Data Structures

Shortest paths

The notes for the lesson “Shortest paths”: chapters 5.7 to 5.8, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

2 chapters | Chapters 5.7 to 5.8

About these notes

These are the notes for the University free n easy video lesson **Shortest paths** on Data Structures: chapters 5.7 to 5.8, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

5.7 Single-source shortest paths: Dijkstra's algorithm

Google Maps from your hostel to Chennai Central, the cheapest IRCTC connection, the fastest route for a packet through routers: all are "shortest path from one source to everywhere" on a weighted graph. BFS (5.3) solves it only when every edge costs 1. Dijkstra's algorithm solves it for any non-negative weights by settling vertices in increasing order of distance, always taking the closest unsettled vertex and relaxing its outgoing edges. It is the most examined algorithm of this unit; tutorials often ask explicitly for the distance table after every step, and some syllabi ask for the priority-queue implementation and for Bellman-Ford, which handles negative weights.

> You need: Prim's structure from 5.6; binary heaps from Unit 4.

The problem and the relaxation step

Given a directed (or undirected) graph with edge weights $w(u, v) \geq 0$ and a source s, find for every vertex v the length $\delta(s, v)$ of the lightest path from s, and the path itself. Keep two arrays: dist[v], the best path length found so far (∞ until a path is known, dist[s] = 0), and parent[v], the previous vertex on that path.

To **relax** an edge (u, v) is to ask whether going through u improves v: if dist[u] + w(u, v) < dist[v], set dist[v] = dist[u] + w(u, v) and parent[v] = u. Every shortest-path algorithm is a strategy for the

order in which to relax edges. Shortest paths have the **optimal substructure** property: any prefix of a shortest path is itself a shortest path, so once dist[u] is right, relaxing u's edges gives correct candidates for its neighbours.

Dijkstra's algorithm

1. dist[s] = 0, dist[v] = ∞ for all other v; all vertices unsettled.
2. Repeat V times: choose the unsettled vertex u with the smallest dist (a min-priority queue keyed on dist); mark u settled. Its dist is now final.
3. For each edge (u, v) with v unsettled, relax it; if dist[v] decreased, decrease its key in the priority queue (or simply push the new (dist, v) pair and ignore stale pairs when they surface).

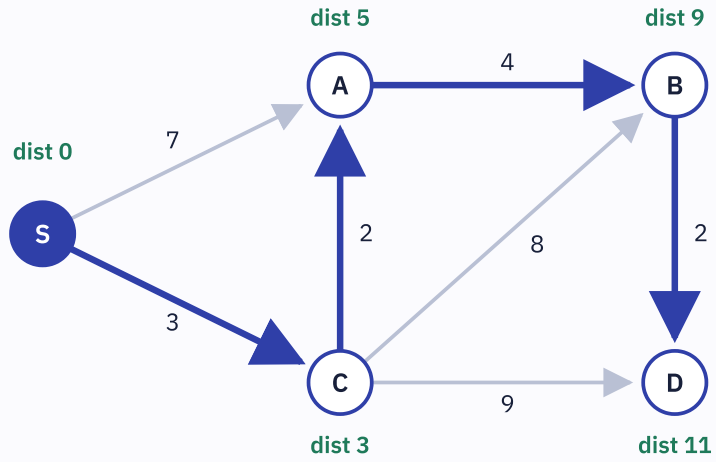
Why is the chosen u final? Every unsettled vertex has dist ≥ dist[u], and any path to u that leaves the settled set must pass through some unsettled vertex x first, costing at least dist[x] ≥ dist[u] already, and more after that because weights are non-negative. So no cheaper path to u exists. The argument breaks with a negative edge, which is Dijkstra's one limitation.

Cost: with an array scan for the minimum, $O(V^2)$ (good for dense graphs); with a binary heap, $O((V + E) \log V)$, usually written $O(E \log V)$; with a Fibonacci heap, $O(E + V \log V)$. The parent pointers form the **shortest-path tree**, and the path to any v is read backwards: v, parent[v], parent[parent[v]], … , s.

Compare Prim's algorithm: identical loop, but Prim's key is the single edge weight w(u, v) while Dijkstra's is the path length dist[u] + w(u, v). Mixing the two rules is the classic exam error.

Negative weights: Bellman-Ford

If weights can be negative (some syllabi list this), use the Bellman-Ford algorithm: relax *every* edge, and repeat that full pass V − 1 times. After pass i all shortest paths of at most i edges are correct, and no shortest path has more than V − 1 edges, so V − 1 passes suffice: O(VE). A V-th pass that still improves something proves a negative-weight cycle, in which case shortest paths are undefined. Bellman-Ford is slower than Dijkstra but needs no priority queue and is the basis of the distance-vector routing protocol RIP.



