26 ≥ 20 → no change.

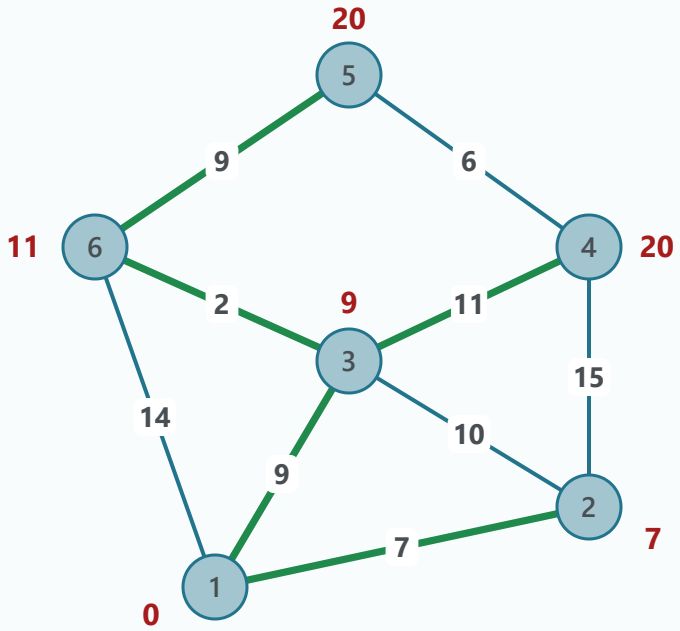
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	**20**	3
5	**20**	6
6	**11**	3

Smallest d among unsettled vertices: **5** (d = 20); it is settled. Settled neighbors (4, 6) are skipped.

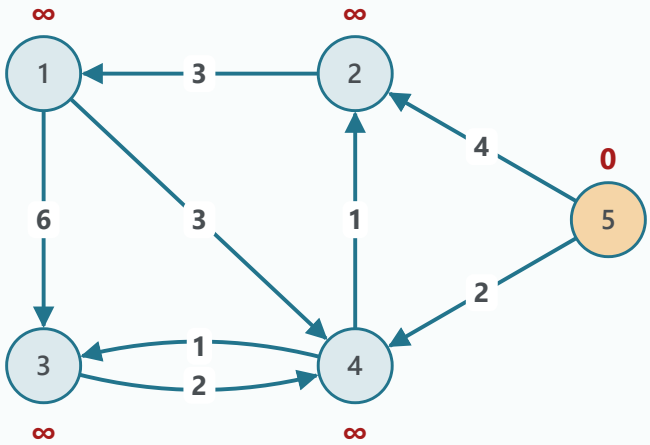
Dijkstra Example 1: Undirected Graph, Source 1



vertex	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	**20**	3
5	**20**	6
6	**11**	3

Done: distances from 1 are 2:7, 3:9, 6:11, 4:20, 5:20. The green edges form the **shortest-path tree** (e.g. 1 → 3 → 6 → 5).

Dijkstra Example 2: Directed Graph, Sets Q and S



Q

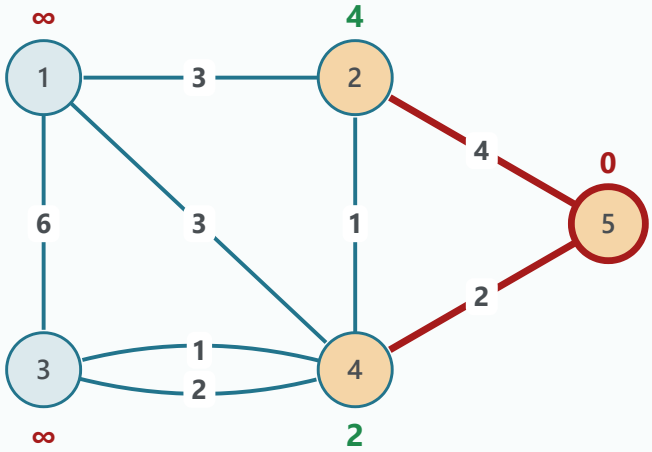
d	u	π
0	5	/
∞	1	/
∞	2	/
∞	3	/
∞	4	/

S

u	d	π

Initialization: source **5** has distance 0, all others ∞. Every vertex is in Q, S = ∅.

Dijkstra Example 2: Directed Graph, Sets Q and S



Q

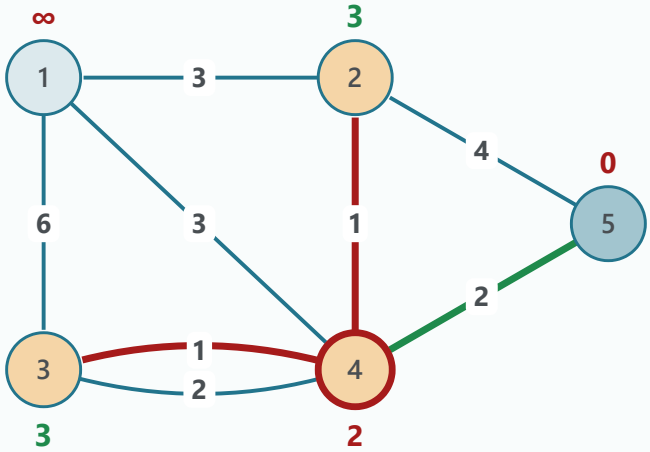
d	u	π
2	4	5
4	2	5
∞	1	/
∞	3	/

S

u	d	π
5	0	/

Step 1: take **5** from Q and put it in S with d = 0. Relax: (5,2): ∞ → **4**; (5,4): ∞ → **2**.

Dijkstra Example 2: Directed Graph, Sets Q and S



Q

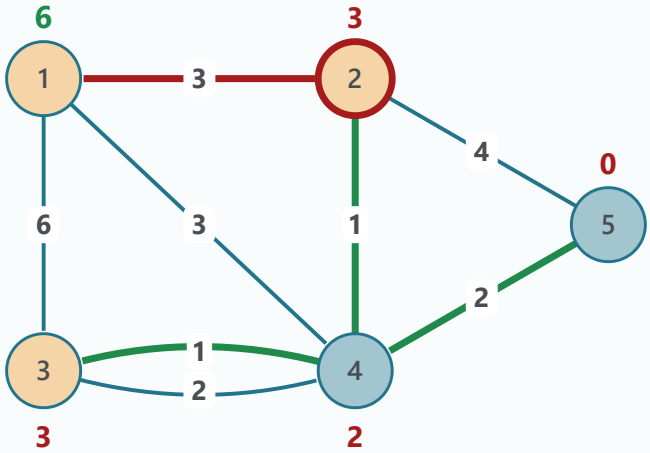
d	u	π
3	2	4
3	3	4
∞	1	/

S

u	d	π
5	0	/
4	2	5

Step 2: take **4** from Q and put it in S with d = 2. Relax: (4,2): 4 → **3**; (4,3): ∞ → **3**.

Dijkstra Example 2: Directed Graph, Sets Q and S



Q

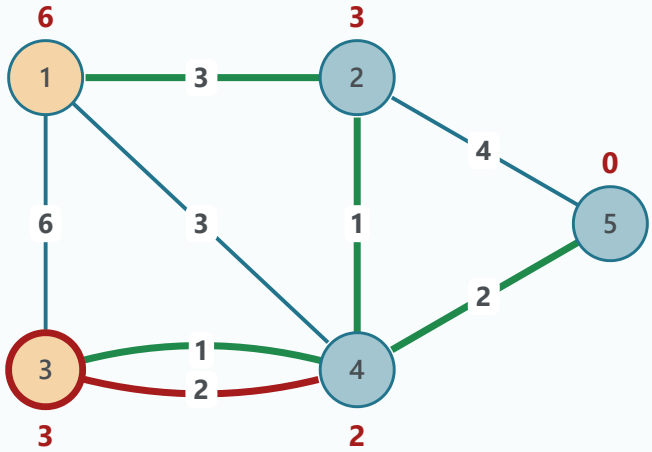
d	u	π
3	3	4
6	1	2

S

u	d	π
5	0	/
4	2	5
2	3	4

Step 3: take **2** from Q and put it in S with d = 3. Relax: (2,1): ∞ → **6**. (2 and 3 tie at 3; the lower-numbered 2 is taken.)

Dijkstra Example 2: Directed Graph, Sets Q and S



Q

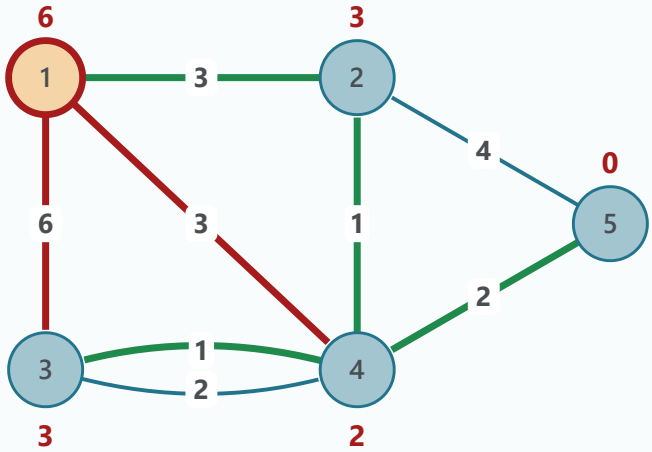
d	u	π
6	1	2

S

u	d	π
5	0	/
4	2	5
2	3	4
3	3	4

Step 4: take **3** from Q and put it in S with d = 3. Relax: (3,4): 4 is already in S. No shorter path (no relaxation).

Dijkstra Example 2: Directed Graph, Sets Q and S



Q

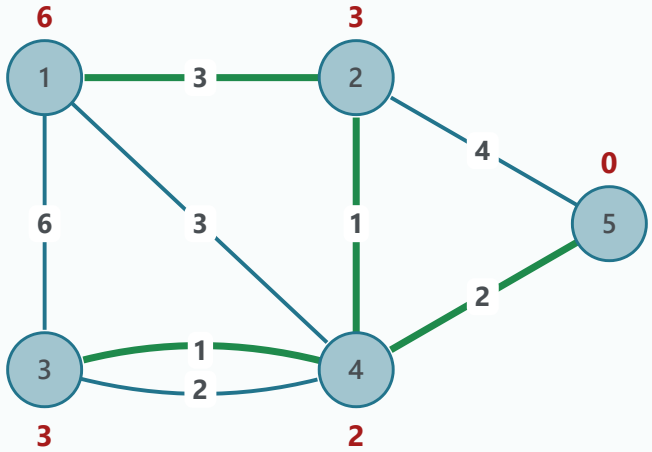
d	u	π

S

u	d	π
5	0	/
4	2	5
2	3	4
3	3	4
1	6	2

Step 5: take **1** from Q and put it in S with d = 6. Relax: (1,3): 3 is already in S; (1,4): 4 is already in S. No shorter path (no relaxation).

Dijkstra Example 2: Directed Graph, Sets Q and S



	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

Final state: shortest paths from 5. Only the π edges (green) remain: 5→4→2→1 (6), 5→4→3 (3).

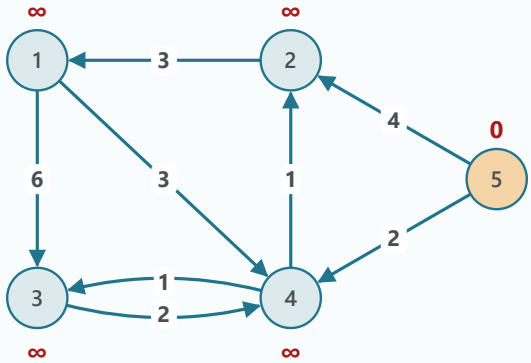
Dijkstra: Pseudocode

```
DIJKSTRA(G, source):
  dist = [ ]; visited = [ ]
  pq = { }
  for each v in V:
    dist[v] = infinity
    visited[v] = 0
  dist[source] = 0
  pq.add(source, 0)
  while pq is not empty do:
    u = pq.extractMin()   // smallest distance
    if visited[u] == 0:
      visited[u] = 1
      for each (u, v) in G[u]:
        if dist[u] + weight(u, v) < dist[v]:
          dist[v] = dist[u] + weight(u, v)
          pq.add(v, dist[v])
  return dist
```

Lazy deletion: a vertex may enter the queue several times; when a stale entry comes out, the `visited` check skips it. To recover paths, relaxation also stores `prev[v] = u`.

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



Set up the arrays (vertices 1..5, index 0 unused); dist[5] = 0 and (5, 0) is added to the queue.

Step 1 / 27

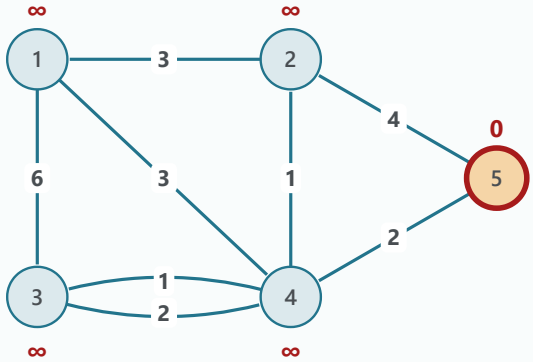
VARIABLES

`u = –` `v = –` `w = –`

`pq = [(5,0)]`

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



Poll the entry with the smallest distance, **(5, 0)**: u = 5.

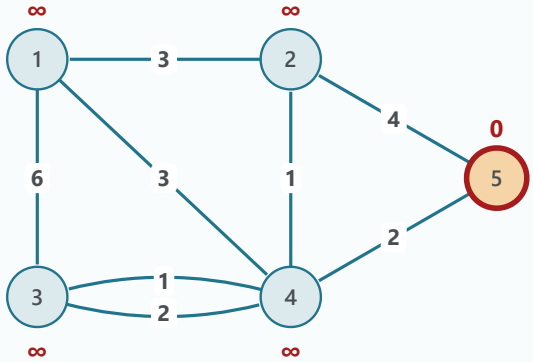
Step 2 / 27

VARIABLES

u = **5** | v = – | w = – | pq = **[]**

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
▶      visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



5 is settled: visited[5] = true.

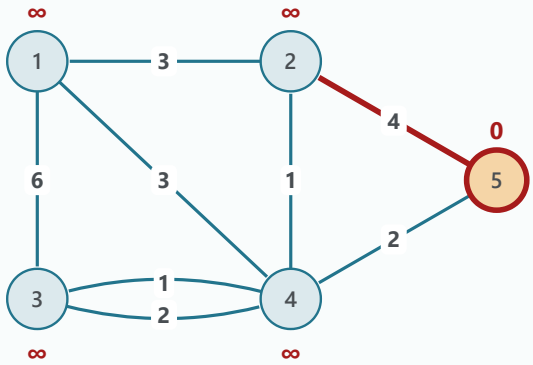
Step 3 / 27

VARIABLES

`u = 5` `v = –` `w = –` `pq = []`

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
▶        int v = e.to, w = e.weight;
▶        if (!visited[v]
▶            && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



Edge 5→2 (w = 4): 0 + 4 < ∞ → condition **true**.

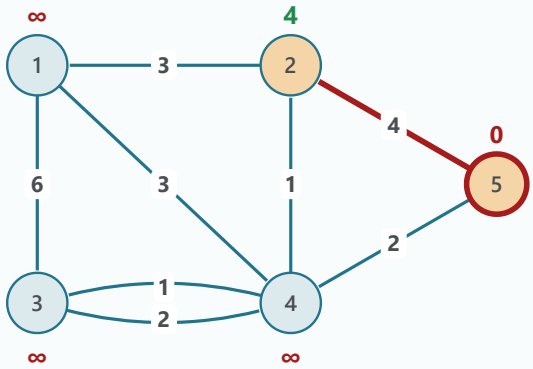
Step 4 / 27

VARIABLES

u = **5** v = **2** w = **4** pq = **[]**

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
▶          dist[v] = dist[u] + w;
▶          prev[v] = u;
▶          pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



dist[2] = 4, prev[2] = 5; add (2, 4) to the queue.

Step 5 / 27

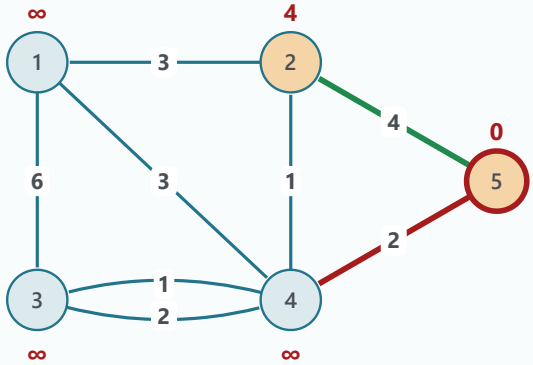
VARIABLES

`u = 5` `v = 2` `w = 4`

`pq = [(2,4)]`

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
▶        int v = e.to, w = e.weight;
▶        if (!visited[v]
▶            && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



Edge 5→4 (w = 2): 0 + 2 < ∞ → condition **true**.

Step 6 / 27

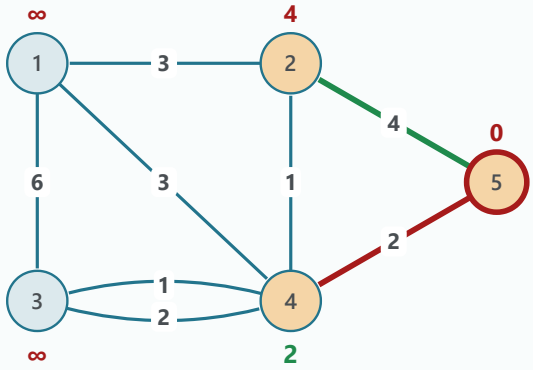
VARIABLES

u = 5 | v = 4 | w = 2

pq = [(2,4)]

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
▶          dist[v] = dist[u] + w;
▶          prev[v] = u;
▶          pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



dist[4] = 2, prev[4] = 5; add (4, 2) to the queue.

Step 7 / 27

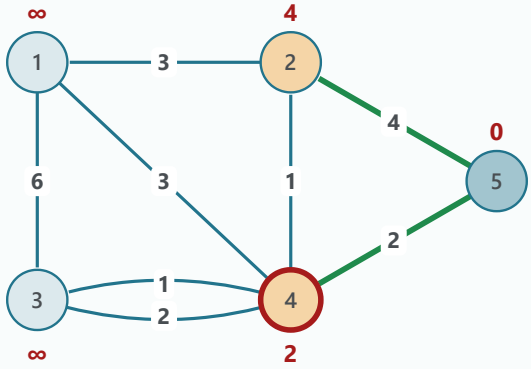
VARIABLES

`u = 5` `v = 4` `w = 2`

`pq = [(4,2), (2,4)]`

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



Poll the entry with the smallest distance, **(4, 2)**: u = 4.

Step 8 / 27

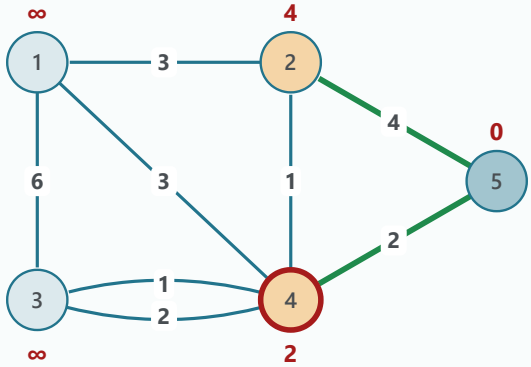
VARIABLES

`u = 4` `v = –` `w = –`

`pq = [(2,4)]`

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
▶      visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



4 is settled: visited[4] = true.

Step 9 / 27

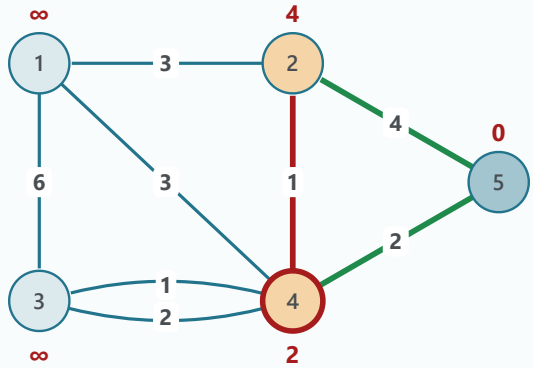
VARIABLES

`u = 4` `v = –` `w = –`

`pq = [(2,4)]`

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
▶        int v = e.to, w = e.weight;
▶        if (!visited[v]
▶            && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



Edge 4→2 (w = 1): 2 + 1 < 4 → condition **true**.

Step 10 / 27

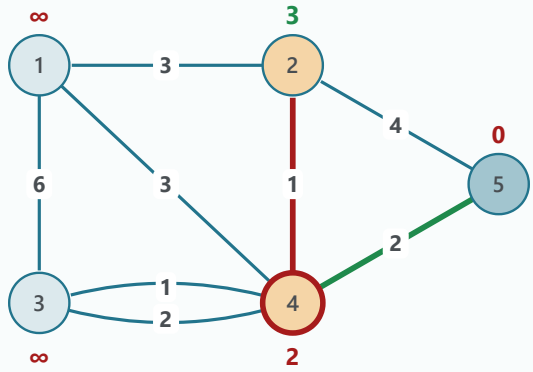
VARIABLES

u = 4 | v = 2 | w = 1

pq = [(2,4)]

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
▶          dist[v] = dist[u] + w;
▶          prev[v] = u;
▶          pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



dist[2] = 3, prev[2] = 4; add (2, 3) to the queue.

Step 11 / 27

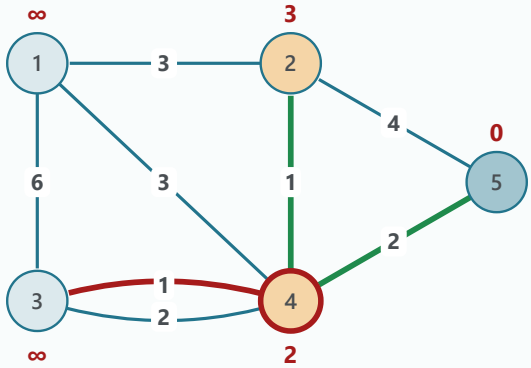
VARIABLES

`u = 4` `v = 2` `w = 1`

`pq = [(2,3), (2,4)]`

Dijkstra (Priority Queue): Step Through the Code

```java
void dijkstra(int src) {
  int[] dist = new int[V + 1];
  int[] prev = new int[V + 1];
  boolean[] visited = new boolean[V + 1];
  Arrays.fill(dist, INF);
  dist[src] = 0;
  PriorityQueue<int[]> pq =
    new PriorityQueue<>((a, b) -> a[1] - b[1]);
  pq.add(new int[] { src, 0 });
  while (!pq.isEmpty()) {
    int u = pq.poll()[0];
    if (visited[u]) continue; // stale entry
    visited[u] = true;
    for (Edge e : adj.get(u)) {
      int v = e.to, w = e.weight;
      if (!visited[v]
          && dist[u] + w < dist[v]) {
        dist[v] = dist[u] + w;
        prev[v] = u;
        pq.add(new int[] { v, dist[v] });
      }
    }
  }
}
```



Edge 4→3 (w = 1): 2 + 1 < ∞ → condition **true**.

Step 12 / 27

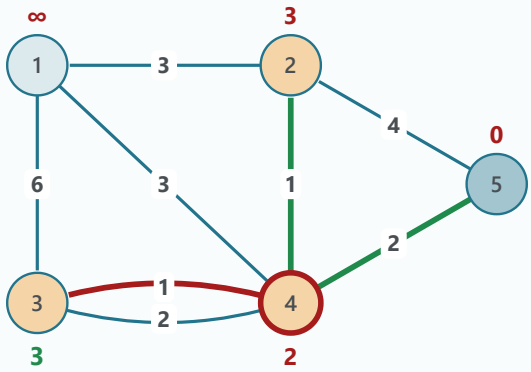
VARIABLES

u = 4 | v = 3 | w = 1

pq = [(2,3), (2,4)]

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
▶        dist[v] = dist[u] + w;
▶        prev[v] = u;
▶        pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



dist[3] = 3, prev[3] = 4; add (3, 3) to the queue.

Step 13 / 27

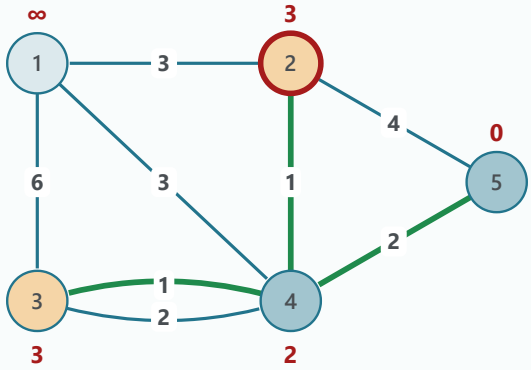
VARIABLES

`u = 4` `v = 3` `w = 1`

`pq = [(2,3), (3,3), (2,4)]`

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



Poll the entry with the smallest distance, **(2, 3)**: u = 2.

Step 14 / 27

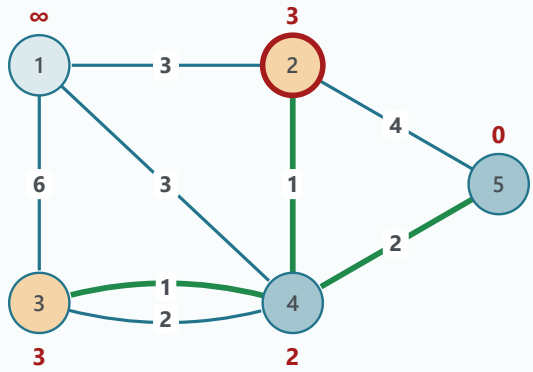
VARIABLES

u = **2** | v = – | w = –

pq = [(3,3), (2,4)]

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
▶      visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



2 is settled: visited[2] = true.