Settle order and distances

S = 0

C = 3 (S→C)

A = 5 (3 + 2, better than 7)

B = 9 (5 + 4, better than 11)

D = 11 (9 + 2, better than 12)

thick arrows = shortest-path tree

Figure 5.7.1 A directed graph with source S. Green numbers are the final distances; thick arrows are the shortest-path tree from the parent pointers. The direct edge S→A (7) loses to the two-step route S→C→A (3 + 2 = 5), and that improvement propagates to B and D. The order in which vertices settle, S, C, A, B, D, is the order of increasing distance.

WORKED EXAMPLE 1: THE DISTANCE TABLE

Question. Run Dijkstra's algorithm from S on the graph of Figure 5.7.1: edges S→A 7, S→C 3, C→A 2, A→B 4, C→B 8, C→D 9, B→D 2. Show the distance table after each step and give the shortest path from S to D.

Solution.

STEP	SETTLED	EDGES RELAXED	S	A	B	C	D
0	—	—	0	∞	∞	∞	∞
1	S (0)	S→A: 0+7 = 7; S→C: 0+3 = 3	0	7	∞	3	∞
2	C (3)	C→A: 3+2 = 5 < 7; C→B: 11; C→D: 12	0	5	11	3	12
3	A (5)	A→B: 5+4 = 9 < 11	0	5	9	3	12
4	B (9)	B→D: 9+2 = 11 < 12	0	5	9	3	11
5	D (11)	none	0	5	9	3	11

Parents: C←S, A←C, B←A, D←B. Path to D read backwards: D, B, A, C, S, so **S → C → A → B → D** with length 3 + 2 + 4 + 2 = **11**. The direct S→C→D route costs 12 and the S→A→B→D route costs 13; both lose.

WORKED EXAMPLE 2: WHY NEGATIVE EDGES BREAK IT

Question. Graph: S→A 2, S→B 5, B→A −4. Run Dijkstra from S and compare with the true shortest distance to A.

Solution. Step 1: settle S; dist A = 2, B = 5. Step 2: settle A (2): A is declared final at 2. Step 3: settle B (5); relax B→A: 5 + (−4) = 1 < 2, but A is already settled, so the update is not applied (or, if applied, A's neighbours are never re-examined). Dijkstra reports dist(A) = 2. The true shortest path is S→B→A with length 1. Bellman-Ford gets it: pass 1 relaxes S→A (2), S→B (5), B→A (1); pass 2 changes nothing; no negative cycle. Answer: Dijkstra's greedy choice assumed that leaving A could never get cheaper, which a negative edge violates.

EXAM TRAPS

- Dijkstra's key is dist[u] + w(u, v), the whole path; Prim's is w(u, v) alone. Write the relaxation line explicitly in your table.
- Settle the vertex with the smallest tentative distance, not the one you discovered first or the one with the smallest edge. Ties may be broken in any stated order.
- Once a vertex is settled its distance never changes. If your table changes a settled value, the trace is wrong (or the graph has a negative edge, which the question should have excluded).
- Negative weights: Dijkstra fails (even with no negative cycle); Bellman-Ford O(VE) works and detects negative cycles. Undirected graphs with a negative edge already contain a negative cycle (go back and forth).
- Favourite question: "Trace Dijkstra from the given source showing the distance table at each iteration; write the shortest path to vertex X and its cost." Add the path, not only the number.

CHECK YOURSELF

– REMEMBER **What does it mean to relax an edge (u, v)?** 2 marks

To test whether the path to v through u is shorter than the best known: if dist[u] + w(u, v) < dist[v], update dist[v] and set parent[v] = u.

– REMEMBER **State the time complexity of Dijkstra's algorithm with a binary heap.** 1 mark

O((V + E) log V), usually quoted as O(E log V) for connected graphs; O(V²) with a simple array.

– UNDERSTAND **Why does Dijkstra's algorithm require non-negative edge weights?** 3 marks

Its correctness proof assumes that extending any path can only make it longer, so the closest unsettled vertex cannot be reached more cheaply later. A negative edge can shorten a path after the vertex is settled, so the greedy finalisation is unsafe.

– UNDERSTAND **Why are V − 1 passes enough in Bellman-Ford?** 3 marks

A shortest path in a graph without negative cycles visits each vertex at most once, so it has at most V − 1 edges. After pass i every shortest path of at most i edges is correct; after V − 1 passes all are.