Step 15 / 27

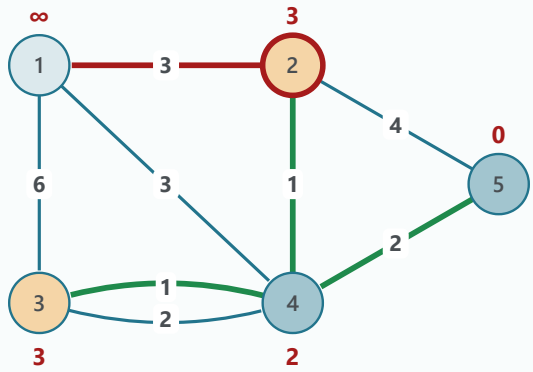
VARIABLES

`u = 2` `v = –` `w = –`

`pq = [(3,3), (2,4)]`

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
▶        int v = e.to, w = e.weight;
▶        if (!visited[v]
▶            && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



Edge 2→1 (w = 3): 3 + 3 < ∞ → condition **true**.

Step 16 / 27

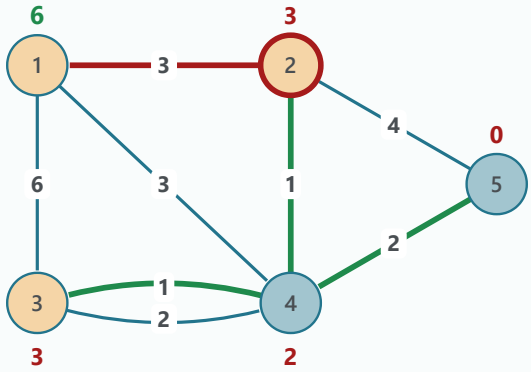
VARIABLES

u = 2 | v = 1 | w = 3

pq = [(3,3), (2,4)]

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
▶          dist[v] = dist[u] + w;
▶          prev[v] = u;
▶          pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



dist[1] = 6, prev[1] = 2; add (1, 6) to the queue.

Step 17 / 27

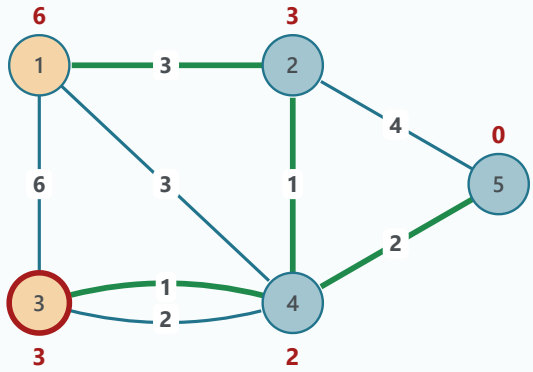
VARIABLES

`u = 2` `v = 1` `w = 3`

`pq = [(3,3), (2,4), (1,6)]`

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



Poll the entry with the smallest distance, **(3, 3)**: u = 3.

Step 18 / 27

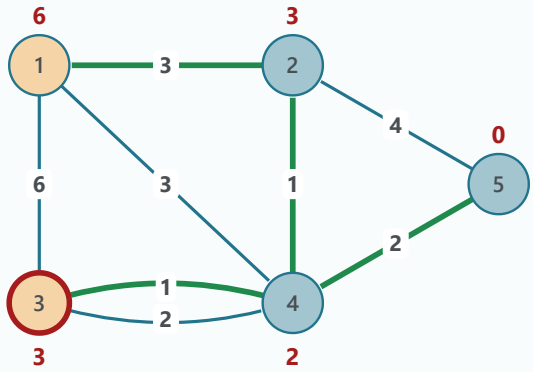
VARIABLES

`u = 3` `v = –` `w = –`

`pq = [(2,4), (1,6)]`

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
▶      visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



3 is settled: visited[3] = true.

Step 19 / 27

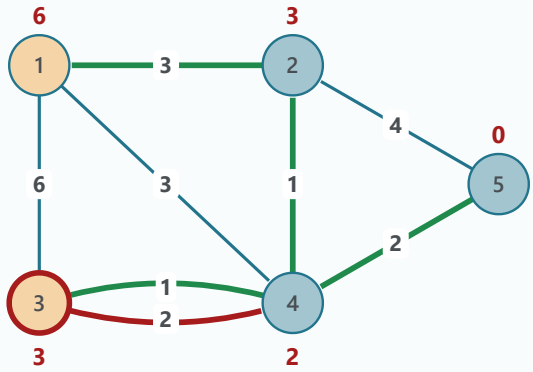
VARIABLES

`u = 3` `v = –` `w = –`

`pq = [(2,4), (1,6)]`

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
▶        int v = e.to, w = e.weight;
▶        if (!visited[v]
▶            && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



Edge 3→4 (w = 2): visited[4] = true → condition **false**.

Step 20 / 27

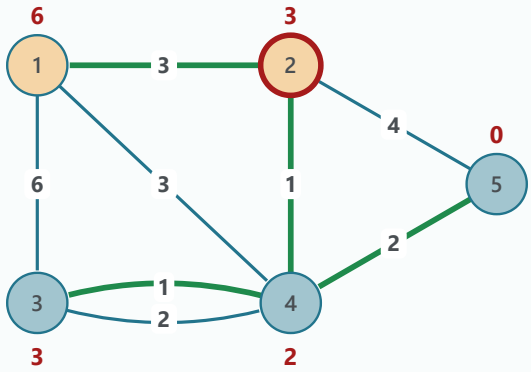
VARIABLES

`u = 3` `v = 4` `w = 2`

`pq = [(2,4), (1,6)]`

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



Poll the entry with the smallest distance, **(2, 4)**: u = 2.

Step 21 / 27

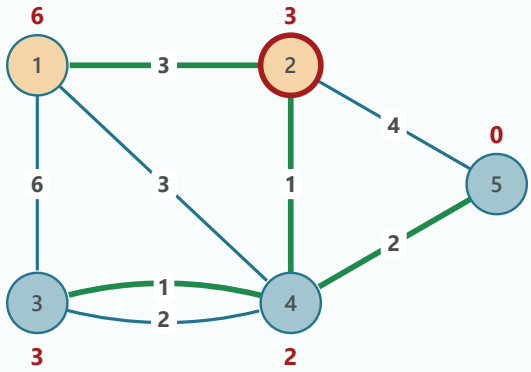
VARIABLES

u = 2 | v = – | w = –

pq = [(1,6)]

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
▶      if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



visited[2] is already true: this is 2's stale entry (2, 4) → skipped with `continue` (lazy deletion).

Step 22 / 27

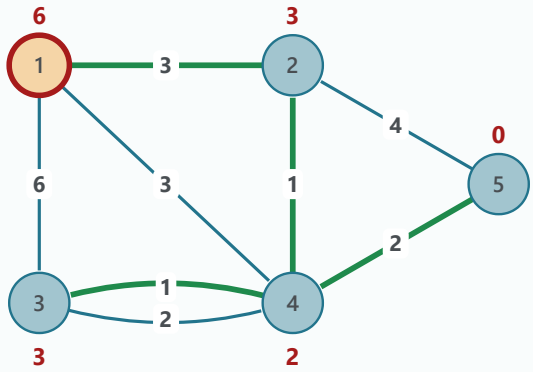
VARIABLES

`u = 2` `v = –` `w = –`

`pq = [(1,6)]`

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



Poll the entry with the smallest distance, **(1, 6)**: u = 1.

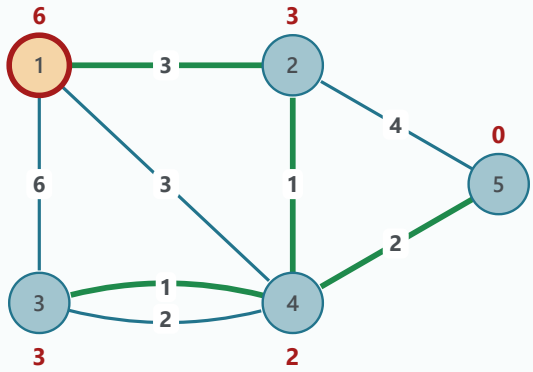
Step 23 / 27

VARIABLES

u = 1 | v = – | w = – | pq = []

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
▶      visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



1 is settled: visited[1] = true.

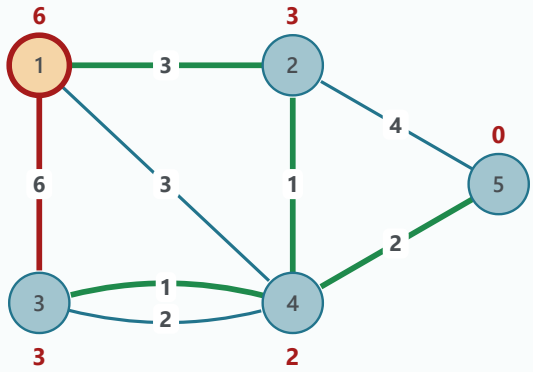
Step 24 / 27

VARIABLES

`u = 1` `v = –` `w = –` `pq = []`

Dijkstra (Priority Queue): Step Through the Code

```java
void dijkstra(int src) {
  int[] dist = new int[V + 1];
  int[] prev = new int[V + 1];
  boolean[] visited = new boolean[V + 1];
  Arrays.fill(dist, INF);
  dist[src] = 0;
  PriorityQueue<int[]> pq =
    new PriorityQueue<>((a, b) -> a[1] - b[1]);
  pq.add(new int[] { src, 0 });
  while (!pq.isEmpty()) {
    int u = pq.poll()[0];
    if (visited[u]) continue; // stale entry
    visited[u] = true;
    for (Edge e : adj.get(u)) {
      int v = e.to, w = e.weight;
      if (!visited[v]
          && dist[u] + w < dist[v]) {
        dist[v] = dist[u] + w;
        prev[v] = u;
        pq.add(new int[] { v, dist[v] });
      }
    }
  }
}
```



Edge 1→3 (w = 6): visited[3] = true → condition **false**.

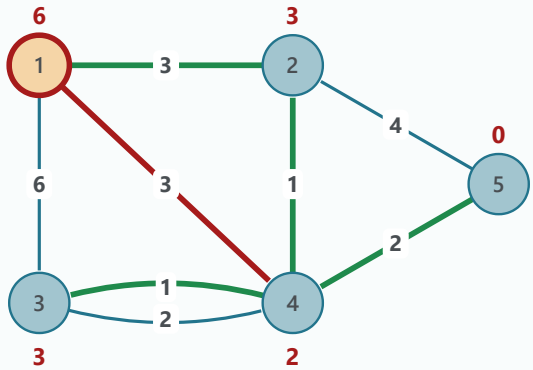
Step 25 / 27

VARIABLES

u = **1** v = **3** w = **6** pq = **[]**

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
10   while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
▶        int v = e.to, w = e.weight;
▶        if (!visited[v]
▶            && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



Edge 1→4 (w = 3): visited[4] = true → condition **false**.

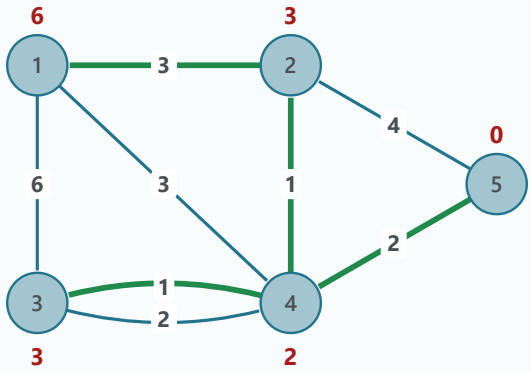
Step 26 / 27

VARIABLES

`u = 1` `v = 4` `w = 3` `pq = []`

Dijkstra (Priority Queue): Step Through the Code

```
1  void dijkstra(int src) {
2    int[] dist = new int[V + 1];
3    int[] prev = new int[V + 1];
4    boolean[] visited = new boolean[V + 1];
5    Arrays.fill(dist, INF);
6    dist[src] = 0;
7    PriorityQueue<int[]> pq =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    pq.add(new int[] { src, 0 });
▶    while (!pq.isEmpty()) {
11     int u = pq.poll()[0];
12     if (visited[u]) continue; // stale entry
13     visited[u] = true;
14     for (Edge e : adj.get(u)) {
15       int v = e.to, w = e.weight;
16       if (!visited[v]
17           && dist[u] + w < dist[v]) {
18         dist[v] = dist[u] + w;
19         prev[v] = u;
20         pq.add(new int[] { v, dist[v] });
21       }
22     }
23   }
24 }
```



The queue is empty and the loop ends. dist = [6, 3, 3, 2, 0], prev = [2, 4, 4, 5, –].

Step 27 / 27

VARIABLES

`u = –` `v = –` `w = –` `pq = []`

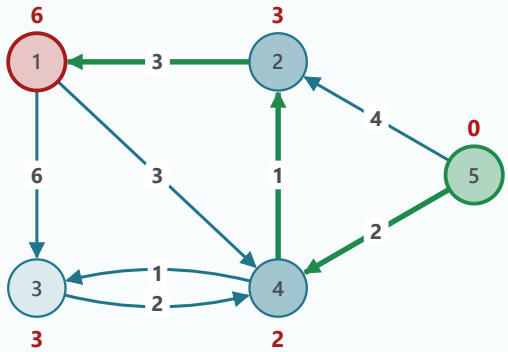
Path Reconstruction: The π (Predecessor) Array

Result of Example 2: π = [2, 4, 4, 5, /]. Walking back from target 1:

1 ← π[1] = **2** ← π[2] = **4** ← π[4] = **5** (source)

Reversed, the path is **5** → **4** → **2** → **1**, of length 2 + 1 + 3 = **6** = d[1].

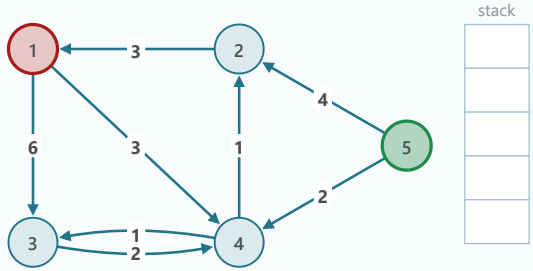
```
void printPath(int[] prev, int[] dist,
               int src, int target) {
  if (dist[target] == INF) {   // unreachable
    System.out.print("no path"); return;
  }
  Deque<Integer> stack = new ArrayDeque<>();
  for (int v = target; v != src; v = prev[v])
    stack.push(v);          // walk back
  stack.push(src);
  while (!stack.isEmpty())
    System.out.print(stack.pop() + " ");
}
```



A stack turns the vertices collected backward into the right order.

Path Reconstruction: Tracing `printPath(prev, dist, 5, 1)`

```
1  void printPath(int[] prev, int[] dist,
2                 int src, int target) {
3    if (dist[target] == INF) {    // unreachable
4      System.out.print("no path"); return;
5    }
6    Deque<Integer> stack = new ArrayDeque<>();
7    for (int v = target; v != src; v = prev[v])
8      stack.push(v);          // walk back
9    stack.push(src);
10   while (!stack.isEmpty())
11     System.out.print(stack.pop() + " ");
12 }
```



From Dijkstra Example 2 (source 5, target 1). d[1] = 6 ≠ INF: reachable. Start with an empty stack.

Step 1 / 10

VARIABLES

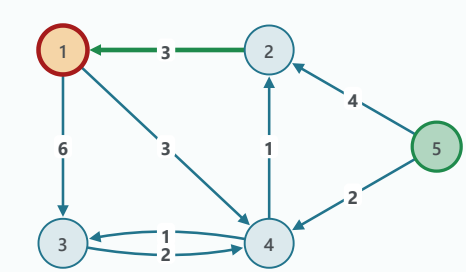
`v = –` `stack = []`

`prev = [2, 4, 4, 5, /]`

OUTPUT

Path Reconstruction: Tracing `printPath(prev, dist, 5, 1)`

```
1  void printPath(int[] prev, int[] dist,
2                 int src, int target) {
3    if (dist[target] == INF) {    // unreachable
4      System.out.print("no path"); return;
5    }
6    Deque<Integer> stack = new ArrayDeque<>();
▶    for (int v = target; v != src; v = prev[v])
▶      stack.push(v);          // walk back
9    stack.push(src);
10   while (!stack.isEmpty())
11     System.out.print(stack.pop() + " ");
12 }
```



1 ≠ source → push it. Next, v = prev[1] = 2.

Step 2 / 10

VARIABLES

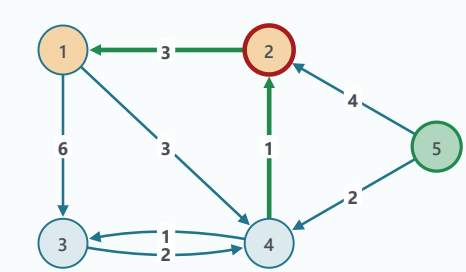
`v = 1` `stack = [1]`

`prev = [2, 4, 4, 5, /]`

OUTPUT

Path Reconstruction: Tracing `printPath(prev, dist, 5, 1)`

```
1  void printPath(int[] prev, int[] dist,
2                 int src, int target) {
3    if (dist[target] == INF) {    // unreachable
4      System.out.print("no path"); return;
5    }
6    Deque<Integer> stack = new ArrayDeque<>();
▶    for (int v = target; v != src; v = prev[v])
▶      stack.push(v);          // walk back
9    stack.push(src);
10   while (!stack.isEmpty())
11     System.out.print(stack.pop() + " ");
12 }
```



2 ≠ source → push it. Next, v = prev[2] = 4.

Step 3 / 10

VARIABLES

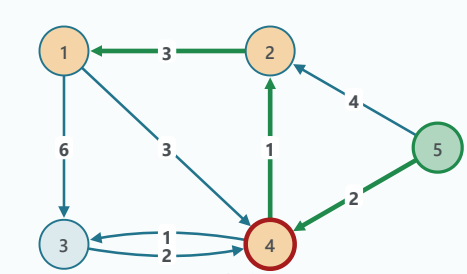
`v = 2` `stack = [2, 1]`

`prev = [2, 4, 4, 5, /]`

OUTPUT

Path Reconstruction: Tracing `printPath(prev, dist, 5, 1)`

```
1  void printPath(int[] prev, int[] dist,
2                 int src, int target) {
3    if (dist[target] == INF) {    // unreachable
4      System.out.print("no path"); return;
5    }
6    Deque<Integer> stack = new ArrayDeque<>();
▶    for (int v = target; v != src; v = prev[v])
▶      stack.push(v);          // walk back
9    stack.push(src);
10   while (!stack.isEmpty())
11     System.out.print(stack.pop() + " ");
12 }
```



4 ≠ source → push it. Next, v = prev[4] = 5.

Step 4 / 10

VARIABLES

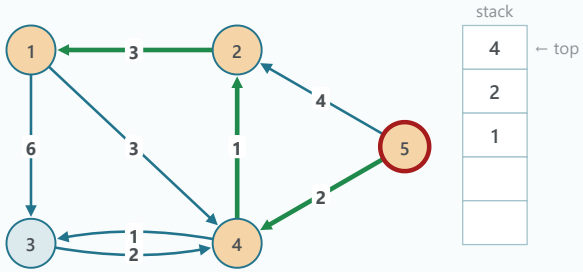
`v = 4` `stack = [4, 2, 1]`

`prev = [2, 4, 4, 5, /]`

OUTPUT

Path Reconstruction: Tracing `printPath(prev, dist, 5, 1)`

```
1  void printPath(int[] prev, int[] dist,
2                 int src, int target) {
3    if (dist[target] == INF) {    // unreachable
4      System.out.print("no path"); return;
5    }
6    Deque<Integer> stack = new ArrayDeque<>();
▶    for (int v = target; v != src; v = prev[v])
8      stack.push(v);          // walk back
9    stack.push(src);
10   while (!stack.isEmpty())
11     System.out.print(stack.pop() + " ");
12 }
```



v = 5 = source: the loop ends. 5 is not on the stack yet.

Step 5 / 10

VARIABLES

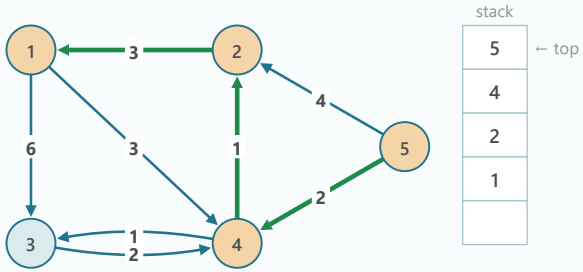
`v = 5` `stack = [4, 2, 1]`

`prev = [2, 4, 4, 5, /]`

OUTPUT

Path Reconstruction: Tracing `printPath(prev, dist, 5, 1)`

```
1  void printPath(int[] prev, int[] dist,
2                 int src, int target) {
3    if (dist[target] == INF) {   // unreachable
4      System.out.print("no path"); return;
5    }
6    Deque<Integer> stack = new ArrayDeque<>();
7    for (int v = target; v != src; v = prev[v])
8      stack.push(v);          // walk back
▶    stack.push(src);
10   while (!stack.isEmpty())
11     System.out.print(stack.pop() + " ");
12 }
```



Push the source 5: on top, with the target at the bottom.

Step 6 / 10

VARIABLES

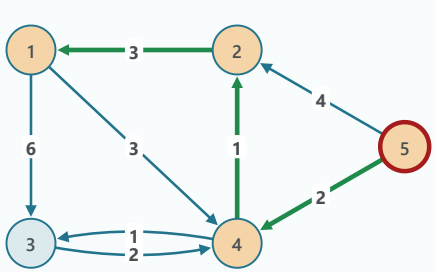
`v = –` `stack = [5, 4, 2, 1]`

`prev = [2, 4, 4, 5, /]`

OUTPUT

Path Reconstruction: Tracing `printPath(prev, dist, 5, 1)`

```
1  void printPath(int[] prev, int[] dist,
2                 int src, int target) {
3    if (dist[target] == INF) {    // unreachable
4      System.out.print("no path"); return;
5    }
6    Deque<Integer> stack = new ArrayDeque<>();
7    for (int v = target; v != src; v = prev[v])
8      stack.push(v);          // walk back
9    stack.push(src);
▶    while (!stack.isEmpty())
▶      System.out.print(stack.pop() + " ");
12 }
```



Pop and print 5 (LIFO: last in, first out).

Step 7 / 10

VARIABLES

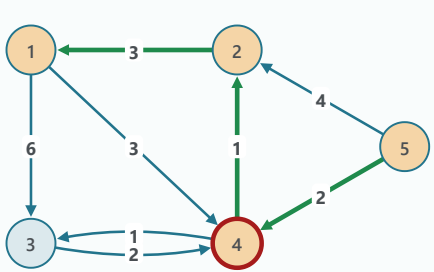
`v = –` `stack = [4, 2, 1]` `prev = [2, 4, 4, 5, /]`

OUTPUT

```
5
```

Path Reconstruction: Tracing `printPath(prev, dist, 5, 1)`

```
1  void printPath(int[] prev, int[] dist,
2                 int src, int target) {
3    if (dist[target] == INF) {    // unreachable
4      System.out.print("no path"); return;
5    }
6    Deque<Integer> stack = new ArrayDeque<>();
7    for (int v = target; v != src; v = prev[v])
8      stack.push(v);          // walk back
9    stack.push(src);
▶    while (!stack.isEmpty())
▶      System.out.print(stack.pop() + " ");
12 }
```



Pop and print 4 (LIFO: last in, first out).

Step 8 / 10

VARIABLES

`v = –` `stack = [2, 1]`

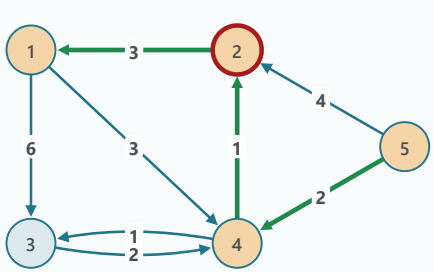
`prev = [2, 4, 4, 5, /]`

OUTPUT

```
5 4
```

Path Reconstruction: Tracing `printPath(prev, dist, 5, 1)`

```
1  void printPath(int[] prev, int[] dist,
2                 int src, int target) {
3    if (dist[target] == INF) {    // unreachable
4      System.out.print("no path"); return;
5    }
6    Deque<Integer> stack = new ArrayDeque<>();
7    for (int v = target; v != src; v = prev[v])
8      stack.push(v);          // walk back
9    stack.push(src);
▶    while (!stack.isEmpty())
▶      System.out.print(stack.pop() + " ");
12 }
```



Pop and print 2 (LIFO: last in, first out).

Step 9 / 10

VARIABLES

`v = –` `stack = [1]`

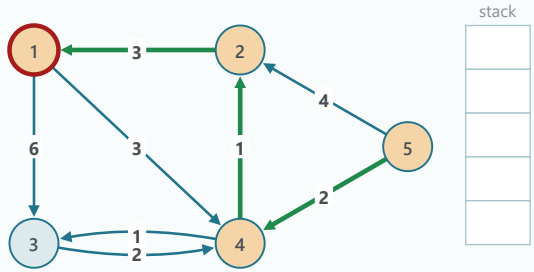
`prev = [2, 4, 4, 5, /]`

OUTPUT

```
5 4 2
```

Path Reconstruction: Tracing `printPath(prev, dist, 5, 1)`

```
1  void printPath(int[] prev, int[] dist,
2                 int src, int target) {
3    if (dist[target] == INF) {    // unreachable
4      System.out.print("no path"); return;
5    }
6    Deque<Integer> stack = new ArrayDeque<>();
7    for (int v = target; v != src; v = prev[v])
8      stack.push(v);          // walk back
9    stack.push(src);
▶    while (!stack.isEmpty())
▶      System.out.print(stack.pop() + " ");
12 }
```



Pop and print 1. Path **5** → **4** → **2** → **1**, length 2 + 1 + 3 = 6.

Step 10 / 10

VARIABLES

`v = –` `stack = []`

`prev = [2, 4, 4, 5, /]`

OUTPUT

```
5 4 2 1
```

Dijkstra: Running Time

The cost depends on how we **select the vertex with the smallest distance**:

Selection structure	Select	Relax	Total	Best for
Array (linear scan)	$V \times O(V)$	$E \times O(1)$	**$O(V^2)$**	dense ($E \approx V^2$)
Binary heap (priority queue)	$O(\log V)$	$O(\log V)$	**$O((V + E) \log V)$**	sparse ($E \ll V^2$)
Fibonacci heap	$O(\log V)$ amortized	$O(1)$ amortized	**$O(E + V \log V)$**	theoretical best

A numeric example

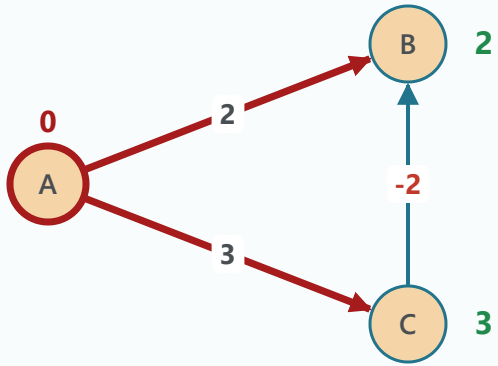
$V = 10^4$, $E = 5 \cdot 10^4$ (a road network): array $\approx 10^8$ operations, heap $\approx 6 \cdot 10^4 \cdot 14 \approx 8.4 \cdot 10^5$ operations.

Memory

The d, π and visited arrays take $O(V)$; with lazy deletion the queue holds at most E entries → $O(V + E)$.

Why Does Dijkstra Fail on Negative Edges?

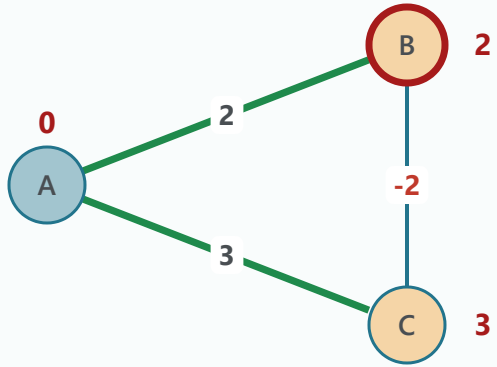
- The correctness proof relies on this: once u is settled, any path leaving S has length at least d[u], because edges are **nonnegative** and a path can only grow.
- A negative edge breaks that assumption: a vertex processed later can open a shorter path to an already settled vertex.
- Adding a constant to every weight is **no fix**: paths with more edges are penalized more, so the shortest path changes.
- With negative edges, use **Bellman-Ford**.



A is settled; relaxation gives d[B] = 2, d[C] = 3.

Why Does Dijkstra Fail on Negative Edges?

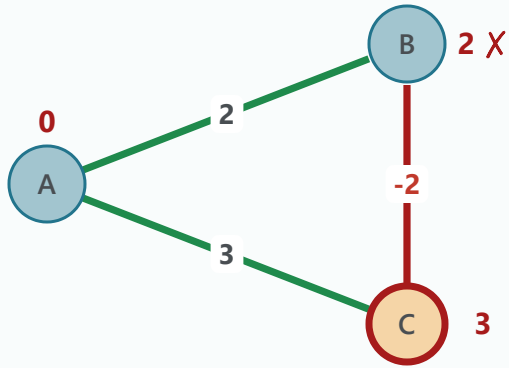
- The correctness proof relies on this: once u is settled, any path leaving S has length at least d[u], because edges are **nonnegative** and a path can only grow.
- A negative edge breaks that assumption: a vertex processed later can open a shorter path to an already settled vertex.
- Adding a constant to every weight is **no fix**: paths with more edges are penalized more, so the shortest path changes.
- With negative edges, use **Bellman-Ford**.



Smallest d: B (2). Dijkstra **settles** B, assuming "no shorter path to B can exist".

Why Does Dijkstra Fail on Negative Edges?

- The correctness proof relies on this: once u is settled, any path leaving S has length at least d[u], because edges are **nonnegative** and a path can only grow.
- A negative edge breaks that assumption: a vertex processed later can open a shorter path to an already settled vertex.
- Adding a constant to every weight is **no fix**: paths with more edges are penalized more, so the shortest path changes.
- With negative edges, use **Bellman-Ford**.



C is processed: d[C] + (−2) = 1 < 2, but B is already in S. The result d[B] = 2 is **wrong**; the true δ(A, B) = 1 (A → C → B).

The Bellman-Ford Algorithm

- Finds shortest paths from a **source** vertex to every other vertex (single-source).
- Named after Richard **Bellman** (1958) and Lester **Ford** Jr. (1956); E. F. Moore gave the same method in 1959.
- Handles **negative edge weights**.
- Detects **negative cycles** reachable from the source.
- Slower than Dijkstra: $O(V \cdot E)$.

How it works

It keeps an array of shortest-path **estimates**, all ∞ except 0 at the source. It examines **every edge** in turn and updates the estimates, repeating this sweep $V - 1$ times. There is no vertex selection and nothing greedy: it is **dynamic programming**, where pass i builds shortest paths of at most i edges from the previous pass.

Bellman-Ford: The Steps

Step 1: Set the start vertex's distance to **0** and every other vertex's to **∞**.

π[v] = undefined.

Bellman-Ford: The Steps

Step 1: Set the start vertex's distance to **0** and every other vertex's to **∞**.

π[v] = undefined.

Step 2: Examine the edges **one by one** and update distances by relaxation; repeat this pass **V − 1** times.

The edge order is fixed within a pass; it affects how many passes are needed, not the result.

Bellman-Ford: The Steps

Step 1: Set the start vertex's distance to **0** and every other vertex's to **∞**.

π[v] = undefined.

Step 2: Examine the edges **one by one** and update distances by relaxation; repeat this pass **V − 1** times.

The edge order is fixed within a pass; it affects how many passes are needed, not the result.

Step 3: Once a vertex's distance improves, its **neighbors'** distances may improve in later checks.

Each pass pushes the improvement at least one edge further along the path.

Bellman-Ford: The Steps

Step 1: Set the start vertex's distance to **0** and every other vertex's to **∞**.

π[v] = undefined.

Step 2: Examine the edges **one by one** and update distances by relaxation; repeat this pass **V − 1** times.

The edge order is fixed within a pass; it affects how many passes are needed, not the result.

Step 3: Once a vertex's distance improves, its **neighbors'** distances may improve in later checks.

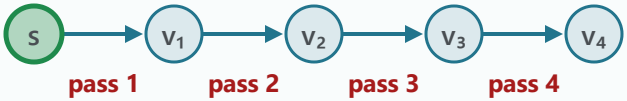
Each pass pushes the improvement at least one edge further along the path.

Step 4: To check for negative cycles, examine every edge **once more**; if some edge still relaxes, a negative cycle is reachable from the source.

Without such a cycle no edge relaxes after V − 1 passes.

Why V − 1 Passes? Complexity

- Without negative cycles every shortest path has **at most V − 1 edges**.
- **Invariant:** after pass i, d[v] = δ(s, v) for every v that has a shortest path with at most i edges.
- Pass i is sure to relax the path's i-th edge; V − 1 passes complete every path.
- Each pass examines E edges: **O(V · E)** time in total, O(V) extra memory.

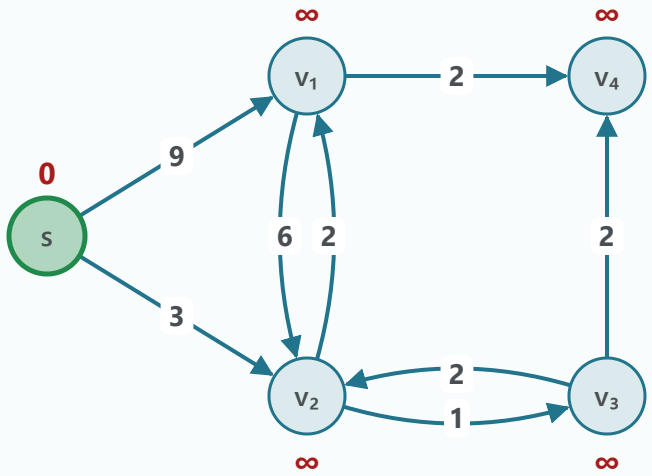


after pass i, the path of i edges is final

Graph	Time
Sparse, E ≈ V	$O(V^2)$
Dense, E ≈ V^2	$O(V^3)$
Best case (early exit)	O(E)

Early exit: if no d changes during a pass, later passes cannot change anything either; leave the loop.

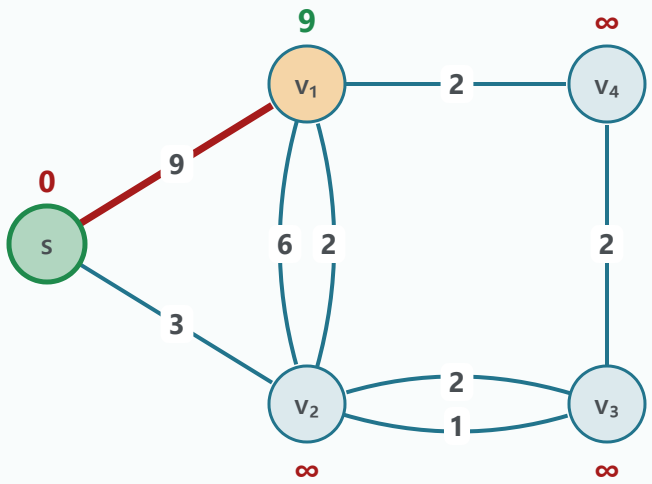
Bellman-Ford Example 1: Relaxing Edge by Edge



(s,v₁) 9 (s,v₂) 3 (v₁,v₂) 6 (v₁,v₄) 2 (v₂,v₁) 2 (v₂,v₃) 1 (v₃,v₂) 2 (v₃,v₄) 2

Initialization: d[s] = 0, all others ∞. Each pass examines the edges in the order shown above.

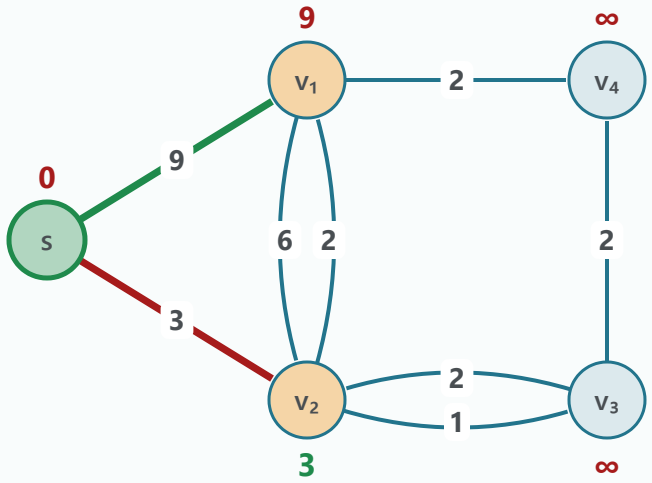
Bellman-Ford Example 1: Relaxing Edge by Edge



(s,v₁) 9 | (s,v₂) 3 | (v₁,v₂) 6 | (v₁,v₄) 2 | (v₂,v₁) 2 | (v₂,v₃) 1 | (v₃,v₂) 2 | (v₃,v₄) 2

Pass 1, edge (s, v_1): 0 + 9 = 9 < ∞ → **d[v_1] = 9**.

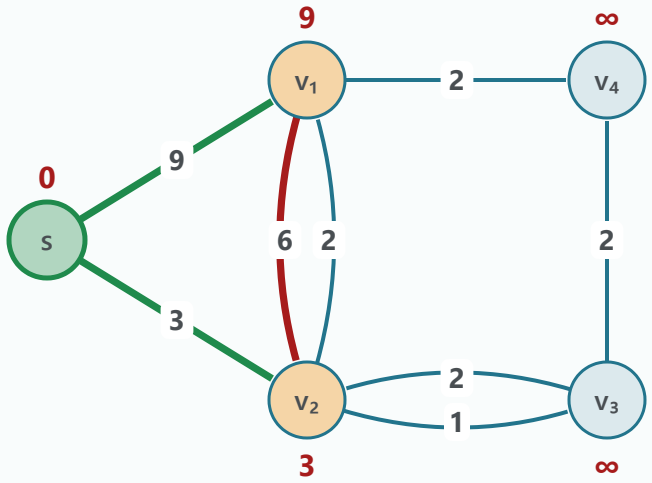
Bellman-Ford Example 1: Relaxing Edge by Edge



(s,v₁) 9 | (s,v₂) 3 | (v₁,v₂) 6 | (v₁,v₄) 2 | (v₂,v₁) 2 | (v₂,v₃) 1 | (v₃,v₂) 2 | (v₃,v₄) 2

Pass 1, edge (s, v_2): 0 + 3 = 3 < ∞ → **$d[v_2] = 3$**.

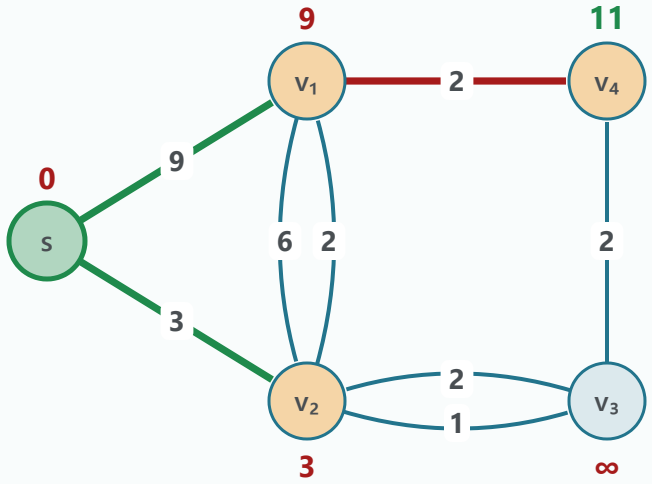
Bellman-Ford Example 1: Relaxing Edge by Edge



(s,v₁) 9 (s,v₂) 3 (v₁,v₂) 6 (v₁,v₄) 2 (v₂,v₁) 2 (v₂,v₃) 1 (v₃,v₂) 2 (v₃,v₄) 2

Pass 1, edge (v_1, v_2): 9 + 6 = 15 ≥ 3 → no change.

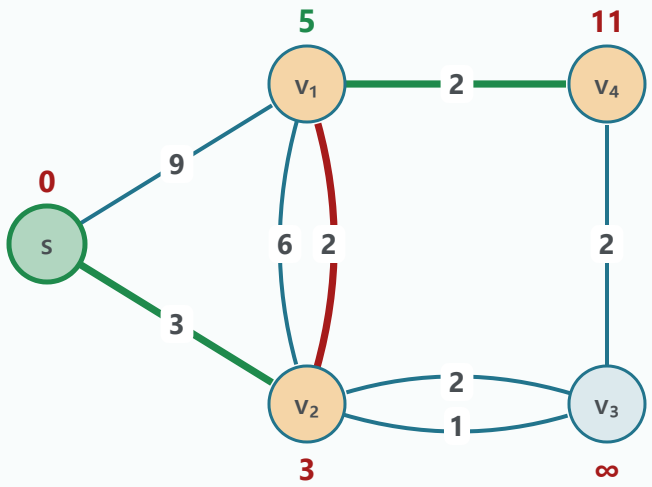
Bellman-Ford Example 1: Relaxing Edge by Edge



(s,v₁) 9 (s,v₂) 3 (v₁,v₂) 6 (v₁,v₄) 2 (v₂,v₁) 2 (v₂,v₃) 1 (v₃,v₂) 2 (v₃,v₄) 2

Pass 1, edge (v_1, v_4): 9 + 2 = 11 < ∞ → **d[v_4] = 11**.

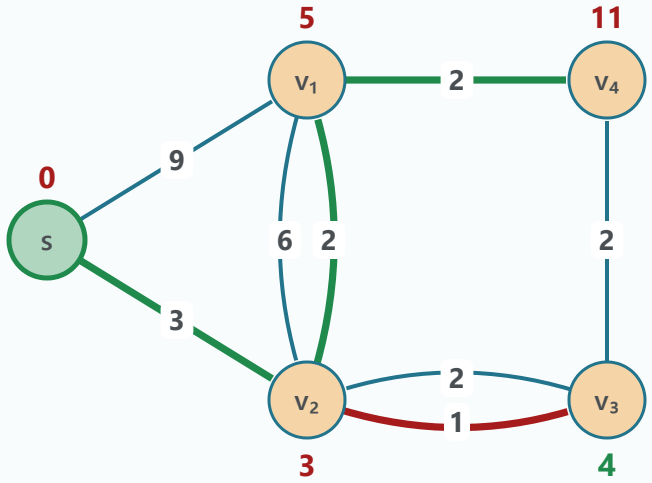
Bellman-Ford Example 1: Relaxing Edge by Edge



(s,v₁) 9 (s,v₂) 3 (v₁,v₂) 6 (v₁,v₄) 2 (v₂,v₁) 2 (v₂,v₃) 1 (v₃,v₂) 2 (v₃,v₄) 2

Pass 1, edge (v_2, v_1): 3 + 2 = 5 < 9 → **d[v_1] = 5**.

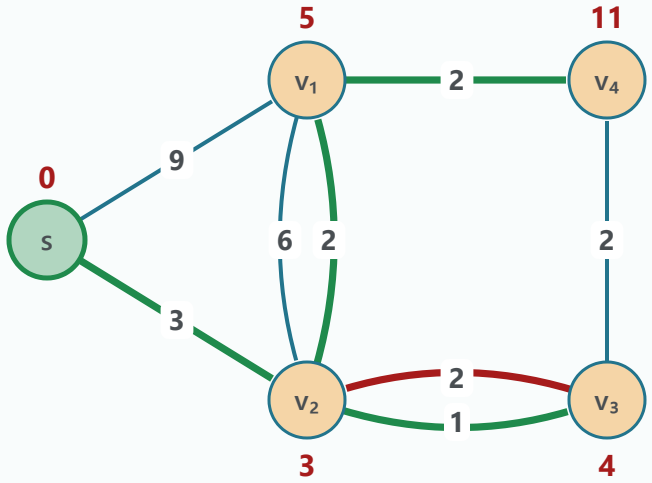
Bellman-Ford Example 1: Relaxing Edge by Edge



(s,v₁) 9 (s,v₂) 3 (v₁,v₂) 6 (v₁,v₄) 2 (v₂,v₁) 2 (v₂,v₃) 1 (v₃,v₂) 2 (v₃,v₄) 2

Pass 1, edge (v_2, v_3): 3 + 1 = 4 < ∞ → **d[v_3] = 4**.

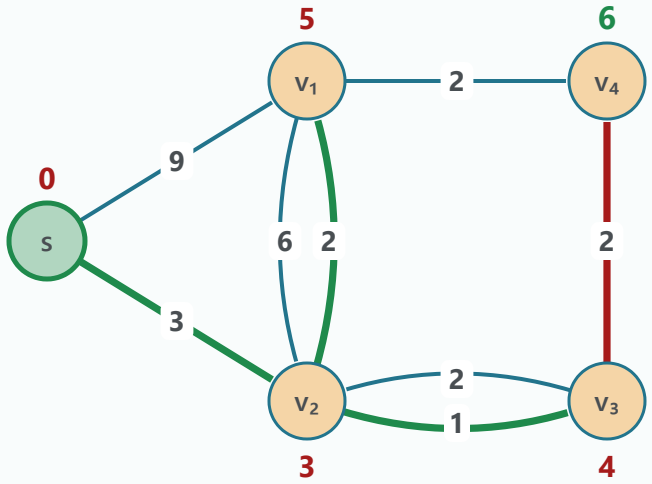
Bellman-Ford Example 1: Relaxing Edge by Edge



(s,v1) 9 (s,v2) 3 (v1,v2) 6 (v1,v4) 2 (v2,v1) 2 (v2,v3) 1 (v3,v2) 2 (v3,v4) 2

Pass 1, edge (v_3, v_2): 4 + 2 = 6 ≥ 3 → no change.

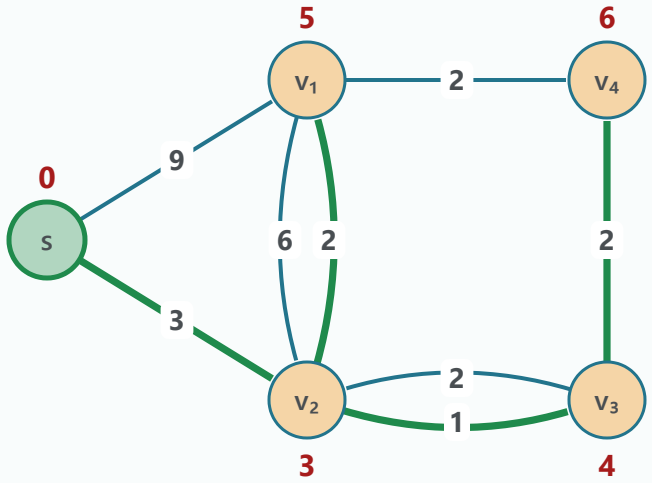
Bellman-Ford Example 1: Relaxing Edge by Edge



(s,v1) 9 (s,v2) 3 (v1,v2) 6 (v1,v4) 2 (v2,v1) 2 (v2,v3) 1 (v3,v2) 2 (v3,v4) 2

Pass 1, edge (v_3, v_4): 4 + 2 = 6 < 11 → **d[v_4] = 6**.

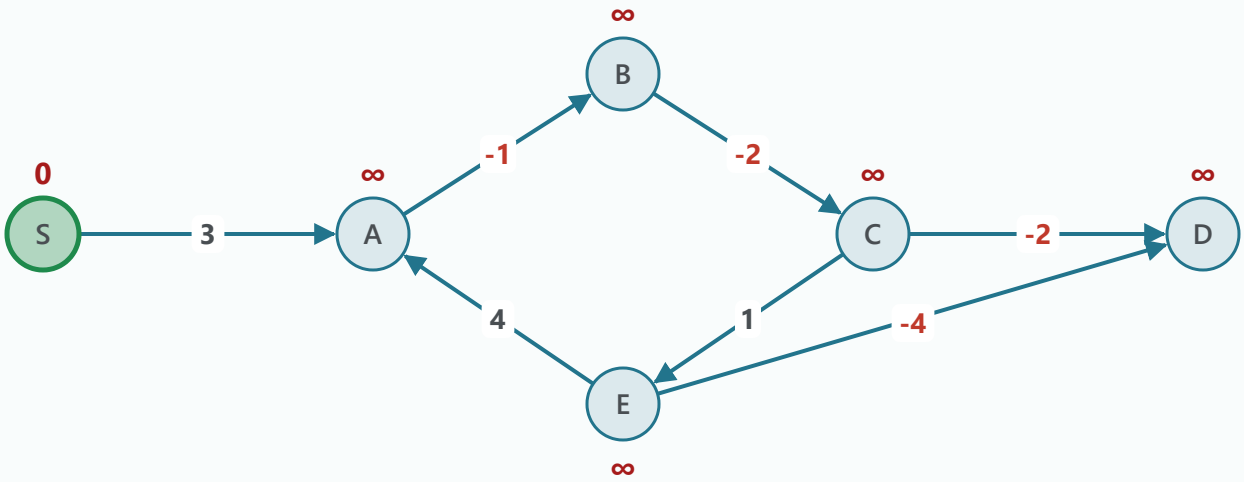
Bellman-Ford Example 1: Relaxing Edge by Edge



(s,v₁) 9 (s,v₂) 3 (v₁,v₂) 6 (v₁,v₄) 2 (v₂,v₁) 2 (v₂,v₃) 1 (v₃,v₂) 2 (v₃,v₄) 2

Pass 2: none of the eight edges relaxes → **early exit**. Result: $v_1 = 5$, $v_2 = 3$, $v_3 = 4$, $v_4 = 6$ (with this order one pass was enough).

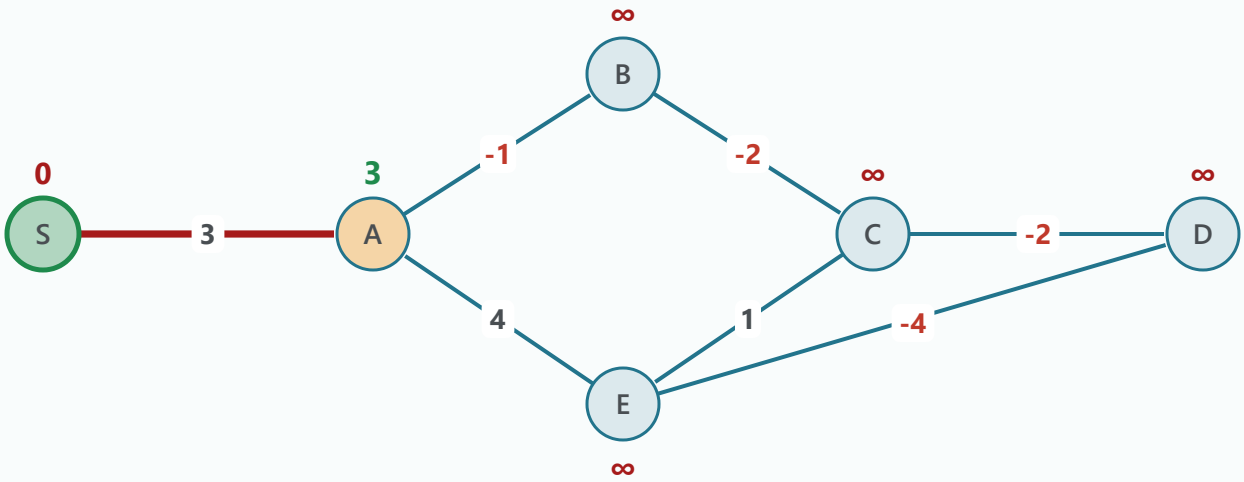
Bellman-Ford Example 2: Negative Edges



`(E,D) -4` `(E,A) 4` `(C,D) -2` `(C,E) 1` `(B,C) -2` `(A,B) -1` `(S,A) 3`

A graph with negative edges, source S. Each pass examines the edges **in the order shown above** (back to front: the worst order).