– APPLY **Vertices 1–4, edges 1→2 (1), 1→3 (4), 2→3 (2), 2→4 (6), 3→4 (1). Dijkstra from 1: give the settle order and all distances.** 4 marks

Settle 1 (0): 2 = 1, 3 = 4. Settle 2 (1): 3 = min(4, 3) = 3, 4 = 7. Settle 3 (3): 4 = min(7, 4) = 4. Settle 4 (4). Distances 0, 1, 3, 4; path to 4 is 1→2→3→4.

– APPLY **In the graph of Figure 5.7.1, reduce S→A from 7 to 4. Which distances change?** 3 marks

A becomes 4 (direct, better than 5), so B = 4 + 4 = 8 and D = 8 + 2 = 10. C stays 3. The tree edge into A changes from C→A to S→A.

– ANALYZE **A road network has 10^6 junctions and 2.5×10^6 roads. Compare the running time of array-based and heap-based Dijkstra, and say which a navigation app should use.** 4 marks

Array: $V^2 = 10^{12}$ steps, hours. Heap: $(V + E) \log V \approx 3.5 \times 10^6 \times 20 = 7 \times 10^7$ steps, well under a second. The graph is sparse (average degree 5), so the heap version wins by orders of magnitude; apps add further tricks (bidirectional search, contraction hierarchies) on top.

BEYOND THE SYLLABUS

Not for the exam. Edsger Dijkstra designed the algorithm in 1956 in about twenty minutes at a café, as he later told it, and published it in 1959. Link-state routing (OSPF, IS-IS) runs Dijkstra inside every router; distance-vector routing (RIP) runs Bellman-Ford. Navigation services precompute shortcuts (contraction hierarchies, ALT landmarks) so that continental queries take milliseconds; A* adds a heuristic estimate of remaining distance to the priority. Johnson's algorithm reweights a graph with Bellman-Ford so Dijkstra can be run from every source even with negative edges. Search: "contraction hierarchies", "A* algorithm", "Johnson's algorithm".

5.8 All-pairs shortest paths: Floyd-Warshall algorithm

A distance chart in a railway timetable gives the shortest distance between every pair of stations, not just from one source. You could run Dijkstra from each of the V vertices, but there is a cleaner way when the graph is small or dense: the Floyd-Warshall algorithm, three nested loops over a V × V matrix that in $O(V^3)$ time produces every shortest distance at once, and works with negative edge weights as long as there is no negative cycle. It is the unit's one dynamic-programming algorithm: instead of a greedy choice it builds the answer from answers to smaller subproblems, "shortest paths that may only pass through the first k vertices". A few syllabi list it by name, and it is also the natural example for syllabi that name dynamic programming in this unit.

> You need: the adjacency matrix from 5.2; relaxation from 5.7.

The subproblem

Number the vertices 1 to V. Define $d^{(k)}[i][j]$ as the length of the shortest path from i to j whose intermediate vertices (all vertices except the two ends) are drawn only from the set {1, 2, ..., k}. For k = 0 no intermediate vertex is allowed, so $d^{(0)}$ is just the weight matrix: w(i, j) for an edge, 0 on the diagonal, ∞ for no edge. For k = V every vertex is allowed and $d^{(V)}[i][j]$ is the true shortest distance δ(i, j).

The step from k − 1 to k has only two cases. Either the best path with intermediates in {1..k} does not use vertex k, so it is the old answer $d^{(k-1)}[i][j]$; or it does use k, exactly once (a shortest path never repeats a vertex when there is no negative cycle), so it is a best path i → k using only {1..k−1} followed by a best path k → j using only {1..k−1}. Hence the recurrence:

```
d(k)[i][j] = min( d(k-1)[i][j],  d(k-1)[i][k] + d(k-1)[k][j] )
```

This is the relaxation of 5.7, "is i → k → j better than what I have?", applied for every pair and every intermediate vertex in a fixed order.

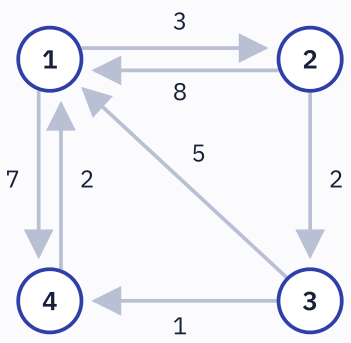
The algorithm

1. Set D = the weight matrix (0 on the diagonal, ∞ where there is no edge). Optionally set next[i][j] = j for every edge, for path recovery.
2. For k = 1 to V: for i = 1 to V: for j = 1 to V: if D[i][k] + D[k][j] < D[i][j], set D[i][j] = D[i][k] + D[k][j] and next[i][j] = next[i][k].