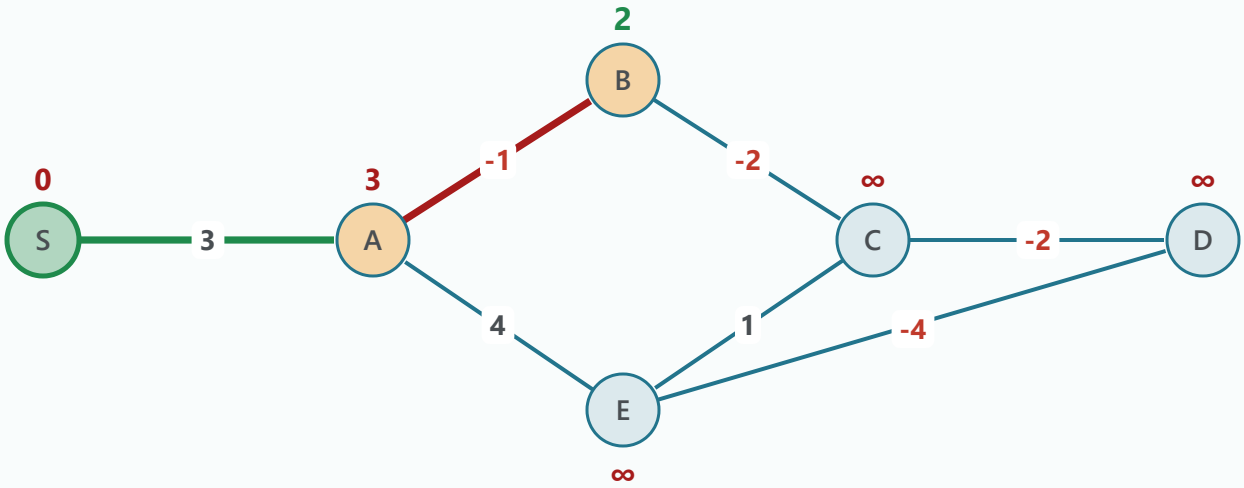
Bellman-Ford Example 2: Negative Edges



`(E,D) -4` `(E,A) 4` `(C,D) -2` `(C,E) 1` `(B,C) -2` `(A,B) -1` `(S,A) 3`

Pass 1: only (S,A): ∞ → **3**. The other edges still start at ∞.

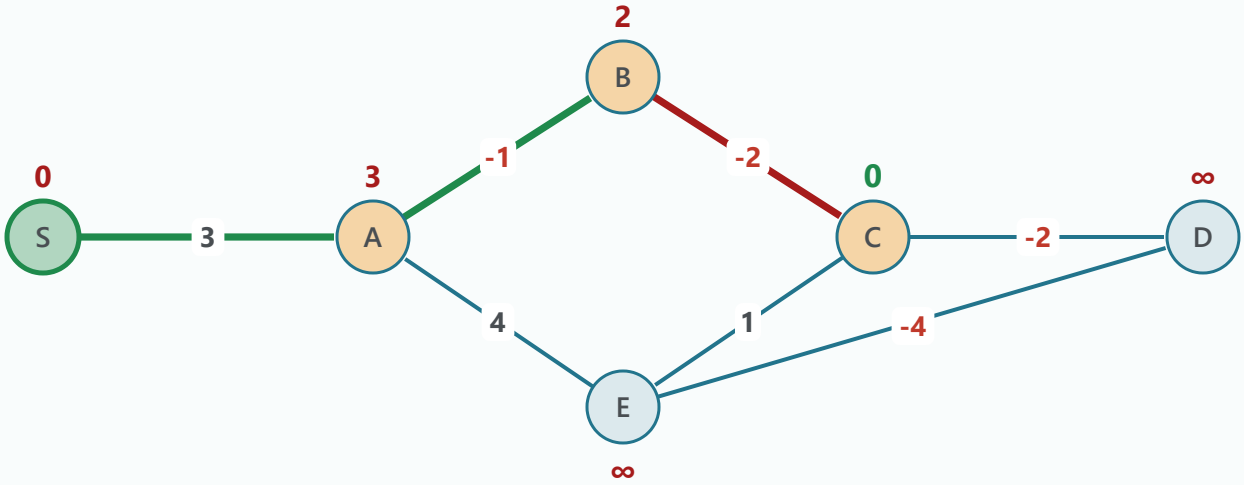
Bellman-Ford Example 2: Negative Edges



`(E,D) -4` `(E,A) 4` `(C,D) -2` `(C,E) 1` `(B,C) -2` `(A,B) -1` `(S,A) 3`

Pass 2: only (A,B): ∞ → **2**.

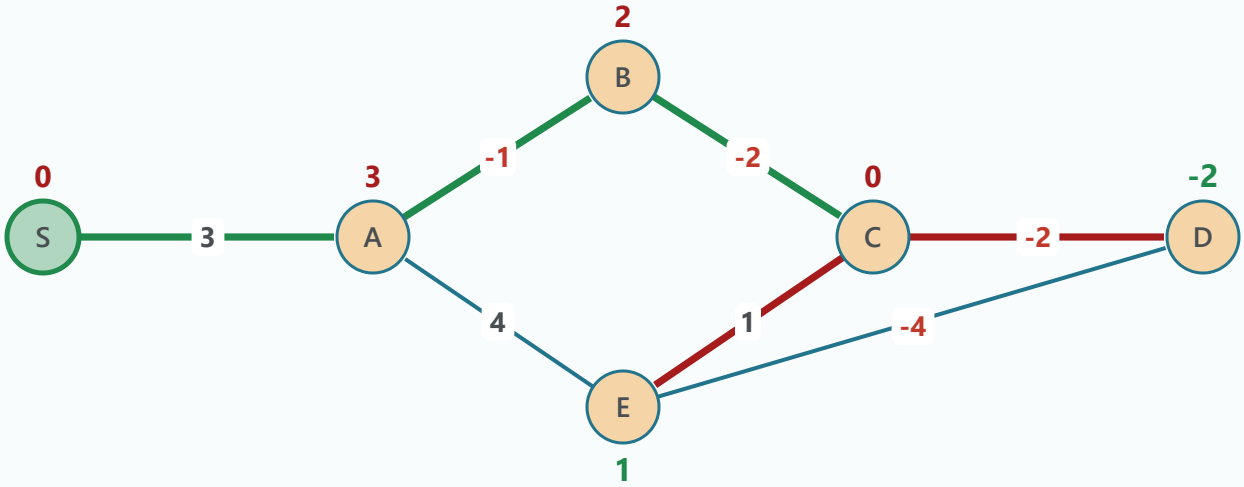
Bellman-Ford Example 2: Negative Edges



`(E,D) -4` `(E,A) 4` `(C,D) -2` `(C,E) 1` `(B,C) -2` `(A,B) -1` `(S,A) 3`

Pass 3: only (B,C): ∞ → **0**.

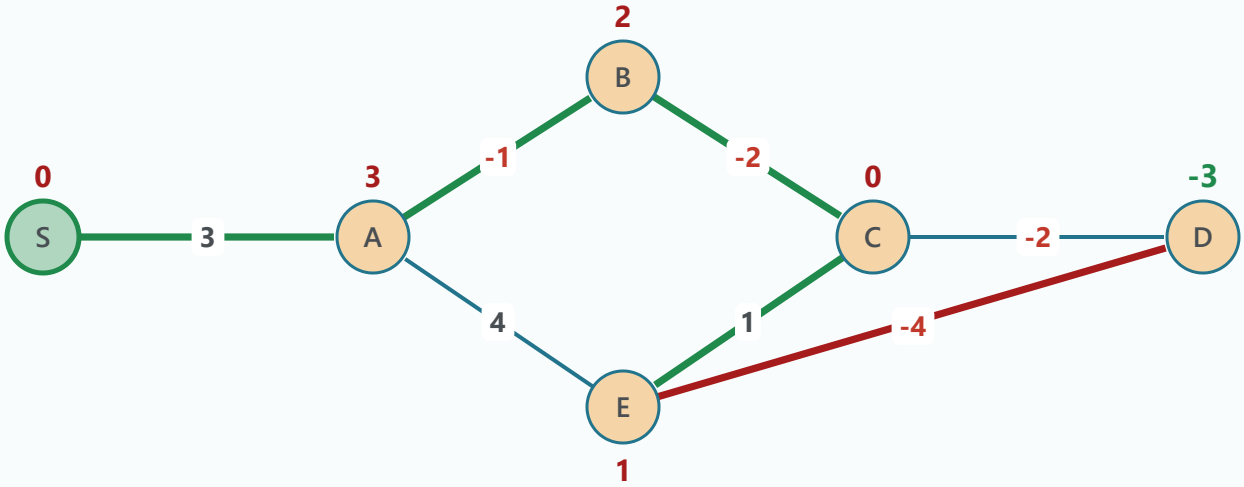
Bellman-Ford Example 2: Negative Edges



`(E,D) -4` `(E,A) 4` `(C,D) -2` `(C,E) 1` `(B,C) -2` `(A,B) -1` `(S,A) 3`

Pass 4: only (C,D): ∞ → **-2**; (C,E): ∞ → **1**.

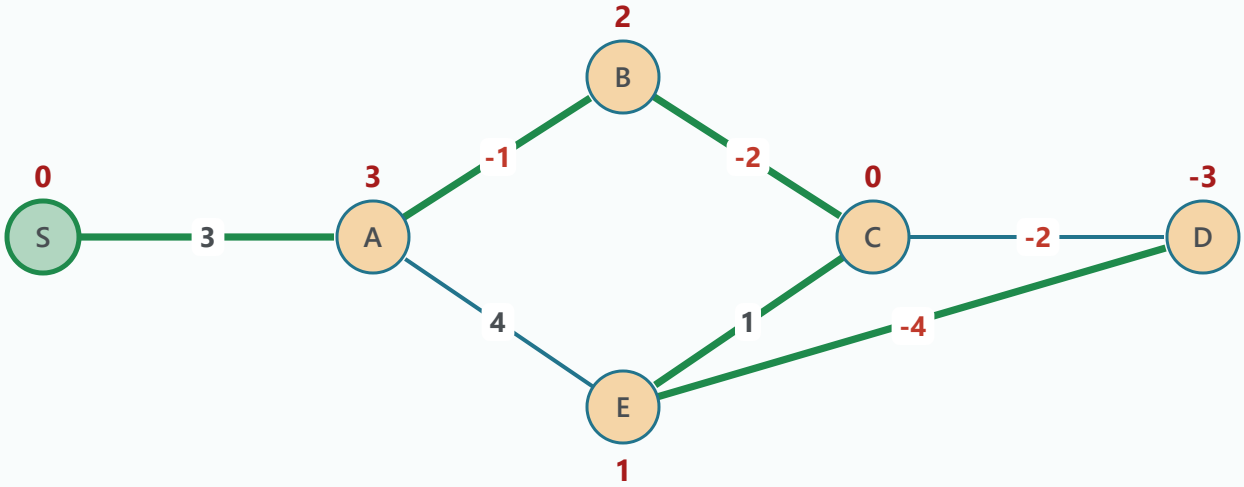
Bellman-Ford Example 2: Negative Edges



`(E,D) -4` `(E,A) 4` `(C,D) -2` `(C,E) 1` `(B,C) -2` `(A,B) -1` `(S,A) 3`

Pass 5: only (E,D): -2 → **-3**. Going through E gives D a shorter path: 1 + (−4) = −3.

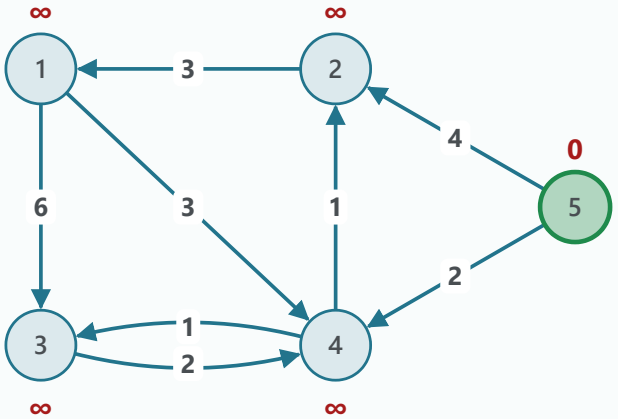
Bellman-Ford Example 2: Negative Edges



`(E,D) -4` `(E,A) 4` `(C,D) -2` `(C,E) 1` `(B,C) -2` `(A,B) -1` `(S,A) 3`

Check pass: no edge relaxes → no negative cycle (A→B→C→E→A = −1 −2 + 1 + 4 = +2). All V − 1 = 5 passes were needed.

Bellman-Ford Example 3: The Graph from Dijkstra Example 2

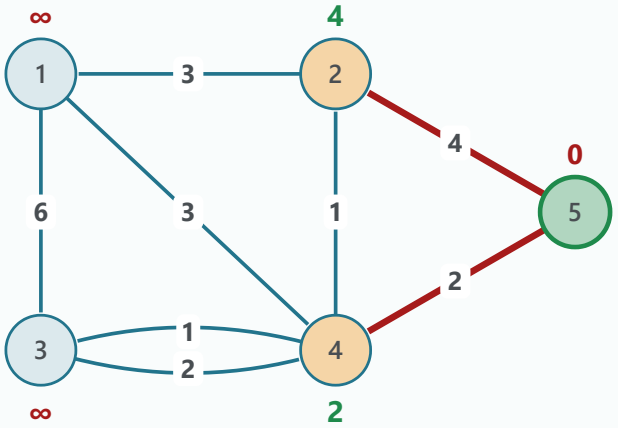


	1	2	3	4	5
d	∞	∞	∞	∞	0
π	/	/	/	/	/

`(1,3) 6` `(1,4) 3` `(2,1) 3` `(3,4) 2`
`(4,2) 1` `(4,3) 1` `(5,2) 4` `(5,4) 2`

Initialization: source 5 has distance 0, all others ∞. Edge order: (1,3), (1,4), (2,1), (3,4), (4,2), (4,3), (5,2), (5,4).

Bellman-Ford Example 3: The Graph from Dijkstra Example 2

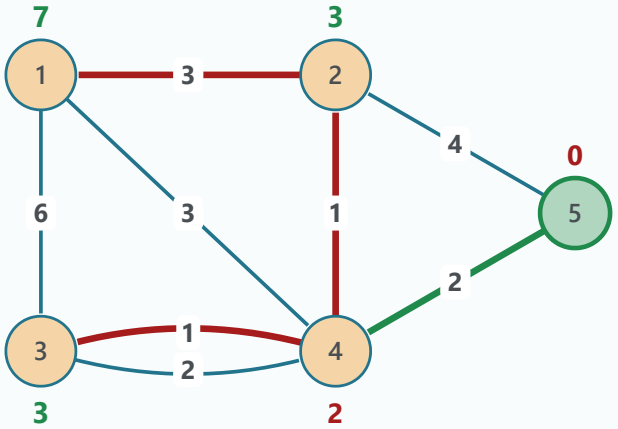


	1	2	3	4	5
d	∞	**4**	∞	**2**	0
π	/	**5**	/	**5**	/

`(1,3) 6` `(1,4) 3` `(2,1) 3` `(3,4) 2`
`(4,2) 1` `(4,3) 1` `(5,2) 4` `(5,4) 2`

Pass 1: (5,2): ∞ → **4**; (5,4): ∞ → **2**.

Bellman-Ford Example 3: The Graph from Dijkstra Example 2

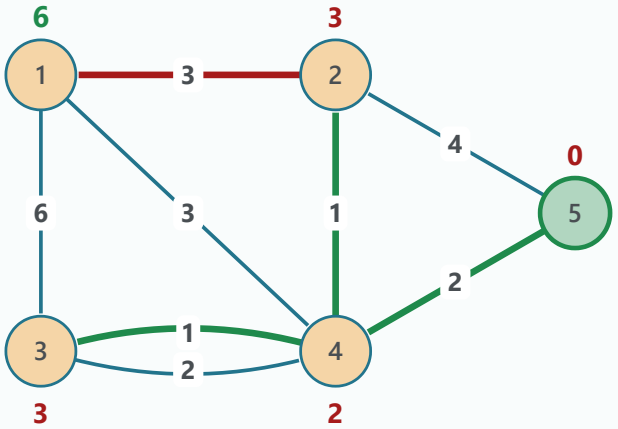


	1	2	3	4	5
d	7	3	3	2	0
π	2	4	4	5	/

`(1,3) 6` `(1,4) 3` `(2,1) 3` `(3,4) 2`
`(4,2) 1` `(4,3) 1` `(5,2) 4` `(5,4) 2`

Pass 2: (2,1): ∞ → **7**; (4,2): 4 → **3**; (4,3): ∞ → **3**. Edge (4,2) found a shorter path to 2.

Bellman-Ford Example 3: The Graph from Dijkstra Example 2

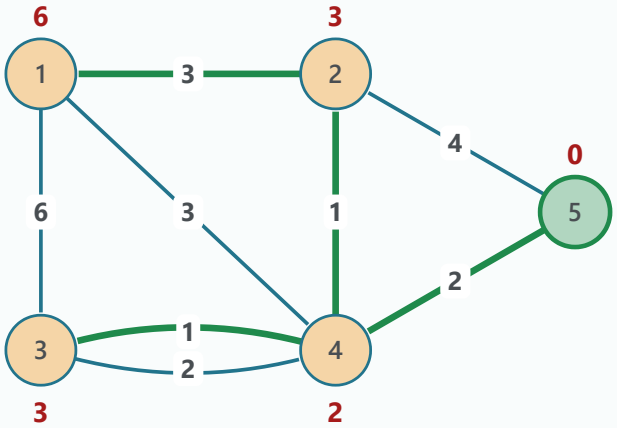


	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

`(1,3) 6` `(1,4) 3` `(2,1) 3` `(3,4) 2`
`(4,2) 1` `(4,3) 1` `(5,2) 4` `(5,4) 2`

Pass 3: (2,1): 7 → **6**. Since 2 improved in the previous pass, (2,1) relaxes again.

Bellman-Ford Example 3: The Graph from Dijkstra Example 2



	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

`(1,3) 6` `(1,4) 3` `(2,1) 3` `(3,4) 2`
`(4,2) 1` `(4,3) 1` `(5,2) 4` `(5,4) 2`

Pass 4: no shorter path found (no edge relaxes).

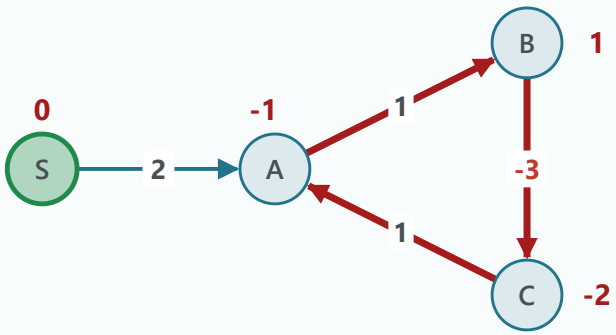
Checking for Negative Cycles (Pass V)

Try relaxing every edge one last time: if some edge still has **d[v] > d[u] + w(u, v)**, a negative cycle is reachable from the source. For the result of Example 3:

edge	d[v]	d[u] + w(u, v)	result
(1, 3)	d[3] = 3	d[1] + w = 6 + 6 = 12	3 ≯ 12 ✓
(1, 4)	d[4] = 2	d[1] + w = 6 + 3 = 9	2 ≯ 9 ✓
(2, 1)	d[1] = 6	d[2] + w = 3 + 3 = 6	6 ≯ 6 ✓
(3, 4)	d[4] = 2	d[3] + w = 3 + 2 = 5	2 ≯ 5 ✓
(4, 2)	d[2] = 3	d[4] + w = 2 + 1 = 3	3 ≯ 3 ✓
(4, 3)	d[3] = 3	d[4] + w = 2 + 1 = 3	3 ≯ 3 ✓
(5, 2)	d[2] = 3	d[5] + w = 0 + 4 = 4	3 ≯ 4 ✓
(5, 4)	d[4] = 2	d[5] + w = 0 + 2 = 2	2 ≯ 2 ✓

No edge relaxes → **no negative cycle**; d = [6, 3, 3, 2, 0] is final.

Detecting a Negative Cycle: Example



Cycle A → B → C → A: 1 + (−3) + 1 = **−1**.
Edge order: (S,A), (A,B), (B,C), (C,A).

pass	S	A	B	C
1	0	1	3	0
2	0	0	2	-1
3	0	-1	1	-2

- After V − 1 = 3 passes d = [0, −1, 1, −2]; the values on the cycle drop by 1 every pass.
- Check: for (A, B), d[A] + 1 = 0 < d[B] = 1 → **a negative cycle exists**.
- Vertices on the cycle, and those reachable from it, have shortest-path length −∞ (undefined).

Bellman-Ford: Pseudocode

```
BELLMAN_FORD(G, source):
  dist = [ ]
  for each v in V:
    dist[v] = infinity
  dist[source] = 0
  repeat for i = 1 to |V| - 1:
    for each (u, v, weight) in E:
      if dist[u] + weight < dist[v]:
        dist[v] = dist[u] + weight
  for each (u, v, weight) in E:
    if dist[u] + weight < dist[v]:
      error "Graph contains a negative-weight cycle"
  return dist
```

Two parts

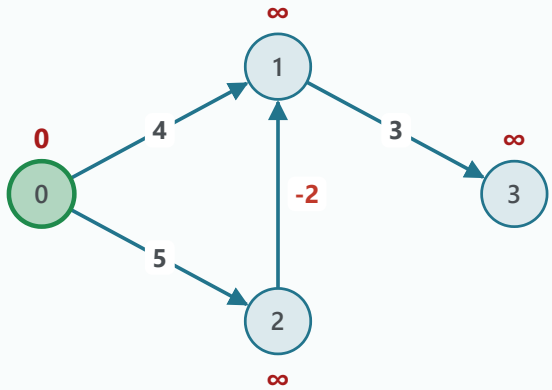
An outer loop of V − 1 passes over **all** edges, then a single check pass.

Edge list

Bellman-Ford never selects vertices, so storing the graph as an edge list E is enough.

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
▶    int[] dist = new int[V];
▶    int[] prev = new int[V];
▶    Arrays.fill(dist, INF);
▶    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
8        int u = e.from, v = e.to, w = e.weight;
9        if (dist[u] != INF
10           && dist[u] + w < dist[v]) {
11         dist[v] = dist[u] + w;
12         prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



Source 0. dist = [0, ∞, ∞, ∞]. Edge list: (2,1,−2), (0,1,4), (0,2,5), (1,3,3).

Step 1 / 19

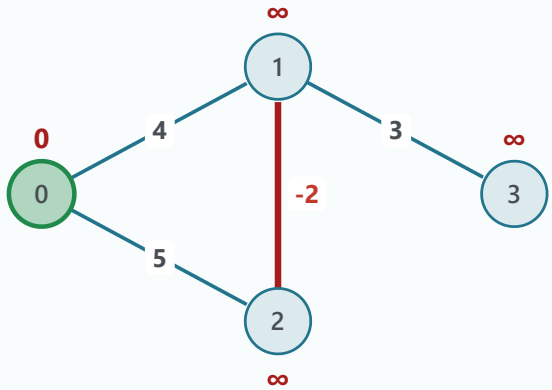
VARIABLES

`i = –` `u = –` `v = –` `w = –`

`dist = [0, ∞, ∞, ∞]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
▶        int u = e.from, v = e.to, w = e.weight;
▶        if (dist[u] != INF
▶            && dist[u] + w < dist[v]) {
11         dist[v] = dist[u] + w;
12         prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



Pass 1, edge (2,1): dist[2] = ∞ → condition false, skip.

Step 2 / 19

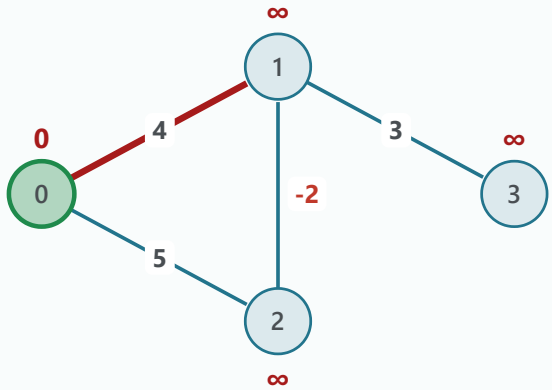
VARIABLES

`i = 1` `u = 2` `v = 1` `w = -2`

`dist = [0, ∞, ∞, ∞]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
▶        int u = e.from, v = e.to, w = e.weight;
▶        if (dist[u] != INF
▶            && dist[u] + w < dist[v]) {
11         dist[v] = dist[u] + w;
12         prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



Pass 1, edge (0,1): 0 + (4) = 4 < ∞ → condition true.

Step 3 / 19

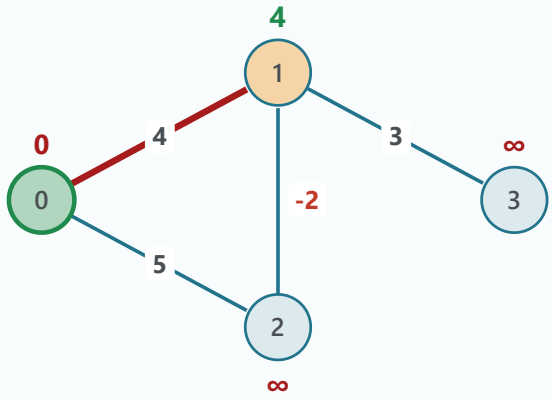
VARIABLES

`i = 1` `u = 0` `v = 1` `w = 4`

`dist = [0, ∞, ∞, ∞]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
8        int u = e.from, v = e.to, w = e.weight;
9        if (dist[u] != INF
10           && dist[u] + w < dist[v]) {
▶          dist[v] = dist[u] + w;
▶          prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



dist[1] = 4, prev[1] = 0.

Step 4 / 19

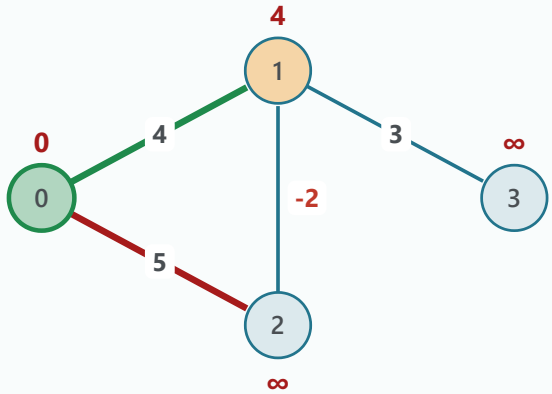
VARIABLES

`i = 1` `u = 0` `v = 1` `w = 4`

`dist = [0, 4, ∞, ∞]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
▶        int u = e.from, v = e.to, w = e.weight;
▶        if (dist[u] != INF
▶            && dist[u] + w < dist[v]) {
11         dist[v] = dist[u] + w;
12         prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



Pass 1, edge (0,2): 0 + (5) = 5 < ∞ → condition true.

Step 5 / 19

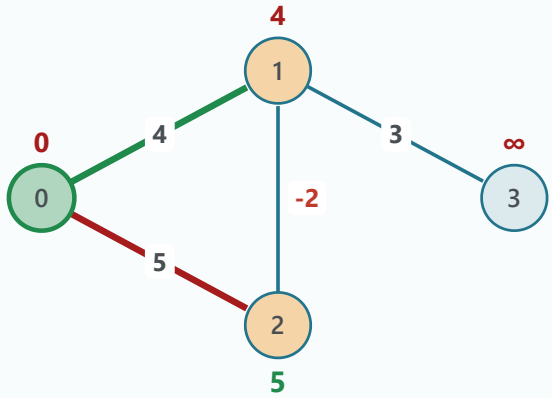
VARIABLES

`i = 1` `u = 0` `v = 2` `w = 5`

`dist = [0, 4, ∞, ∞]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
8        int u = e.from, v = e.to, w = e.weight;
9        if (dist[u] != INF
10           && dist[u] + w < dist[v]) {
▶          dist[v] = dist[u] + w;
▶          prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



dist[2] = 5, prev[2] = 0.

Step 6 / 19

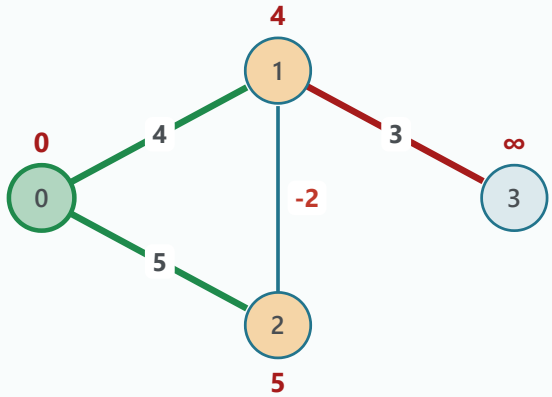
VARIABLES

`i = 1` `u = 0` `v = 2` `w = 5`

`dist = [0, 4, 5, ∞]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
▶        int u = e.from, v = e.to, w = e.weight;
▶        if (dist[u] != INF
▶            && dist[u] + w < dist[v]) {
11         dist[v] = dist[u] + w;
12         prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



Pass 1, edge (1,3): 4 + (3) = 7 < ∞ → condition true.

Step 7 / 19

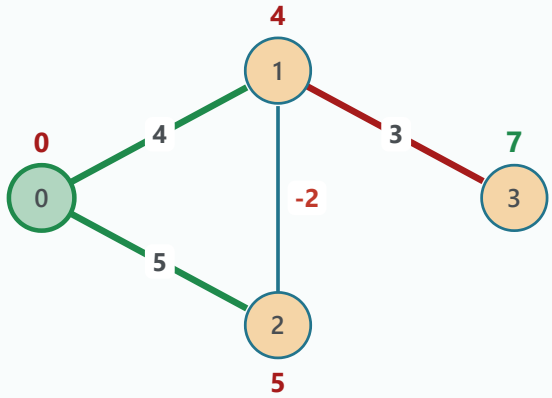
VARIABLES

`i = 1` `u = 1` `v = 3` `w = 3`

`dist = [0, 4, 5, ∞]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
8        int u = e.from, v = e.to, w = e.weight;
9        if (dist[u] != INF
10           && dist[u] + w < dist[v]) {
▶          dist[v] = dist[u] + w;
▶          prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



dist[3] = 7, prev[3] = 1. Pass 1 is over: edge (2,1) was wasted because it was examined while 2 was still ∞.

Step 8 / 19

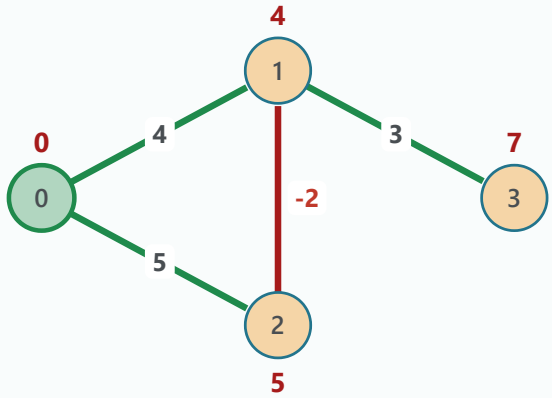
VARIABLES

`i = 1` `u = 1` `v = 3` `w = 3`

`dist = [0, 4, 5, 7]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
▶        int u = e.from, v = e.to, w = e.weight;
▶        if (dist[u] != INF
▶            && dist[u] + w < dist[v]) {
11         dist[v] = dist[u] + w;
12         prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



Pass 2, edge (2,1): 5 + (-2) = 3 < 4 → condition true.

Step 9 / 19

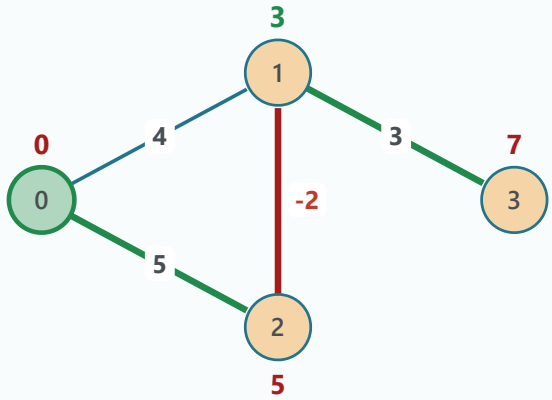
VARIABLES

`i = 2` `u = 2` `v = 1` `w = -2`

`dist = [0, 4, 5, 7]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
8        int u = e.from, v = e.to, w = e.weight;
9        if (dist[u] != INF
10           && dist[u] + w < dist[v]) {
▶          dist[v] = dist[u] + w;
▶          prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



dist[1] = 3, prev[1] = 2.

Step 10 / 19

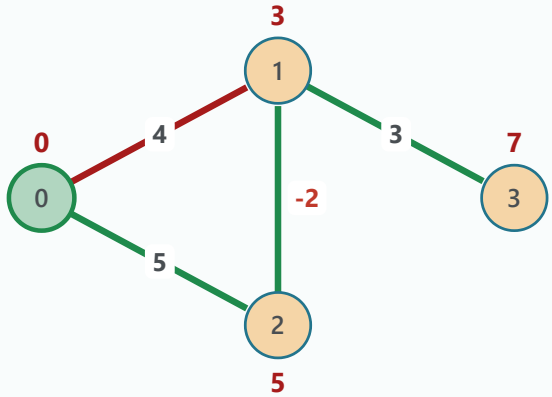
VARIABLES

`i = 2` `u = 2` `v = 1` `w = -2`

`dist = [0, 3, 5, 7]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
▶        int u = e.from, v = e.to, w = e.weight;
▶        if (dist[u] != INF
▶            && dist[u] + w < dist[v]) {
11         dist[v] = dist[u] + w;
12         prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



Pass 2, edge (0,1): 0 + (4) = 4 ≥ 3 → no change.

Step 11 / 19

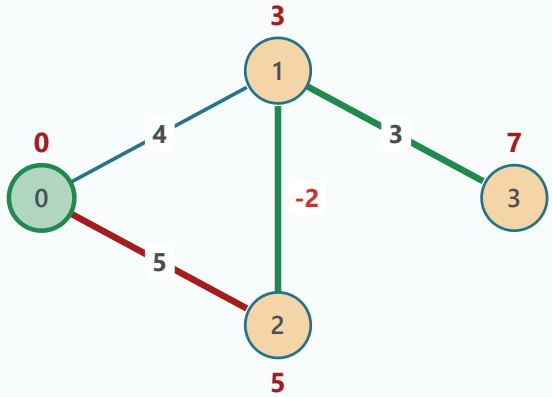
VARIABLES

`i = 2` `u = 0` `v = 1` `w = 4`

`dist = [0, 3, 5, 7]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
▶        int u = e.from, v = e.to, w = e.weight;
▶        if (dist[u] != INF
▶            && dist[u] + w < dist[v]) {
11         dist[v] = dist[u] + w;
12         prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



Pass 2, edge (0,2): 0 + (5) = 5 ≥ 5 → no change.

Step 12 / 19

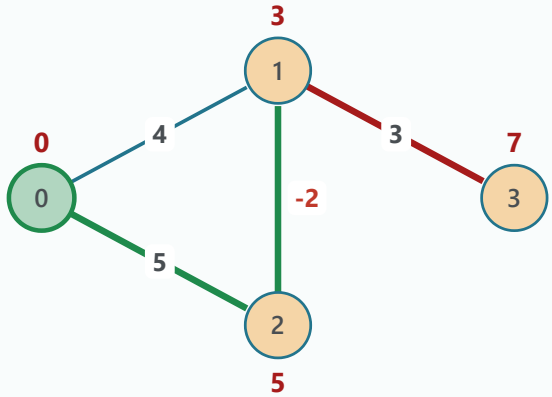
VARIABLES

`i = 2` `u = 0` `v = 2` `w = 5`

`dist = [0, 3, 5, 7]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
▶        int u = e.from, v = e.to, w = e.weight;
▶        if (dist[u] != INF
▶            && dist[u] + w < dist[v]) {
11         dist[v] = dist[u] + w;
12         prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



Pass 2, edge (1,3): 3 + (3) = 6 < 7 → condition true.

Step 13 / 19

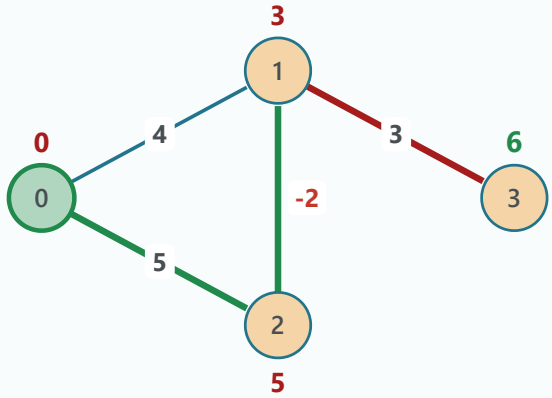
VARIABLES

`i = 2` `u = 1` `v = 3` `w = 3`

`dist = [0, 3, 5, 7]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
8        int u = e.from, v = e.to, w = e.weight;
9        if (dist[u] != INF
10           && dist[u] + w < dist[v]) {
▶          dist[v] = dist[u] + w;
▶          prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



dist[3] = 6, prev[3] = 1. Pass 2 is over: the negative edge brought 1 down to 3, and that brought 3 down to 6.

Step 14 / 19

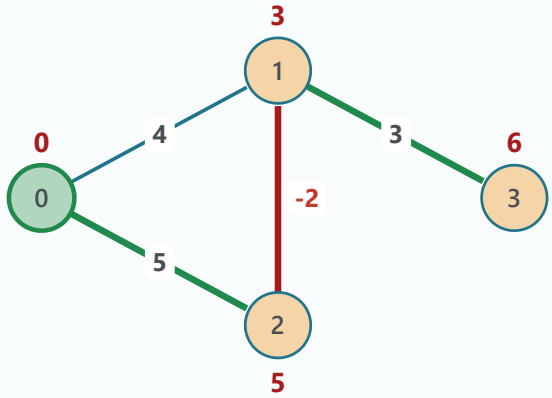
VARIABLES

`i = 2` `u = 1` `v = 3` `w = 3`

`dist = [0, 3, 5, 6]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
▶        int u = e.from, v = e.to, w = e.weight;
▶        if (dist[u] != INF
▶            && dist[u] + w < dist[v]) {
11         dist[v] = dist[u] + w;
12         prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



Pass 3, edge (2,1): 5 + (-2) = 3 ≥ 3 → no change.

Step 15 / 19

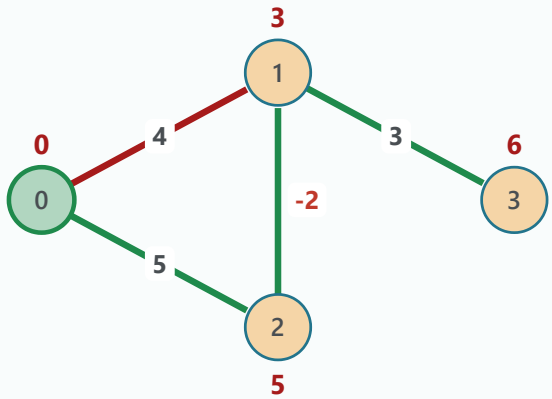
VARIABLES

`i = 3` `u = 2` `v = 1` `w = -2`

`dist = [0, 3, 5, 6]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
▶        int u = e.from, v = e.to, w = e.weight;
▶        if (dist[u] != INF
▶            && dist[u] + w < dist[v]) {
11         dist[v] = dist[u] + w;
12         prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



Pass 3, edge (0,1): 0 + (4) = 4 ≥ 3 → no change.

Step 16 / 19

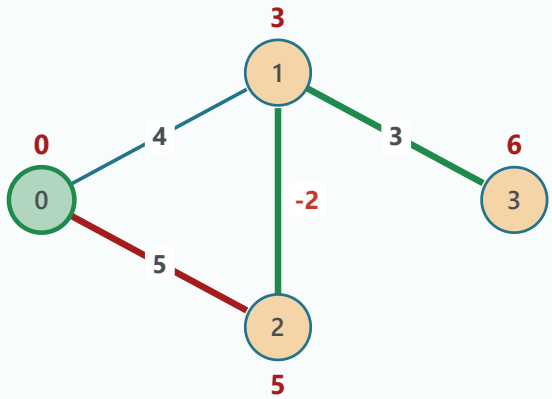
VARIABLES

`i = 3` `u = 0` `v = 1` `w = 4`

`dist = [0, 3, 5, 6]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
▶        int u = e.from, v = e.to, w = e.weight;
▶        if (dist[u] != INF
▶            && dist[u] + w < dist[v]) {
11         dist[v] = dist[u] + w;
12         prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



Pass 3, edge (0,2): 0 + (5) = 5 ≥ 5 → no change.

Step 17 / 19

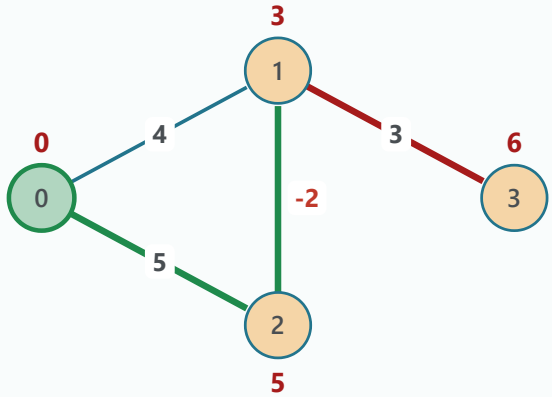
VARIABLES

`i = 3` `u = 0` `v = 2` `w = 5`

`dist = [0, 3, 5, 6]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
▶        int u = e.from, v = e.to, w = e.weight;
▶        if (dist[u] != INF
▶            && dist[u] + w < dist[v]) {
11         dist[v] = dist[u] + w;
12         prev[v] = u;
13       }
14     }
15   }
16   for (Edge e : edges) { // check
17     int u = e.from, v = e.to, w = e.weight;
18     if (dist[u] != INF
19         && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



Pass 3, edge (1,3): 3 + (3) = 6 ≥ 6 → no change.

Step 18 / 19

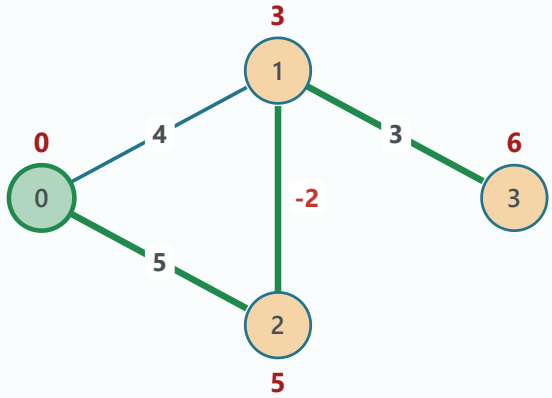
VARIABLES

`i = 3` `u = 1` `v = 3` `w = 3`

`dist = [0, 3, 5, 6]`

Bellman-Ford: Step Through the Code

```
1  void bellmanFord(int src) {
2    int[] dist = new int[V];
3    int[] prev = new int[V];
4    Arrays.fill(dist, INF);
5    dist[src] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Edge e : edges) {
8        int u = e.from, v = e.to, w = e.weight;
9        if (dist[u] != INF
10           && dist[u] + w < dist[v]) {
11         dist[v] = dist[u] + w;
12         prev[v] = u;
13       }
14     }
15   }
▶    for (Edge e : edges) { // check
▶      int u = e.from, v = e.to, w = e.weight;
▶      if (dist[u] != INF
▶          && dist[u] + w < dist[v])
20       throw new IllegalStateException(
21           "Negative cycle");
22   }
23 }
```



Pass 3 changed nothing. Check pass (lines 16–20): no edge relaxes and no exception is thrown. Result [0, 3, 5, 6].

Step 19 / 19

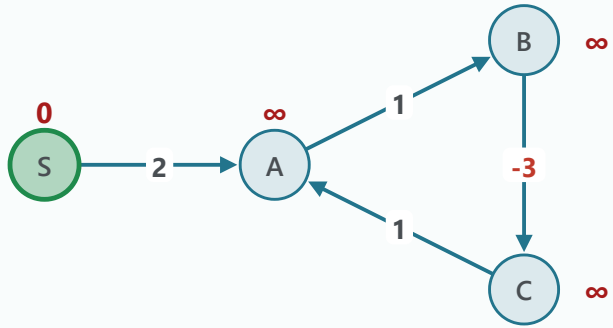
VARIABLES

i = – u = 1 v = 3 w = 3

dist = [0, 3, 5, 6]

Bellman-Ford: Tracing the Negative-Cycle Check

```
▶ for (int i = 1; i < V; i++) {
2   for (Edge e : edges) {
3     int u = e.from, v = e.to, w = e.weight;
4     if (dist[u] != INF
5         && dist[u] + w < dist[v]) {
6       dist[v] = dist[u] + w;
7       prev[v] = u;
8     }
9   }
10 }
11 for (Edge e : edges) { // check
12   int u = e.from, v = e.to, w = e.weight;
13   if (dist[u] != INF
14       && dist[u] + w < dist[v])
15     throw new IllegalStateException(
16         "Negative cycle");
17 }
```



start

S → A (2) and the cycle A → B → C → A: 1 + (−3) + 1 = −1. dist = [S, A, B, C] = [0, ∞, ∞, ∞]; the edges SA, AB, BC, CA are relaxed in that order.

Step 1 / 9

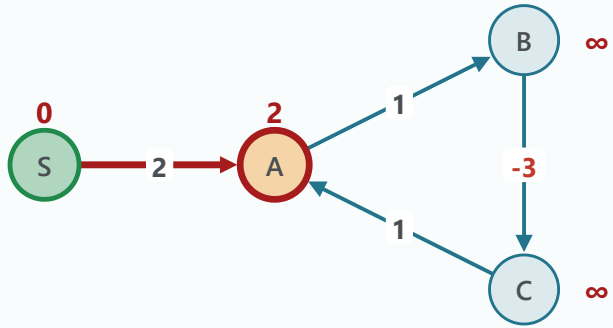
VARIABLES

`i = –` `u = –` `v = –` `w = –`

`dist = [0, ∞, ∞, ∞]`

Bellman-Ford: Tracing the Negative-Cycle Check

```
1  for (int i = 1; i < V; i++) {
2    for (Edge e : edges) {
▶      int u = e.from, v = e.to, w = e.weight;
▶      if (dist[u] != INF
▶          && dist[u] + w < dist[v]) {
▶        dist[v] = dist[u] + w;
▶        prev[v] = u;
8      }
9    }
10 }
11 for (Edge e : edges) { // check
12   int u = e.from, v = e.to, w = e.weight;
13   if (dist[u] != INF
14       && dist[u] + w < dist[v])
15     throw new IllegalStateException(
16         "Negative cycle");
17 }
```



pass 1

Pass 1, edge SA: 0 + (2) = 2 is smaller → dist[A] = 2.

Step 2 / 9

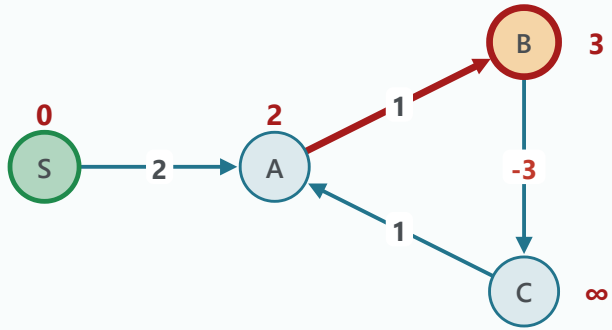
VARIABLES

`i = 1` `u = S` `v = A` `w = 2`

`dist = [0, 2, ∞, ∞]`

Bellman-Ford: Tracing the Negative-Cycle Check

```
1  for (int i = 1; i < V; i++) {
2    for (Edge e : edges) {
▶      int u = e.from, v = e.to, w = e.weight;
▶      if (dist[u] != INF
▶          && dist[u] + w < dist[v]) {
▶        dist[v] = dist[u] + w;
▶        prev[v] = u;
8      }
9    }
10 }
11 for (Edge e : edges) { // check
12   int u = e.from, v = e.to, w = e.weight;
13   if (dist[u] != INF
14       && dist[u] + w < dist[v])
15     throw new IllegalStateException(
16         "Negative cycle");
17 }
```



pass 1

Pass 1, edge AB: 2 + (1) = 3 is smaller → dist[B] = 3.

Step 3 / 9

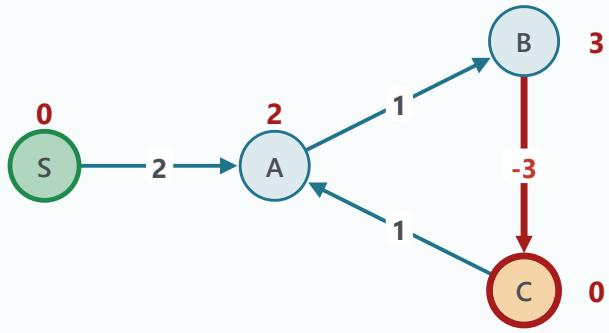
VARIABLES

`i = 1` `u = A` `v = B` `w = 1`

`dist = [0, 2, 3, ∞]`

Bellman-Ford: Tracing the Negative-Cycle Check

```
1  for (int i = 1; i < V; i++) {
2    for (Edge e : edges) {
▶      int u = e.from, v = e.to, w = e.weight;
▶      if (dist[u] != INF
▶          && dist[u] + w < dist[v]) {
▶        dist[v] = dist[u] + w;
▶        prev[v] = u;
8      }
9    }
10 }
11 for (Edge e : edges) { // check
12   int u = e.from, v = e.to, w = e.weight;
13   if (dist[u] != INF
14       && dist[u] + w < dist[v])
15     throw new IllegalStateException(
16         "Negative cycle");
17 }
```



pass 1

Pass 1, edge BC: 3 + (-3) = 0 is smaller → dist[C] = 0.

Step 4 / 9

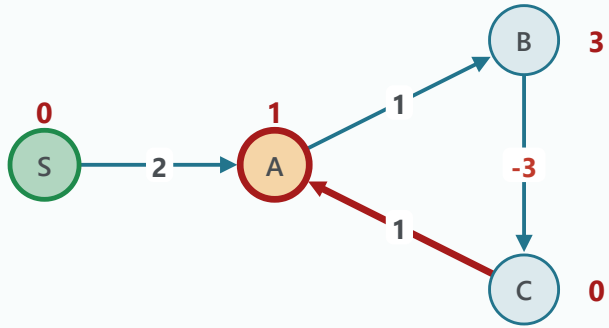
VARIABLES

`i = 1` `u = B` `v = C` `w = -3`

`dist = [0, 2, 3, 0]`

Bellman-Ford: Tracing the Negative-Cycle Check

```
1  for (int i = 1; i < V; i++) {
2    for (Edge e : edges) {
▶      int u = e.from, v = e.to, w = e.weight;
▶      if (dist[u] != INF
▶          && dist[u] + w < dist[v]) {
▶        dist[v] = dist[u] + w;
▶        prev[v] = u;
8      }
9    }
10 }
11 for (Edge e : edges) { // check
12   int u = e.from, v = e.to, w = e.weight;
13   if (dist[u] != INF
14       && dist[u] + w < dist[v])
15     throw new IllegalStateException(
16         "Negative cycle");
17 }
```



pass 1

Pass 1, edge CA: 0 + (1) = 1 is smaller → dist[A] = 1. Going around the cycle brought A down from 2 to 1.

Step 5 / 9

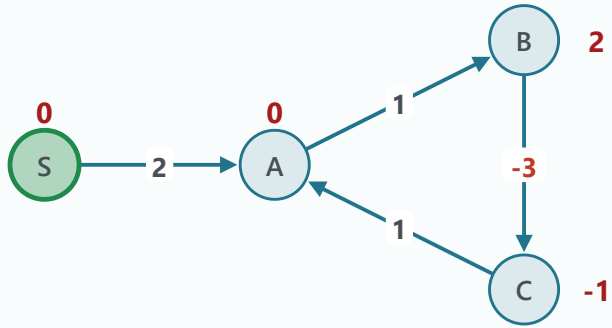
VARIABLES

`i = 1` `u = C` `v = A` `w = 1`

`dist = [0, 1, 3, 0]`

Bellman-Ford: Tracing the Negative-Cycle Check

```
for (int i = 1; i < V; i++) {
  for (Edge e : edges) {
    int u = e.from, v = e.to, w = e.weight;
    if (dist[u] != INF
        && dist[u] + w < dist[v]) {
      dist[v] = dist[u] + w;
      prev[v] = u;
    }
  }
}
for (Edge e : edges) { // check
  int u = e.from, v = e.to, w = e.weight;
  if (dist[u] != INF
      && dist[u] + w < dist[v])
    throw new IllegalStateException(
        "Negative cycle");
}
```



pass 2

Pass 2 is over: three of the four edges relaxed again (B = 2, C = -1, A = 0). The values on the cycle drop by 1 every pass.

Step 6 / 9

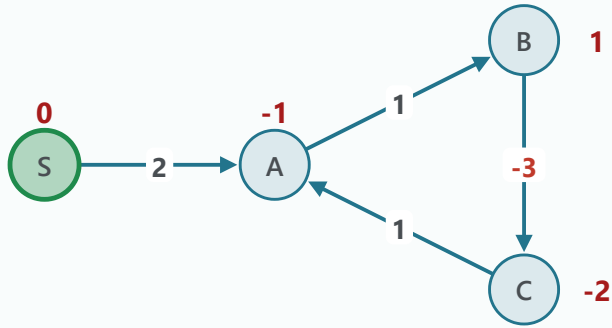
VARIABLES

`i = 2` `u = C` `v = A` `w = 1`

`dist = [0, 0, 2, -1]`

Bellman-Ford: Tracing the Negative-Cycle Check

```
for (int i = 1; i < V; i++) {
  for (Edge e : edges) {
    int u = e.from, v = e.to, w = e.weight;
    if (dist[u] != INF
        && dist[u] + w < dist[v]) {
      dist[v] = dist[u] + w;
      prev[v] = u;
    }
  }
}
for (Edge e : edges) { // check
  int u = e.from, v = e.to, w = e.weight;
  if (dist[u] != INF
      && dist[u] + w < dist[v])
    throw new IllegalStateException(
        "Negative cycle");
}
```



pass 3

Pass 3 is over: three of the four edges relaxed again (B = 1, C = -2, A = -1). The values on the cycle drop by 1 every pass.

Step 7 / 9

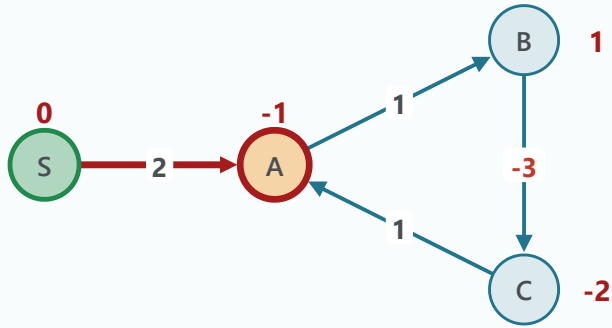
VARIABLES

`i = 3` `u = C` `v = A` `w = 1`

`dist = [0, -1, 1, -2]`

Bellman-Ford: Tracing the Negative-Cycle Check

```
1  for (int i = 1; i < V; i++) {
2    for (Edge e : edges) {
3      int u = e.from, v = e.to, w = e.weight;
4      if (dist[u] != INF
5          && dist[u] + w < dist[v]) {
6        dist[v] = dist[u] + w;
7        prev[v] = u;
8      }
9    }
10 }
▶ for (Edge e : edges) { // check
▶   int u = e.from, v = e.to, w = e.weight;
▶   if (dist[u] != INF
▶       && dist[u] + w < dist[v])
15    throw new IllegalStateException(
16        "Negative cycle");
17 }
```



check pass

V − 1 = 3 passes are over; the check pass begins. Edge SA: 0 + 2 = 2 is not < -1, fine.

Step 8 / 9

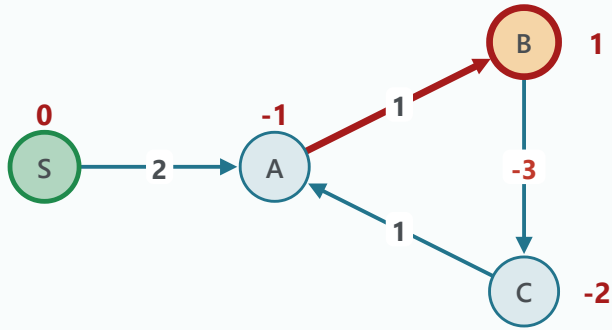
VARIABLES

`i = –` `u = S` `v = A` `w = 2`

`dist = [0, -1, 1, -2]`

Bellman-Ford: Tracing the Negative-Cycle Check

```
1  for (int i = 1; i < V; i++) {
2    for (Edge e : edges) {
3      int u = e.from, v = e.to, w = e.weight;
4      if (dist[u] != INF
5          && dist[u] + w < dist[v]) {
6        dist[v] = dist[u] + w;
7        prev[v] = u;
8      }
9    }
10 }
11 for (Edge e : edges) { // check
▶    int u = e.from, v = e.to, w = e.weight;
▶    if (dist[u] != INF
▶        && dist[u] + w < dist[v])
▶      throw new IllegalStateException(
▶          "Negative cycle");
17 }
```



check pass **negative cycle!**

Check, edge AB: -1 + 1 = 0 < 1 → it still relaxes after V − 1 passes: **a negative cycle exists**, and the exception is thrown.

Step 9 / 9

VARIABLES

`i = –` `u = A` `v = B` `w = 1`

`dist = [0, -1, 1, -2]`

The Floyd-Warshall Algorithm

- Finds shortest paths between **all pairs of vertices**.
- Found independently by Bernard Roy (1959), Stephen **Warshall** (1962, transitive closure) and Robert **Floyd** (1962, shortest paths).
- Works with **negative edge weights** and **detects** negative cycles (in which case shortest paths are undefined).
- Uses **dynamic programming**, keeping the results in a V × V matrix D.

Single-source or all-pairs?

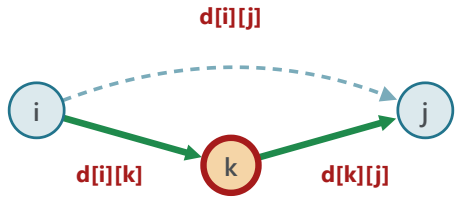
Bellman-Ford and Dijkstra find shortest paths **from one source**. Floyd-Warshall computes shortest paths between **all pairs** at once with three nested loops. The code is tiny, and it is practical on dense graphs.

Floyd-Warshall: Steps and Recurrence

1. **Step 1:** Fill D with the edge weights: D[i][i] = 0, and ∞ where there is no direct edge.
2. **Step 2:** Take intermediate vertices k = 1, 2, …, n in turn; for every pair (i, j)…
3. **Step 3:** Compute the length of the path through k, D[i][k] + D[k][j], and compare it with the current D[i][j].
4. **Step 4:** If it is shorter, write the new value into the matrix (and update Π as well).

Subproblem

$D^{(k)}[i][j]$: the length of a shortest i ⇝ j path whose intermediate vertices all come from {1, …, k}.



$D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$

Either k is not on the path, or the path passes through k once. $D^{(n)}$ is the answer.

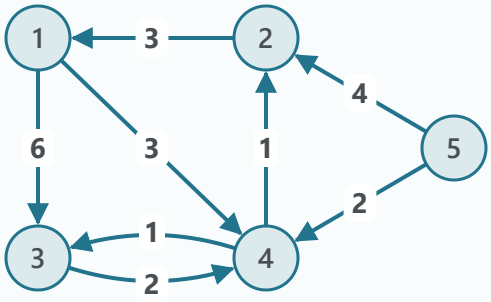
Floyd-Warshall: Complexity

- Time **$\Theta(V^3)$**: three nested for loops, each running V times.
  - the first (outermost) loop runs over all **intermediate** vertices (k),
  - the second over all **source** vertices (i),
  - the last over all **target** vertices (j).
- Memory **$\Theta(V^2)$**: D can be updated in place; row and column k do not change during pass k.

For all pairs	Time
Floyd-Warshall	$O(V^3)$
V × Dijkstra (array)	$O(V^3)$
V × Dijkstra (heap)	$O(V(V + E) \log V)$
V × Bellman-Ford	$O(V^2 E)$

Loop order matters: **k must be outermost**. With k inside, the result is wrong.

Floyd-Warshall Example: The D and Π Matrices



$D^{(0)}$

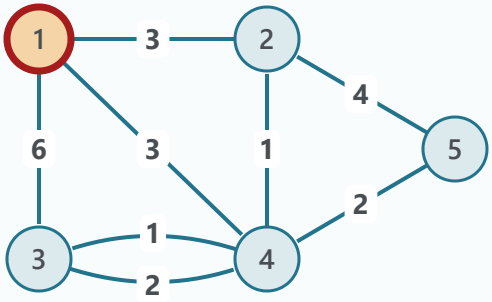
	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	∞	∞	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

$\Pi^{(0)}$

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	/	/	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

k = 0 (initialization): fill D with edge weights, 0 on the diagonal, ∞ where there is no edge. Π[i][j] = i if there is a direct edge, otherwise /.

Floyd-Warshall Example: The D and Π Matrices



$D^{(1)}$

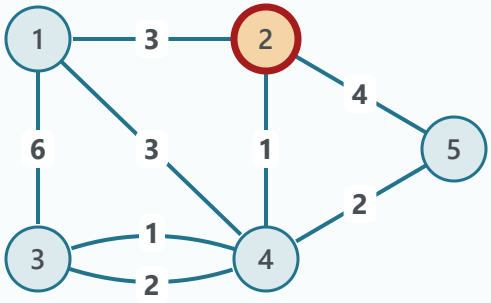
	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	**9**	**6**	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

$\Pi^{(1)}$

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	**1**	**1**	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

k = 1: via vertex 1 we find the paths **2** ⇝ **3** (3 + 6 = 9) and **2** ⇝ **4** (3 + 3 = 6).

Floyd-Warshall Example: The D and Π Matrices



$D^{(2)}$

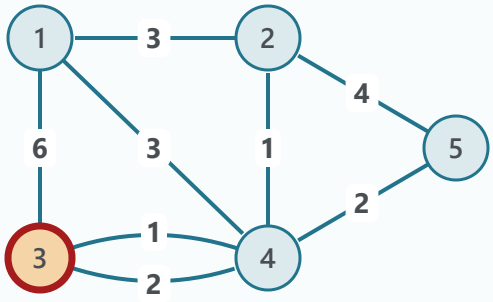
	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	**4**	1	1	0	∞
5	**7**	4	**13**	2	0

$\Pi^{(2)}$

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	**2**	4	4	/	/
5	**2**	5	**1**	5	/

k = 2: via vertex 2 we find **4** ⇝ **1** (1 + 3 = 4), **5** ⇝ **1** (4 + 3 = 7) and **5** ⇝ **3** (4 + 9 = 13).

Floyd-Warshall Example: The D and Π Matrices



$D^{(3)}$

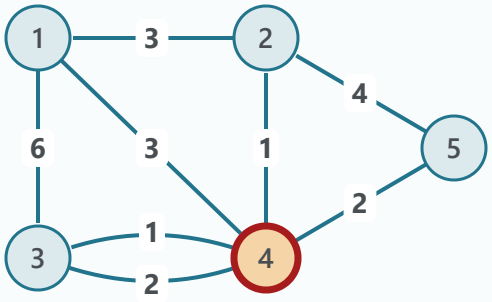
	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	4	1	1	0	∞
5	7	4	13	2	0

$\Pi^{(3)}$

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	2	4	4	/	/
5	2	5	1	5	/

k = 3: no shorter path via vertex 3 (3 has an edge only to 4, and no value improves).

Floyd-Warshall Example: The D and Π Matrices



$D^{(4)}$

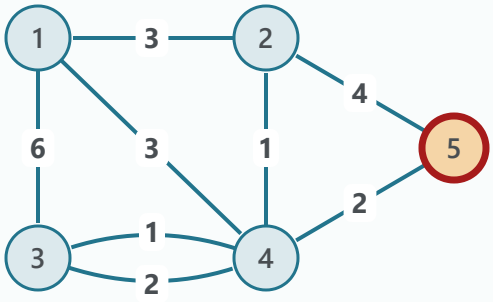
	1	2	3	4	5
1	0	**4**	**4**	3	∞
2	3	0	**7**	6	∞
3	**6**	**3**	0	2	∞
4	4	1	1	0	∞
5	**6**	**3**	**3**	2	0

$\Pi^{(4)}$

	1	2	3	4	5
1	/	**4**	**4**	1	/
2	2	/	**4**	1	/
3	**2**	**4**	/	3	/
4	2	4	4	/	/
5	**2**	**4**	**4**	5	/

k = 4: via vertex 4, the paths 1⇝2, 1⇝3, 2⇝3, 3⇝1, 3⇝2, 5⇝1, 5⇝2, 5⇝3 get shorter (8 cells). E.g. 5⇝3: 2 + 1 = 3 < 13.

Floyd-Warshall Example: The D and Π Matrices



$D^{(5)}$

	1	2	3	4	5
1	0	4	4	3	∞
2	3	0	7	6	∞
3	6	3	0	2	∞
4	4	1	1	0	∞
5	6	3	3	2	0

$\Pi^{(5)}$

	1	2	3	4	5
1	/	4	4	1	/
2	2	/	4	1	/
3	2	4	/	3	/
4	2	4	4	/	/
5	2	4	4	5	/

k = 5: no shorter path via vertex 5 (no edge enters 5, so D[i][5] = ∞). The final matrices are ready.

Final State and Reading a Path

D (distances)

	1	2	3	4	5
1	0	4	4	3	∞
2	3	0	7	6	∞
3	6	3	0	2	∞
4	4	1	1	0	∞
5	6	3	3	2	0

Π (predecessors)

	1	2	3	4	5
1	/	4	4	1	/
2	2	/	4	1	/
3	2	4	/	3	/
4	2	4	4	/	/
5	2	4	4	5	/

- Π[i][j]: the vertex **just before j** on a shortest path from i to j.
- 5 ⇝ 1: Π[5][1] = 2, Π[5][2] = 4, Π[5][4] = 5 → **5** → **4** → **2** → **1**, D[5][1] = 6.
- 1 ⇝ 2: Π[1][2] = 4, Π[1][4] = 1 → **1** → **4** → **2**, D[1][2] = 4.
- Column 5 is ∞: no vertex can reach 5.
- Row 5 matches the [6, 3, 3, 2, 0] that Dijkstra and Bellman-Ford found for source 5.

Floyd-Warshall: Pseudocode

1. Initialization

```
FLOYD_WARSHALL(G):
  dist = [ ][ ]
  for each u in V:
    for each v in V:
      if u == v:
        dist[u][v] = 0
      else if (u, v) in E:
        dist[u][v] = weight(u, v)
      else:
        dist[u][v] = infinity
```

2. Three nested loops and the check

```
  for each k in V:
    for each i in V:
      for each j in V:
        if dist[i][k] + dist[k][j]
             < dist[i][j]:
          dist[i][j] =
            dist[i][k] + dist[k][j]
  for each i in V:
    if dist[i][i] < 0:
      error "Graph contains a negative-weight cycle"
  return dist
```

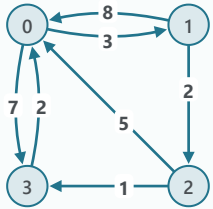
Floyd-Warshall: Java

```
void floydWarshall(int[][] d) {  // V x V
  for (int k = 0; k < V; k++)       // via
    for (int i = 0; i < V; i++)     // from
      for (int j = 0; j < V; j++)   // to
        if (d[i][k] != INF
            && d[k][j] != INF
            && d[i][k] + d[k][j] < d[i][j]) {
          d[i][j] = d[i][k] + d[k][j];
          prev[i][j] = prev[k][j];
        }
  for (int i = 0; i < V; i++)
    if (d[i][i] < 0)
      throw new IllegalStateException(
          "Negative cycle");
}
```

- The matrix is updated **in place**; no separate copy of $D^{(k)}$ is needed.
- The INF check prevents overflow when adding to `Integer.MAX_VALUE`.
- `prev[i][j] = prev[k][j]`: on the new path, the vertex before j is the same as on the k ⇝ j path.
- Negative cycle: some vertex has a negative distance **to itself** (D[i][i] < 0).

Floyd-Warshall: Step Through the Code

```
▶ void floydWarshall(int[][] d) {  // V x V
2   for (int k = 0; k < V; k++)        // via
3     for (int i = 0; i < V; i++)      // from
4       for (int j = 0; j < V; j++)    // to
5         if (d[i][k] != INF
6             && d[k][j] != INF
7             && d[i][k] + d[k][j] < d[i][j]) {
8           d[i][j] = d[i][k] + d[k][j];
9           prev[i][j] = prev[k][j];
10        }
11  for (int i = 0; i < V; i++)
12    if (d[i][i] < 0)
13      throw new IllegalStateException(
14          "Negative cycle");
15 }
```



d

	0	1	2	3
0	0	3	∞	7
1	8	0	2	∞
2	5	∞	0	1
3	2	∞	∞	0

prev

	0	1	2	3
0	/	0	/	0
1	1	/	1	/
2	2	/	/	2
3	3	/	/	/

Start: d holds the edge weights (0 on the diagonal, ∞ where there is no edge); prev[i][j] = i. The matrix on the right is d, the one below it prev.

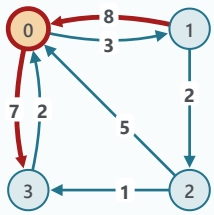
Step 1 / 13

VARIABLES

k = – i = – j = –

Floyd-Warshall: Step Through the Code

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)        // via
3      for (int i = 0; i < V; i++)      // from
4        for (int j = 0; j < V; j++)    // to
▶          if (d[i][k] != INF
▶              && d[k][j] != INF
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            prev[i][j] = prev[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negative cycle");
15 }
```



d

	0	1	2	3
0	0	3	∞	7
1	8	0	2	15
2	5	∞	0	1
3	2	∞	∞	0

prev

	0	1	2	3
0	/	0	/	0
1	1	/	1	0
2	2	/	/	2
3	3	/	/	/

gray: row and column k = 0

k = 0 as intermediate: d[1][0] + d[0][3] = 8 + 7 = 15 < ∞ → d[1][3] = 15, prev[1][3] = prev[0][3] = 0.

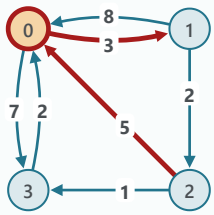
Step 2 / 13

VARIABLES

`k = 0` `i = 1` `j = 3` `d[i][j] = 15`

Floyd-Warshall: Step Through the Code

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)        // via
3      for (int i = 0; i < V; i++)      // from
4        for (int j = 0; j < V; j++)    // to
▶          if (d[i][k] != INF
▶              && d[k][j] != INF
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            prev[i][j] = prev[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negative cycle");
15 }
```



d

	0	1	2	3
0	0	3	∞	7
1	8	0	2	15
2	5	8	0	1
3	2	∞	∞	0

prev

	0	1	2	3
0	/	0	/	0
1	1	/	1	0
2	2	0	/	2
3	3	/	/	/

gray: row and column k = 0

k = 0 as intermediate: d[2][0] + d[0][1] = 5 + 3 = 8 < ∞ → d[2][1] = 8, prev[2][1] = prev[0][1] = 0.

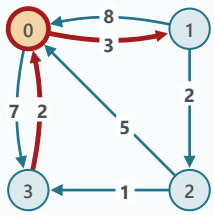
Step 3 / 13

VARIABLES

`k = 0` `i = 2` `j = 1` `d[i][j] = 8`

Floyd-Warshall: Step Through the Code

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)        // via
3      for (int i = 0; i < V; i++)      // from
4        for (int j = 0; j < V; j++)    // to
▶          if (d[i][k] != INF
▶              && d[k][j] != INF
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            prev[i][j] = prev[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negative cycle");
15 }
```



d

	0	1	2	3
0	0	3	∞	7
1	8	0	2	15
2	5	8	0	1
3	2	5	∞	0

prev

	0	1	2	3
0	/	0	/	0
1	1	/	1	0
2	2	0	/	2
3	3	0	/	/

gray: row and column k = 0

k = 0 as intermediate: d[3][0] + d[0][1] = 2 + 3 = 5 < ∞ → d[3][1] = 5, prev[3][1] = prev[0][1] = 0.

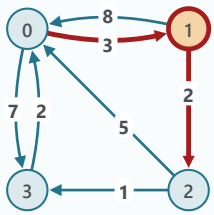
Step 4 / 13

VARIABLES

`k = 0` `i = 3` `j = 1` `d[i][j] = 5`

Floyd-Warshall: Step Through the Code

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)        // via
3      for (int i = 0; i < V; i++)      // from
4        for (int j = 0; j < V; j++)    // to
▶          if (d[i][k] != INF
▶              && d[k][j] != INF
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            prev[i][j] = prev[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negative cycle");
15 }
```



d

	0	1	2	3
0	0	3	5	7
1	8	0	2	15
2	5	8	0	1
3	2	5	∞	0

prev

	0	1	2	3
0	/	0	1	0
1	1	/	1	0
2	2	0	/	2
3	3	0	/	/

gray: row and column k = 1

k = 1 as intermediate: d[0][1] + d[1][2] = 3 + 2 = 5 < ∞ → d[0][2] = 5, prev[0][2] = prev[1][2] = 1.

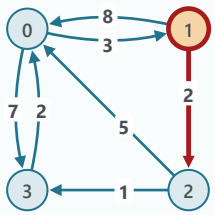
Step 5 / 13

VARIABLES

`k = 1` `i = 0` `j = 2` `d[i][j] = 5`

Floyd-Warshall: Step Through the Code

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)        // via
3      for (int i = 0; i < V; i++)      // from
4        for (int j = 0; j < V; j++)    // to
▶          if (d[i][k] != INF
▶              && d[k][j] != INF
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            prev[i][j] = prev[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negative cycle");
15 }
```



d

	0	1	2	3
0	0	3	5	7
1	8	0	2	15
2	5	8	0	1
3	2	5	7	0

prev

	0	1	2	3
0	/	0	1	0
1	1	/	1	0
2	2	0	/	2
3	3	0	1	/

gray: row and column k = 1

k = 1 as intermediate: d[3][1] + d[1][2] = 5 + 2 = 7 < ∞ → d[3][2] = 7, prev[3][2] = prev[1][2] = 1.

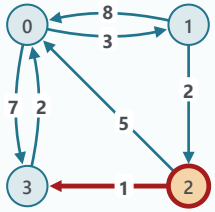
Step 6 / 13

VARIABLES

`k = 1` `i = 3` `j = 2` `d[i][j] = 7`

Floyd-Warshall: Step Through the Code

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)        // via
3      for (int i = 0; i < V; i++)      // from
4        for (int j = 0; j < V; j++)    // to
5          if (d[i][k] != INF
6              && d[k][j] != INF
7              && d[i][k] + d[k][j] < d[i][j]) {
8            d[i][j] = d[i][k] + d[k][j];
9            prev[i][j] = prev[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negative cycle");
15 }
```



d

	0	1	2	3
0	0	3	5	6
1	8	0	2	15
2	5	8	0	1
3	2	5	7	0

prev

	0	1	2	3
0	/	0	1	2
1	1	/	1	0
2	2	0	/	2
3	3	0	1	/

gray: row and column k = 2

k = 2 as intermediate: d[0][2] + d[2][3] = 5 + 1 = 6 < 7 → d[0][3] = 6, prev[0][3] = prev[2][3] = 2.

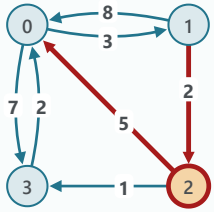
Step 7 / 13

VARIABLES

`k = 2` `i = 0` `j = 3` `d[i][j] = 6`

Floyd-Warshall: Step Through the Code

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)        // via
3      for (int i = 0; i < V; i++)      // from
4        for (int j = 0; j < V; j++)    // to
▶          if (d[i][k] != INF
▶              && d[k][j] != INF
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            prev[i][j] = prev[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negative cycle");
15 }
```



d

	0	1	2	3
0	0	3	5	6
1	7	0	2	15
2	5	8	0	1
3	2	5	7	0

prev

	0	1	2	3
0	/	0	1	2
1	2	/	1	0
2	2	0	/	2
3	3	0	1	/

gray: row and column k = 2

k = 2 as intermediate: d[1][2] + d[2][0] = 2 + 5 = 7 < 8 → d[1][0] = 7, prev[1][0] = prev[2][0] = 2.

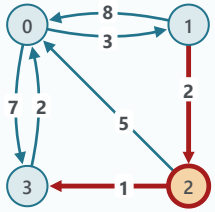
Step 8 / 13

VARIABLES

`k = 2` `i = 1` `j = 0` `d[i][j] = 7`

Floyd-Warshall: Step Through the Code

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)        // via
3      for (int i = 0; i < V; i++)      // from
4        for (int j = 0; j < V; j++)    // to
▶          if (d[i][k] != INF
▶              && d[k][j] != INF
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            prev[i][j] = prev[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negative cycle");
15 }
```



d

	0	1	2	3
0	0	3	5	6
1	7	0	2	3
2	5	8	0	1
3	2	5	7	0

prev

	0	1	2	3
0	/	0	1	2
1	2	/	1	2
2	2	0	/	2
3	3	0	1	/

gray: row and column k = 2

k = 2 as intermediate: d[1][2] + d[2][3] = 2 + 1 = 3 < 15 → d[1][3] = 3, prev[1][3] = prev[2][3] = 2.

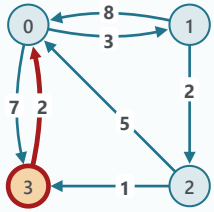
Step 9 / 13

VARIABLES

`k = 2` `i = 1` `j = 3` `d[i][j] = 3`

Floyd-Warshall: Step Through the Code

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)        // via
3      for (int i = 0; i < V; i++)      // from
4        for (int j = 0; j < V; j++)    // to
▶          if (d[i][k] != INF
▶              && d[k][j] != INF
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            prev[i][j] = prev[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negative cycle");
15 }
```



d

	0	1	2	3
0	0	3	5	6
1	5	0	2	3
2	5	8	0	1
3	2	5	7	0

prev

	0	1	2	3
0	/	0	1	2
1	3	/	1	2
2	2	0	/	2
3	3	0	1	/

gray: row and column k = 3

k = 3 as intermediate: d[1][3] + d[3][0] = 3 + 2 = 5 < 7 → d[1][0] = 5, prev[1][0] = prev[3][0] = 3.

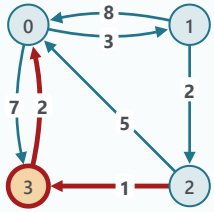
Step 10 / 13

VARIABLES

`k = 3` `i = 1` `j = 0` `d[i][j] = 5`

Floyd-Warshall: Step Through the Code

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)        // via
3      for (int i = 0; i < V; i++)      // from
4        for (int j = 0; j < V; j++)    // to
▶          if (d[i][k] != INF
▶              && d[k][j] != INF
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            prev[i][j] = prev[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negative cycle");
15 }
```



d

	0	1	2	3
0	0	3	5	6
1	5	0	2	3
2	3	8	0	1
3	2	5	7	0

prev

	0	1	2	3
0	/	0	1	2
1	3	/	1	2
2	3	0	/	2
3	3	0	1	/

gray: row and column k = 3

k = 3 as intermediate: d[2][3] + d[3][0] = 1 + 2 = 3 < 5 → d[2][0] = 3, prev[2][0] = prev[3][0] = 3.

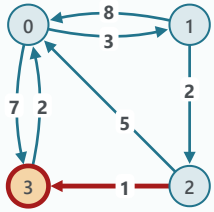
Step 11 / 13

VARIABLES

`k = 3` `i = 2` `j = 0` `d[i][j] = 3`

Floyd-Warshall: Step Through the Code

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)        // via
3      for (int i = 0; i < V; i++)      // from
4        for (int j = 0; j < V; j++)    // to
▶          if (d[i][k] != INF
▶              && d[k][j] != INF
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            prev[i][j] = prev[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negative cycle");
15 }
```



d

	0	1	2	3
0	0	3	5	6
1	5	0	2	3
2	3	6	0	1
3	2	5	7	0

prev

	0	1	2	3
0	/	0	1	2
1	3	/	1	2
2	3	0	/	2
3	3	0	1	/

gray: row and column k = 3

k = 3 as intermediate: d[2][3] + d[3][1] = 1 + 5 = 6 < 8 → d[2][1] = 6, prev[2][1] = prev[3][1] = 0.

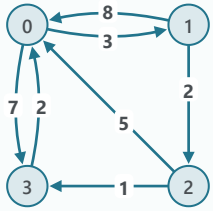
Step 12 / 13

VARIABLES

`k = 3` `i = 2` `j = 1` `d[i][j] = 6`

Floyd-Warshall: Step Through the Code

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)        // via
3      for (int i = 0; i < V; i++)      // from
4        for (int j = 0; j < V; j++)    // to
5          if (d[i][k] != INF
6              && d[k][j] != INF
7              && d[i][k] + d[k][j] < d[i][j]) {
8            d[i][j] = d[i][k] + d[k][j];
9            prev[i][j] = prev[k][j];
10         }
▶   for (int i = 0; i < V; i++)
▶     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negative cycle");
15 }
```



d

	0	1	2	3
0	0	3	5	6
1	5	0	2	3
2	3	6	0	1
3	2	5	7	0

prev

	0	1	2	3
0	/	0	1	2
1	3	/	1	2
2	3	0	/	2
3	3	0	1	/

All four passes are done. No negative value on the diagonal → no negative cycle. Sample path: prev[1][3] = 2, prev[1][2] = 1 → 1 → 2 → 3, d[1][3] = 3.

Step 13 / 13

VARIABLES

k = – i = – j = –

The A* (A-Star) Search Algorithm

- A **search** algorithm that finds a shortest path between two points (start → goal).
- Developed in 1968 by Peter **Hart**, Nils **Nilsson** and Bertram **Raphael**.
- Combines Dijkstra's actual cost (uniform-cost search) with the goal estimate of **greedy best-first search**.
- It needs a good **estimate function** (a heuristic) to work well.
- Time and memory grow quickly as the number of vertices increases.

How it works

Each vertex has an **estimate** h(n) of its distance to the goal. At every step, among the vertices in the **open list**, it picks the one with the smallest sum of actual cost and estimated cost. This way it heads toward the goal while still keeping the cheapest path.

How Does A* Work? f(n) = g(n) + h(n)

g(n): actual cost
The cost of the path **found so far** from the start to n (the "g-cost"). Same as d[n] in Dijkstra.

h(n): heuristic
The **estimated** cost from n to the goal (the "h-cost"), such as straight-line or Manhattan distance.

f(n) = g(n) + h(n)
The estimated total cost of a solution through n. The vertex with the smallest f in the open list is expanded.

Selection rule	Algorithm	Behavior
g(n) only	**Dijkstra**	spreads equally in all directions; correct but expands many vertices
h(n) only	**Greedy best-first**	rushes toward the goal; fast, but no shortest-path guarantee
g(n) + h(n)	**A***	heads toward the goal; with a suitable h, guarantees a shortest path

Vertices that **look close** to the goal by the heuristic get priority, but thanks to g a vertex reached by an expensive path is held back.

A*: The Steps

1. Set g(start) = 0 and add it to the **open list** (a priority queue). All other vertices get g = ∞.

Open list: discovered but not yet expanded vertices.

A*: The Steps

1. Set g(start) = 0 and add it to the **open list** (a priority queue). All other vertices get g = ∞.

Open list: discovered but not yet expanded vertices.

2. Remove the vertex with the smallest **f = g + h** from the open list; put it on the **closed list**.

The rule is the sum f = g + h, not g or h alone.

A*: The Steps

1. Set g(start) = 0 and add it to the **open list** (a priority queue). All other vertices get g = ∞.

Open list: discovered but not yet expanded vertices.

2. Remove the vertex with the smallest **f = g + h** from the open list; put it on the **closed list**.

The rule is the sum f = g + h, not g or h alone.

3. If it is the **goal**, stop: rebuild the path from the predecessors.

The goal is final when removed, not when added.

A*: The Steps

1. Set g(start) = 0 and add it to the **open list** (a priority queue). All other vertices get g = ∞.

Open list: discovered but not yet expanded vertices.

2. Remove the vertex with the smallest **f = g + h** from the open list; put it on the **closed list**.

The rule is the sum f = g + h, not g or h alone.

3. If it is the **goal**, stop: rebuild the path from the predecessors.

The goal is final when removed, not when added.

4. Otherwise, for each neighbor v let g' = g(u) + w(u, v); if g' is smaller, **update** g, f and the predecessor and add v to the open list.

Dijkstra's relaxation. A closed v with a better g is reopened (never with a consistent h).

A*: The Steps

1. Set g(start) = 0 and add it to the **open list** (a priority queue). All other vertices get g = ∞.

Open list: discovered but not yet expanded vertices.

2. Remove the vertex with the smallest **f = g + h** from the open list; put it on the **closed list**.

The rule is the sum f = g + h, not g or h alone.

3. If it is the **goal**, stop: rebuild the path from the predecessors.

The goal is final when removed, not when added.

4. Otherwise, for each neighbor v let g' = g(u) + w(u, v); if g' is smaller, **update** g, f and the predecessor and add v to the open list.

Dijkstra's relaxation. A closed v with a better g is reopened (never with a consistent h).

5. Go back to step 2. If the open list empties, there is **no path**.

The choice is from the whole open list, not just the last vertex's neighbors.

Properties of a Heuristic

Admissible

$h(n) \leq h^*(n)$: it **never overestimates** the true remaining cost. Then, when A* removes the goal, the path it found is shortest.

Consistent

$h(u) \leq w(u, v) + h(v)$ and $h(\text{goal}) = 0$. With a consistent h, a vertex is final once it enters the closed list (it is never reopened).

Grid heuristics

4 directions: Manhattan $|\Delta x| + |\Delta y|$. 8 directions: octile $14 \cdot \min + 10 \cdot (\max - \min)$. Free plane: Euclidean distance.

Extreme cases

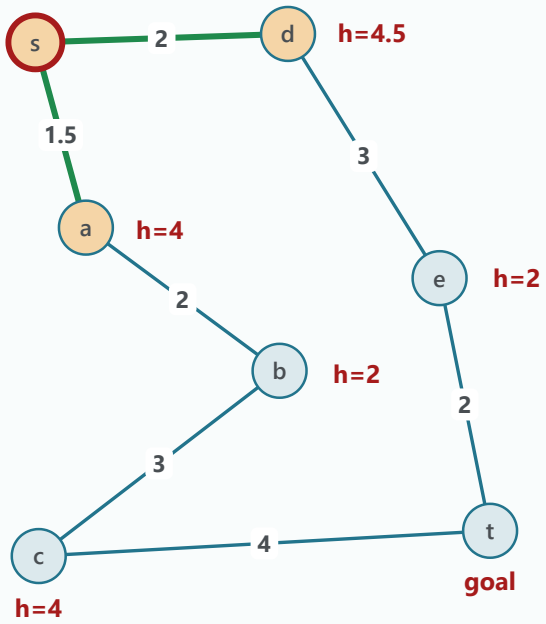
$h = 0 \rightarrow$ A* = Dijkstra. $h = h^*$ (perfect) → only vertices on a shortest path are expanded. $h > h^*$ → fast, but no shortest-path guarantee.

A*: Running Time

- Worst-case time and memory in terms of the search tree **$O(b^d)$**:
  - **b**: branching factor (average number of neighbors), **d**: depth of the goal (number of steps in the solution).
- On a finite graph with a consistent h, A* does at most **$O((V + E) \log V)$** work, like Dijkstra.
- The better the heuristic, the fewer vertices are expanded: with a **perfect** estimate ($h = h^*$) only the d vertices on the path are expanded.
- Its weak point is **memory**: the open and closed lists grow; variants such as IDA* and SMA* bound the memory.

$O(b^d)$ is the bound for the **worst** case (a useless h, e.g. $h = 0$), not for a perfect estimate; with a perfect estimate the search is linear in the path length.

A* Example: s to t (h values next to the vertices)



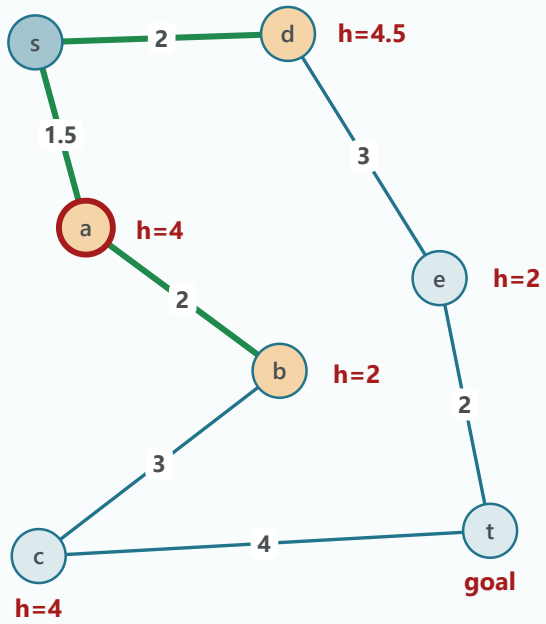
Open list

vertex	g	h	f
a	1.5	4	**5.5**
d	2	4.5	6.5

Closed: s

Expand s: neighbors a (g = 1.5) and d (g = 2) are opened. f(a) = 1.5 + 4 = **5.5**, f(d) = 2 + 4.5 = 6.5.

A* Example: s to t (h values next to the vertices)



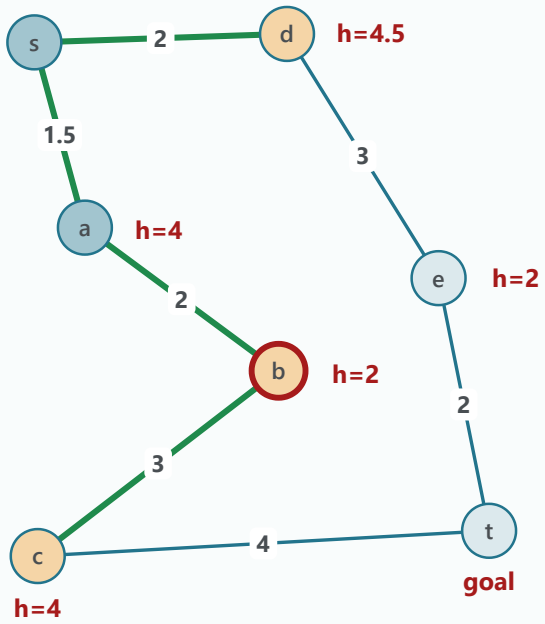
Open list

vertex	g	h	f
b	3.5	2	**5.5**
d	2	4.5	6.5

Closed: s, a

Smallest f: a (5.5). Expand a → b: g = 3.5, f(b) = 3.5 + 2 = **5.5**.

A* Example: s to t (h values next to the vertices)



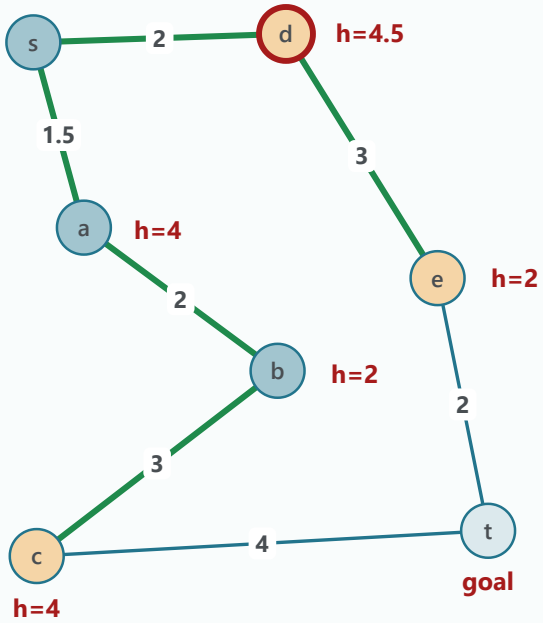
Open list

vertex	g	h	f
d	2	4.5	**6.5**
c	6.5	4	10.5

Closed: s, a, b

Expand b (5.5 < 6.5) → c: g = 6.5, f(c) = 6.5 + 4 = 10.5. This path is getting expensive.

A* Example: s to t (h values next to the vertices)



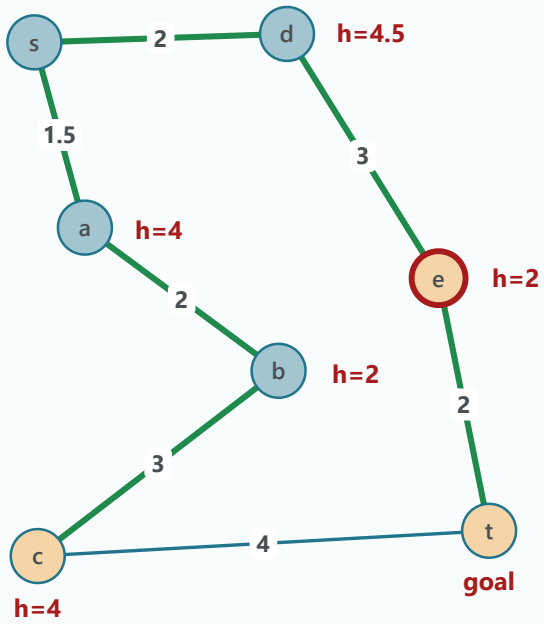
Open list

vertex	g	h	f
e	5	2	**7**
c	6.5	4	10.5

Closed: s, a, b, d

Now the smallest f is d (6.5). The search switches branches → e: g = 5, f(e) = 5 + 2 = **7**.

A* Example: s to t (h values next to the vertices)



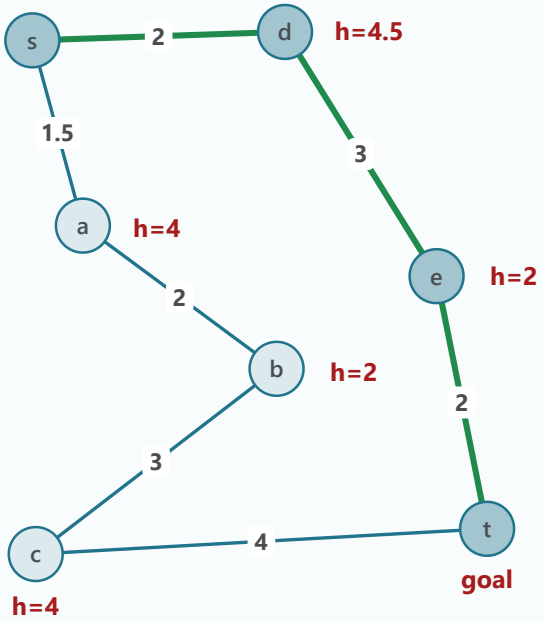
Open list

vertex	g	h	f
t	7	0	**7**
c	6.5	4	10.5

Closed: s, a, b, d, e

Expand e → t: g = 5 + 2 = 7, f(t) = 7 + 0 = **7** < f(c) = 10.5.

A* Example: s to t (h values next to the vertices)



Open list

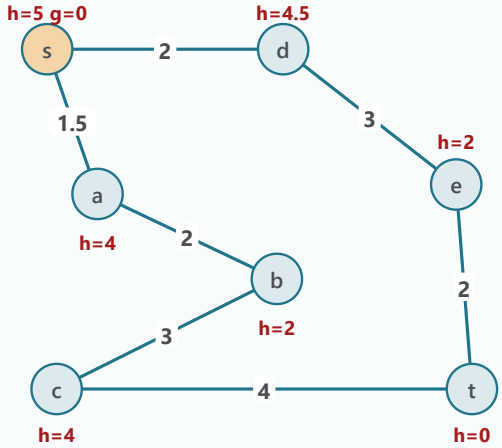
vertex	g	h	f
c	6.5	4	**10.5**

Closed: s, a, b, d, e, t

t is removed from the open list: the **goal**. Shortest path s → d → e → t, cost 7. c was never expanded.

A*: Step Through the Code

```
1  double aStar(int s, int t) {
▶    Arrays.fill(g, INF); g[s] = 0;
▶    Queue<Integer> open = new PriorityQueue<>(
▶      Comparator.comparingDouble(v -> g[v] + h[v]));
▶    open.add(s);
6    while (!open.isEmpty()) {
7      int u = open.poll();       // smallest f
8      if (u == t) return g[t];   // goal
9      for (Edge e : adj[u])
10       if (g[u] + e.w < g[e.v]) {
11         g[e.v] = g[u] + e.w; parent[e.v] = u;
12         open.remove(e.v); open.add(e.v);
13       }
14   }
15   return INF;
16 }
```



Undirected graph, goal t; h beside each vertex estimates the distance to t. g(s) = 0; open = {s} (f in parentheses).

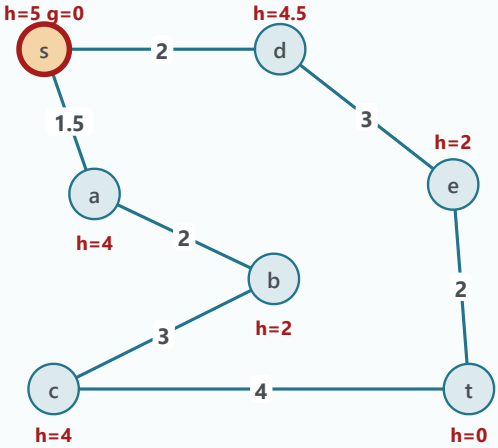
Step 1 / 12

VARIABLES

`u = –` `g(u) = –` `open = [s(5)]`

A*: Step Through the Code

```
1  double aStar(int s, int t) {
2    Arrays.fill(g, INF); g[s] = 0;
3    Queue<Integer> open = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    open.add(s);
6    while (!open.isEmpty()) {
▶      int u = open.poll();       // smallest f
▶      if (u == t) return g[t];   // goal
9      for (Edge e : adj[u])
10       if (g[u] + e.w < g[e.v]) {
11         g[e.v] = g[u] + e.w; parent[e.v] = u;
12         open.remove(e.v); open.add(e.v);
13       }
14   }
15   return INF;
16 }
```



Remove s, the smallest f (0 + 5 = 5). Not the goal; examine its neighbors.

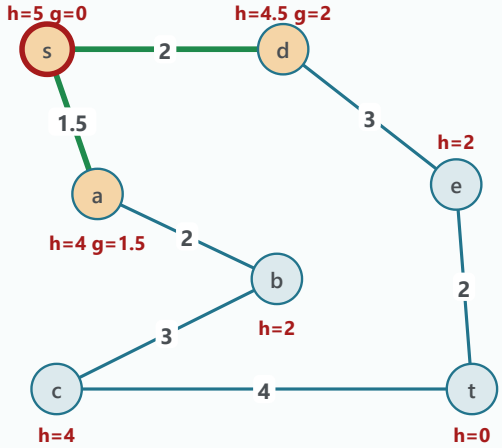
Step 2 / 12

VARIABLES

`u = s` `g(u) = 0` `open = []`

A*: Step Through the Code

```
1  double aStar(int s, int t) {
2    Arrays.fill(g, INF); g[s] = 0;
3    Queue<Integer> open = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    open.add(s);
6    while (!open.isEmpty()) {
7      int u = open.poll();       // smallest f
8      if (u == t) return g[t];   // goal
▶      for (Edge e : adj[u])
▶        if (g[u] + e.w < g[e.v]) {
▶          g[e.v] = g[u] + e.w; parent[e.v] = u;
▶          open.remove(e.v); open.add(e.v);
13       }
14   }
15   return INF;
16 }
```



g(a) = 0 + 1.5 = 1.5, f = 5.5; g(d) = 0 + 2 = 2, f = 6.5.

Step 3 / 12

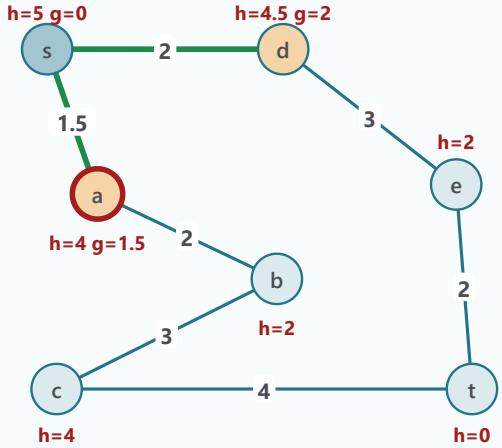
VARIABLES

`u = s` `g(u) = 0`

`open = [a(5.5), d(6.5)]`

A*: Step Through the Code

```
1  double aStar(int s, int t) {
2    Arrays.fill(g, INF); g[s] = 0;
3    Queue<Integer> open = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    open.add(s);
6    while (!open.isEmpty()) {
▶      int u = open.poll();       // smallest f
▶      if (u == t) return g[t];   // goal
9      for (Edge e : adj[u])
10       if (g[u] + e.w < g[e.v]) {
11         g[e.v] = g[u] + e.w; parent[e.v] = u;
12         open.remove(e.v); open.add(e.v);
13       }
14   }
15   return INF;
16 }
```



Remove a, the smallest f (1.5 + 4 = 5.5). Not the goal; examine its neighbors.

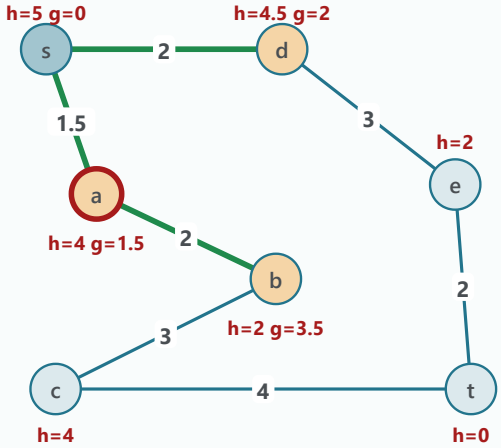
Step 4 / 12

VARIABLES

`u = a` `g(u) = 1.5` `open = [d(6.5)]`

A*: Step Through the Code

```
1  double aStar(int s, int t) {
2    Arrays.fill(g, INF); g[s] = 0;
3    Queue<Integer> open = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    open.add(s);
6    while (!open.isEmpty()) {
7      int u = open.poll();      // smallest f
8      if (u == t) return g[t];  // goal
▶      for (Edge e : adj[u])
▶        if (g[u] + e.w < g[e.v]) {
▶          g[e.v] = g[u] + e.w; parent[e.v] = u;
▶          open.remove(e.v); open.add(e.v);
13       }
14   }
15   return INF;
16 }
```



g(b) = 1.5 + 2 = 3.5, f = 5.5.

Step 5 / 12

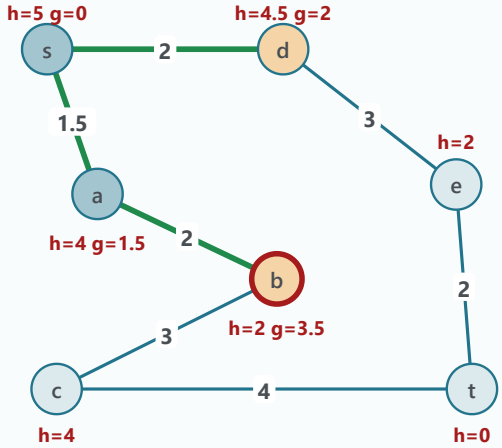
VARIABLES

`u = a` `g(u) = 1.5`

`open = [b(5.5), d(6.5)]`

A*: Step Through the Code

```
1  double aStar(int s, int t) {
2    Arrays.fill(g, INF); g[s] = 0;
3    Queue<Integer> open = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    open.add(s);
6    while (!open.isEmpty()) {
▶      int u = open.poll();       // smallest f
▶      if (u == t) return g[t];   // goal
9      for (Edge e : adj[u])
10       if (g[u] + e.w < g[e.v]) {
11         g[e.v] = g[u] + e.w; parent[e.v] = u;
12         open.remove(e.v); open.add(e.v);
13       }
14   }
15   return INF;
16 }
```



Remove b, the smallest f (3.5 + 2 = 5.5). Not the goal; examine its neighbors.

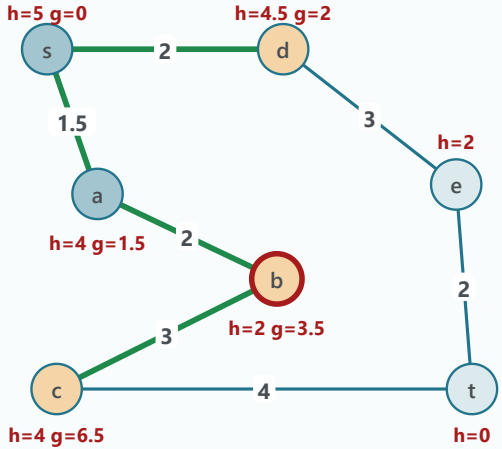
Step 6 / 12

VARIABLES

`u = b` `g(u) = 3.5` `open = [d(6.5)]`

A*: Step Through the Code

```
1  double aStar(int s, int t) {
2    Arrays.fill(g, INF); g[s] = 0;
3    Queue<Integer> open = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    open.add(s);
6    while (!open.isEmpty()) {
7      int u = open.poll();       // smallest f
8      if (u == t) return g[t];   // goal
▶      for (Edge e : adj[u])
▶        if (g[u] + e.w < g[e.v]) {
▶          g[e.v] = g[u] + e.w; parent[e.v] = u;
▶          open.remove(e.v); open.add(e.v);
13       }
14   }
15   return INF;
16 }
```



g(c) = 3.5 + 3 = 6.5, f = 10.5. Too large: d (f = 6.5) is next.

Step 7 / 12

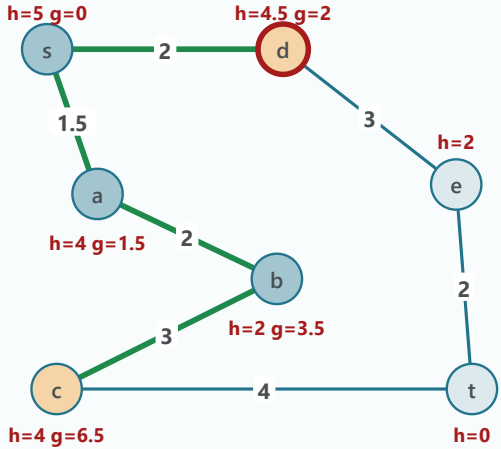
VARIABLES

`u = b` `g(u) = 3.5`

`open = [d(6.5), c(10.5)]`

A*: Step Through the Code

```
1  double aStar(int s, int t) {
2    Arrays.fill(g, INF); g[s] = 0;
3    Queue<Integer> open = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    open.add(s);
6    while (!open.isEmpty()) {
▶      int u = open.poll();       // smallest f
▶      if (u == t) return g[t];   // goal
9      for (Edge e : adj[u])
10       if (g[u] + e.w < g[e.v]) {
11         g[e.v] = g[u] + e.w; parent[e.v] = u;
12         open.remove(e.v); open.add(e.v);
13       }
14   }
15   return INF;
16 }
```



Remove d, the smallest f (2 + 4.5 = 6.5). Not the goal; examine its neighbors.

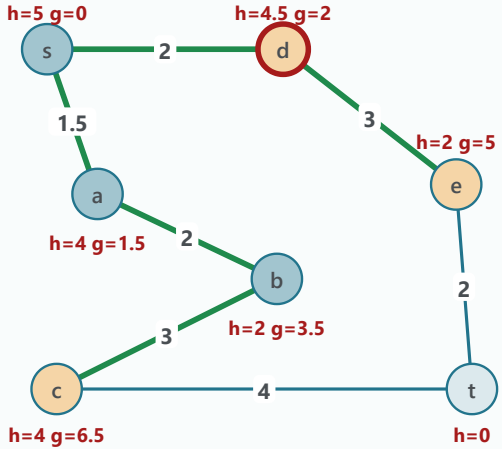
Step 8 / 12

VARIABLES

`u = d` `g(u) = 2` `open = [c(10.5)]`

A*: Step Through the Code

```
1  double aStar(int s, int t) {
2    Arrays.fill(g, INF); g[s] = 0;
3    Queue<Integer> open = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    open.add(s);
6    while (!open.isEmpty()) {
7      int u = open.poll();      // smallest f
8      if (u == t) return g[t];  // goal
▶      for (Edge e : adj[u])
▶        if (g[u] + e.w < g[e.v]) {
▶          g[e.v] = g[u] + e.w; parent[e.v] = u;
▶          open.remove(e.v); open.add(e.v);
13       }
14   }
15   return INF;
16 }
```



g(e) = 2 + 3 = 5, f = 7.

Step 9 / 12

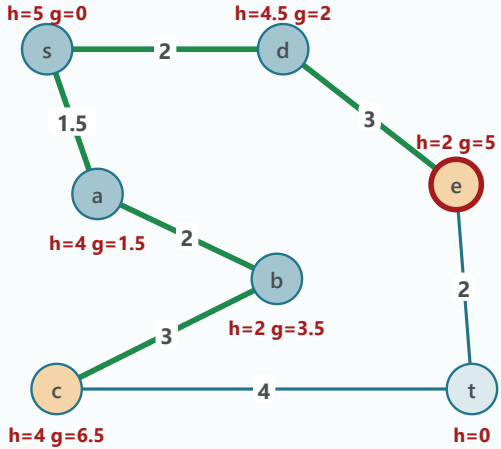
VARIABLES

`u = d` `g(u) = 2`

`open = [e(7), c(10.5)]`

A*: Step Through the Code

```
1  double aStar(int s, int t) {
2    Arrays.fill(g, INF); g[s] = 0;
3    Queue<Integer> open = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    open.add(s);
6    while (!open.isEmpty()) {
▶      int u = open.poll();       // smallest f
▶      if (u == t) return g[t];   // goal
9      for (Edge e : adj[u])
10       if (g[u] + e.w < g[e.v]) {
11         g[e.v] = g[u] + e.w; parent[e.v] = u;
12         open.remove(e.v); open.add(e.v);
13       }
14   }
15   return INF;
16 }
```



Remove e, the smallest f (5 + 2 = 7). Not the goal; examine its neighbors.

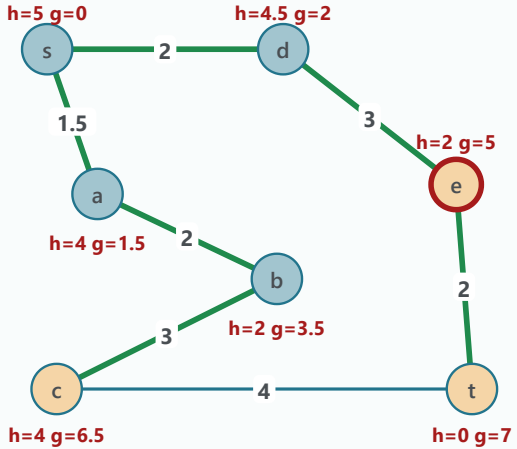
Step 10 / 12

VARIABLES

`u = e` `g(u) = 5` `open = [c(10.5)]`

A*: Step Through the Code

```
1  double aStar(int s, int t) {
2    Arrays.fill(g, INF); g[s] = 0;
3    Queue<Integer> open = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    open.add(s);
6    while (!open.isEmpty()) {
7      int u = open.poll();      // smallest f
8      if (u == t) return g[t];  // goal
▶      for (Edge e : adj[u])
▶        if (g[u] + e.w < g[e.v]) {
▶          g[e.v] = g[u] + e.w; parent[e.v] = u;
▶          open.remove(e.v); open.add(e.v);
13       }
14   }
15   return INF;
16 }
```



g(t) = 5 + 2 = 7, f = 7.

Step 11 / 12

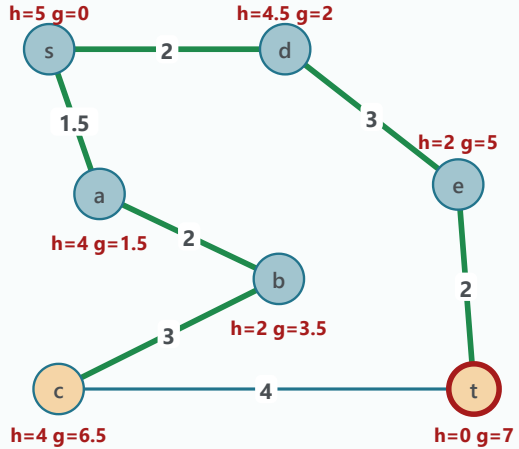
VARIABLES

`u = e` `g(u) = 5`

`open = [t(7), c(10.5)]`

A*: Step Through the Code

```
1  double aStar(int s, int t) {
2    Arrays.fill(g, INF); g[s] = 0;
3    Queue<Integer> open = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    open.add(s);
6    while (!open.isEmpty()) {
▶      int u = open.poll();       // smallest f
▶      if (u == t) return g[t];   // goal
9      for (Edge e : adj[u])
10       if (g[u] + e.w < g[e.v]) {
11         g[e.v] = g[u] + e.w; parent[e.v] = u;
12         open.remove(e.v); open.add(e.v);
13       }
14   }
15   return INF;
16 }
```



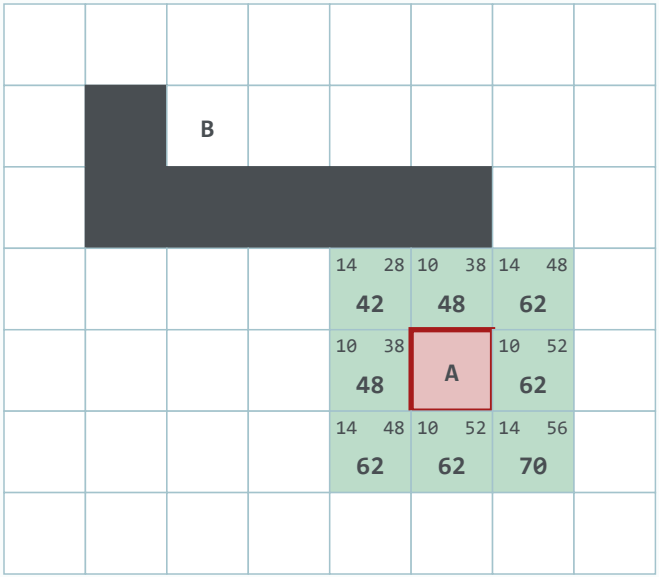
Smallest f: t (7), the goal → `return g[t]` = **7**. Path s → d → e → t; c never expanded.

Step 12 / 12

VARIABLES

`u = t` `g(u) = 7` `open = [c(10.5)]`

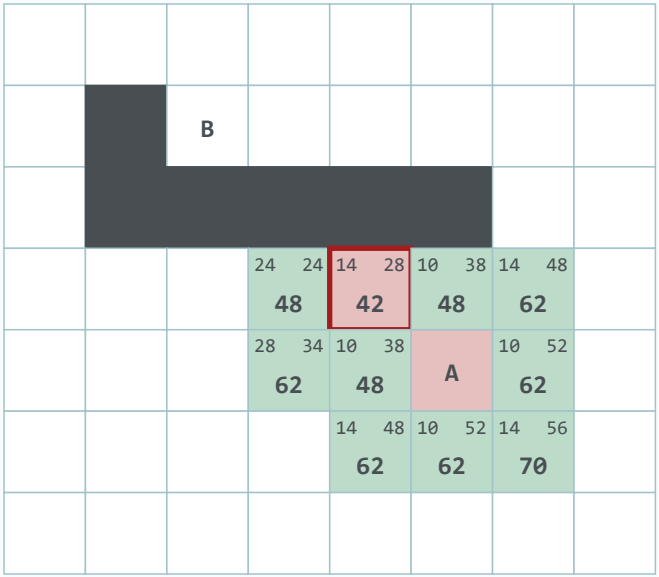
A* Example: Around a Wall on a Grid



open closed path wall top left g · top right h · center f

Expand A: 8 neighbors are opened. A straight step costs 10, a diagonal 14; h = octile distance (walls ignored). Smallest f = 42.

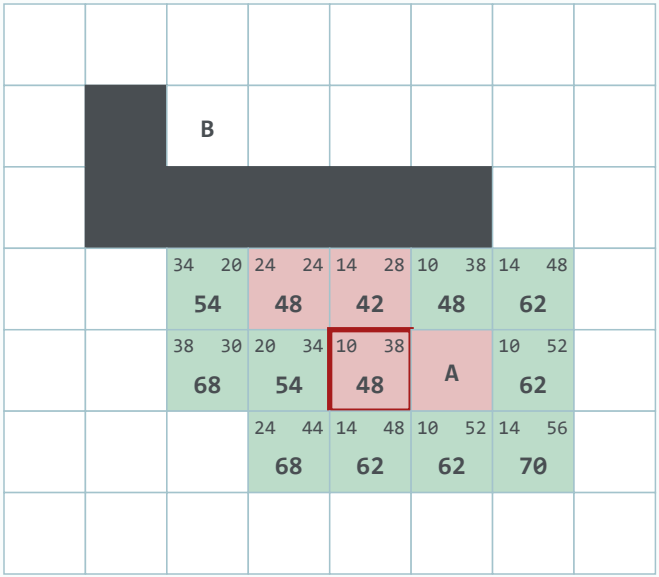
A* Example: Around a Wall on a Grid



open closed path wall top left g · top right h · center f

Expansion 2: the cell with the smallest f = 42 (orange frame) moves to the closed list; its neighbors are opened or updated.

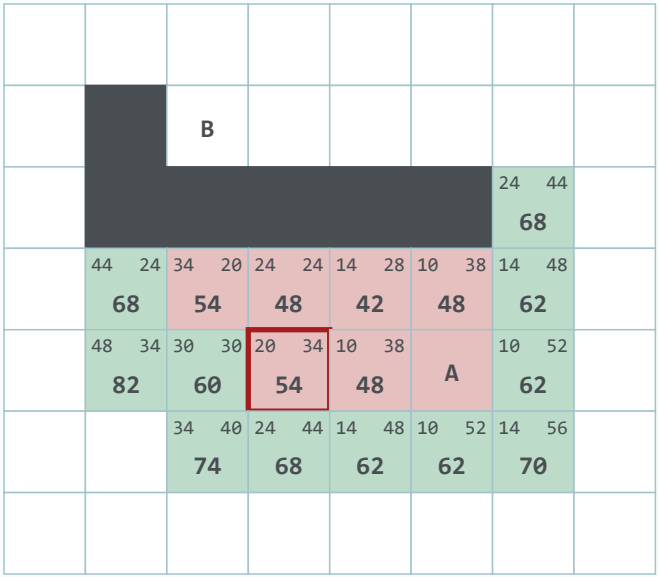
A* Example: Around a Wall on a Grid



open closed path wall top left g · top right h · center f

Expansion 4: the cell with the smallest f = 48 (orange frame) moves to the closed list; its neighbors are opened or updated.

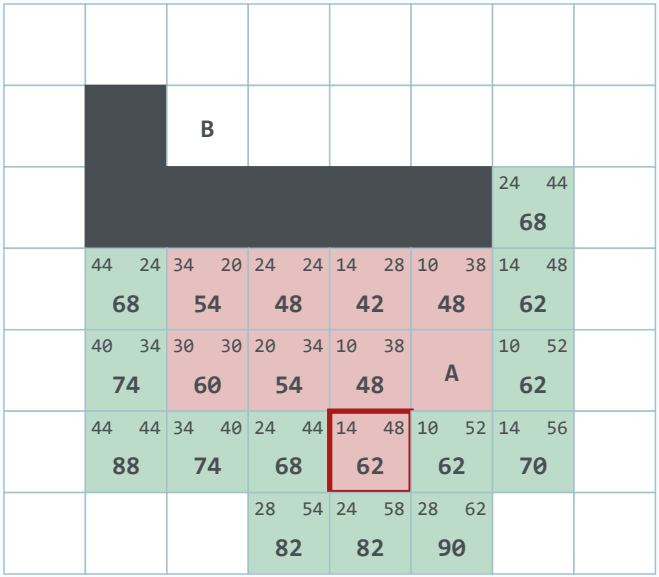
A* Example: Around a Wall on a Grid



open closed path wall top left g · top right h · center f

Expansion 7: the cell with the smallest f = 54 (orange frame) moves to the closed list; its neighbors are opened or updated. Cells below the wall, with f = 42…62, are tried first.

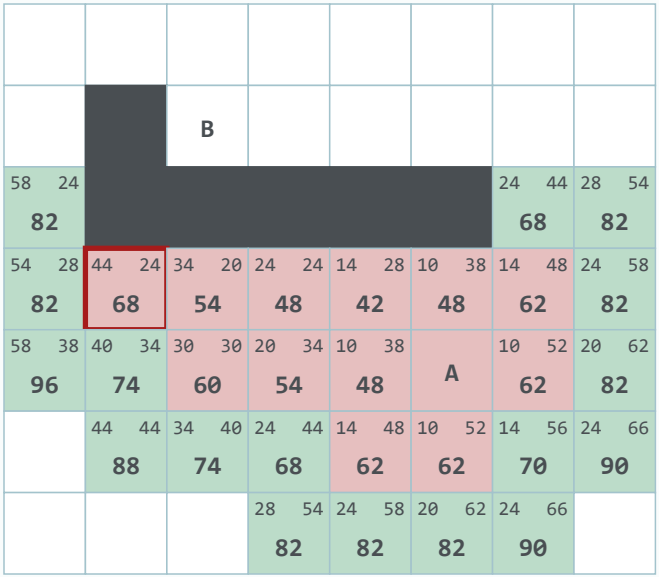
A* Example: Around a Wall on a Grid



open closed path wall top left g · top right h · center f

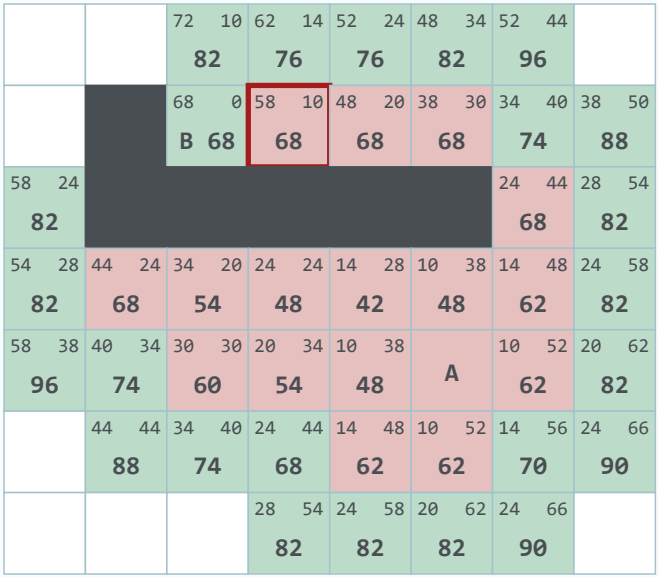
Expansion 9: the cell with the smallest f = 62 (orange frame) moves to the closed list; its neighbors are opened or updated.

A* Example: Around a Wall on a Grid



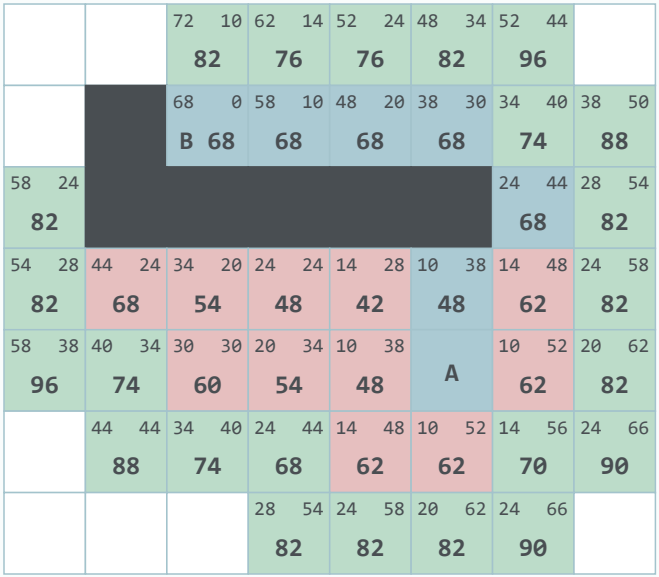
Expansion 13: the cell with the smallest f = 68 (orange frame) moves to the closed list; its neighbors are opened or updated. The lower route is blocked: f has risen to 68, and the search spreads to both sides of the wall.

A* Example: Around a Wall on a Grid



Expansion 17 (f = 68): the path around the right of the wall has reached B; B is in the open list with f = 68.

A* Example: Around a Wall on a Grid

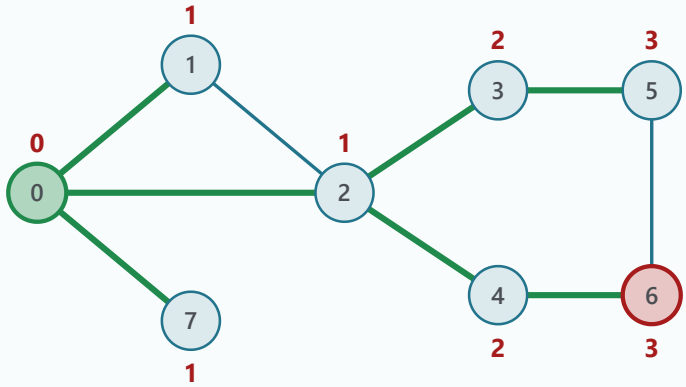


open closed path wall top left g · top right h · center f

B is removed: the path goes up from A → diagonal up-right → diagonal up-left → three steps left over the wall, cost **68** (2 diagonals × 14 + 4 straight × 10). 18 cells were expanded.

BFS: Shortest Paths in an Unweighted Graph

- When all edges are equal (weight 1), the shortest path = the path with the fewest edges.
- **Breadth-first search** (BFS) visits vertices level by level, in order of distance from the source: d[v] = d[u] + 1.
- A FIFO queue replaces Dijkstra's priority queue: **O(V + E)**.
- Example (graph at right): the shortest path 0 → 6 is **0** → **2** → **4** → **6** (3 edges).
- **Bidirectional search**: run BFS from both the source and the target and stop when the two waves meet; about $2 \cdot b^{d/2}$ vertices are expanded.



Numbers: BFS level (edge count). Green: the BFS tree.

COMPARISON

Comparing Shortest-Path Algorithms

Algorithm	Problem	Negative edges	Negative cycle	Time	Approach
BFS	single source, unweighted	—	—	$O(V + E)$	level by level
Dijkstra	single source	no	no	$O(V^2)$ / $O((V + E) \log V)$	greedy
Bellman-Ford	single source	yes	detects	$O(V \cdot E)$	repeated relaxation (DP)
Floyd-Warshall	all pairs	yes	detects	$\Theta(V^3)$	dynamic programming
A*	single pair	no	no	worst case $O(b^d)$	heuristic + greedy

Unweighted?
BFS is enough.

Weights ≥ 0?
Dijkstra; A* if there is a single target and a good h.

Negative edges?
Bellman-Ford.

All pairs, small V?
Floyd-Warshall.

Summary

Topic	Key points
Relaxation	if d[u] + w < d[v], update d[v] and π[v]; the operation all the algorithms share
Dijkstra	settle the vertex with the smallest d, relax its neighbors; weights ≥ 0; O((V + E) log V)
Bellman-Ford	relax every edge for V − 1 passes; an edge that relaxes in pass V = a negative cycle reachable from the source; O(V · E)
Floyd-Warshall	D[i][j] = min(D[i][j], D[i][k] + D[k][j]), k outermost; D[i][i] < 0 → negative cycle; Θ(V^3)
A*	f = g + h; shortest path with an admissible h; h = 0 → Dijkstra
BFS	fewest-edge path in an unweighted graph; O(V + E)
Paths	follow the π (Π) pointers back from the target; shortest-path tree

SUMMARY

Common Mistakes

Dijkstra with negative edges

A shorter path to a settled vertex can appear later; the result is silently wrong. Negative edges need Bellman-Ford.

∞ + w overflow

`Integer.MAX_VALUE + w` wraps around to a negative number and looks "shorter". That is why `d[u] != INF` is checked first.

Floyd-Warshall loop order

The intermediate-vertex loop k must be **outermost**; the order i, j, k gives wrong results.

Number of passes

Bellman-Ford may need all V − 1 passes (Example 2); one pass is not always enough. The check pass is separate.

Stopping at the goal in A*

Stop when the goal is **removed** from the open list, not when it is **added**. An h that overestimates can miss the shortest path.

Negative weight on an undirected edge

An undirected negative edge is a negative cycle by itself: u → v → u.

SUMMARY

Worked Problems

Directed graph: A→B 4, A→C 2, C→B 1, B→D 5, C→D 8, C→E 10, D→E 3; source A.

1. In what order does Dijkstra settle the vertices? What are the final d values?
2. Write the shortest path from A to E using the π array.
3. If the edges are examined in alphabetical order (A→B, A→C, B→D, C→B, C→D, C→E, D→E), how many relaxations happen in each pass of Bellman-Ford?
4. What is the final Floyd-Warshall D matrix for the graph 1→2 (4), 2→3 (1), 1→3 (7), 3→1 (−2)? Is there a negative cycle?
5. V = 1000, E = 5000, and there are negative edges. Which algorithm, and roughly how many operations?

▶ **Show answers**

Exercises

1. Find all distances in the graph of Dijkstra Example 1 with source **5**.
2. `dijkstra(src, target)` : write a version that stops when the target leaves the queue.
3. Extend Dijkstra to also count the **number of shortest paths** to each vertex.
4. Add early exit to Bellman-Ford; what is the best-case running time?
5. Adapt Floyd-Warshall to compute the **transitive closure** (Warshall) with a boolean matrix.
6. Why is Manhattan distance admissible on a 4-direction grid but not on an 8-direction grid?

▶ **Answers / hints**

References

- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., 2022 (Chapters 22–23).
- Sedgewick and Wayne, *Algorithms*, 4th ed., 2011 (Section 4.4: Shortest Paths).
- E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs", *Numerische Mathematik* 1, 1959.
- R. Bellman, "On a Routing Problem", *Quarterly of Applied Mathematics* 16, 1958.
- R. W. Floyd, "Algorithm 97: Shortest Path", *Communications of the ACM* 5(6), 1962; S. Warshall, "A Theorem on Boolean Matrices", *Journal of the ACM* 9(1), 1962.
- P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Trans. SSC* 4(2), 1968.

The End

Chapter 6: Shortest Paths

Previous: Chapter 5: Graph Traversal · Next: Chapter 7: Minimum Spanning Trees