3. After the k loop, D[i][j] = δ(i, j). If any D[i][i] < 0, the graph has a negative cycle and the distances through it are meaningless.

One matrix is enough: in round k the entries D[i][k] and D[k][j] are not changed by round k itself (going through k to reach k gains nothing), so updating in place is safe. The outer loop must be k; putting k innermost is the standard wrong answer and gives incorrect distances. Three loops of V each: $\Theta(V^3)$ time, $\Theta(V^2)$ space, independent of E, which is why it suits dense graphs and small V (a few hundred) but not a million-vertex road map. Setting every weight to 1 and using OR/AND instead of min/+ gives Warshall's algorithm for the **transitive closure**: which vertices can reach which.

To read a path from the next matrix: start at i, then repeatedly move to next[current][j] until j is reached. Comparing with the alternatives: V runs of Dijkstra with a heap cost O(V E log V), which beats V^3 on sparse graphs but needs non-negative weights; V runs of Bellman-Ford cost $O(V^2 E)$, worse than V^3 in every case.



d_{21}: 8 direct → 5 via 2→3→4→1 (2+1+2)

$D^{(0)}$ = direct edges

	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

$D^{(4)}$ = all shortest paths

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

shaded = entries improved by going through an intermediate vertex

Figure 5.8.1 A four-vertex digraph, its starting matrix $D^{(0)}$ of direct edge weights, and the final matrix $D^{(4)}$ after the four rounds. Shaded cells changed during the run. The entry d[2][1] falls from 8 (the direct edge) to 5 by the route 2 → 3 → 4 → 1, which uses three edges of weights 2, 1 and 2.

WORKED EXAMPLE

Question. Apply Floyd-Warshall to the digraph with edges 1→2 (3), 1→4 (7), 2→1 (8), 2→3 (2), 3→1 (5), 3→4 (1), 4→1 (2). Show the matrix after each round and the shortest path from 2 to 1.

Solution. Rows are sources i, columns destinations j. Only the entries that change are discussed; each change is the test d[i][k] + d[k][j] < d[i][j].

```
D(0)            k = 1 (via 1)     k = 2 (via 2)
    1  2  3  4       1  2  3  4        1  2  3  4
1   0  3  ∞  7   1   0  3  ∞  7    1   0  3  5  7
2   8  0  2  ∞   2   8  0  2 15    2   8  0  2 15
3   5  ∞  0  1   3   5  8  0  1    3   5  8  0  1
4   2  ∞  ∞  0   4   2  5  ∞  0    4   2  5  7  0
```

1. k = 1: paths through 1. d[2][4] = d[2][1] + d[1][4] = 8 + 7 = 15 < ∞. d[3][2] = 5 + 3 = 8. d[4][2] = 2 + 3 = 5. (d[4][3] stays ∞ because d[1][3] = ∞.)
2. k = 2: paths through 2. d[1][3] = d[1][2] + d[2][3] = 3 + 2 = 5. d[4][3] = 5 + 2 = 7. d[1][4] = 3 + 15 = 18 is not < 7, no change.

```
k = 3 (via 3)     k = 4 (via 4) = final
    1  2  3  4        1  2  3  4
1   0  3  5  6    1   0  3  5  6
2   7  0  2  3    2   5  0  2  3
3   5  8  0  1    3   3  6  0  1
4   2  5  7  0    4   2  5  7  0
```

1. k = 3: paths through 3. d[1][4] = d[1][3] + d[3][4] = 5 + 1 = 6 < 7. d[2][1] = 2 + 5 = 7 < 8. d[2][4] = 2 + 1 = 3 < 15.
2. k = 4: paths through 4. d[2][1] = d[2][4] + d[4][1] = 3 + 2 = 5 < 7. d[3][1] = d[3][4] + d[4][1] = 1 + 2 = 3 < 5. d[3][2] = d[3][4] + d[4][2] = 1 + 5 = 6 < 8. No diagonal entry is negative: no negative cycle.
3. Path 2 to 1: the final improvement came through 4, and d[2][4] = 3 came through 3, so the path is **2 → 3 → 4 → 1** with length 2 + 1 + 2 = **5**; the direct edge 2→1 (8) is not used. With a next matrix: next[2][1] = 3, next[3][1] = 4, next[4][1] = 1.

Total work: 4 rounds × 16 cells = 64 relaxation tests, of which 11 produced an update.

EXAM TRAPS

- k must be the *outer* loop. Writing the loops as i, j, k gives a wrong matrix and is the most common loss of marks.
- In round k, row k and column k never change; use that as a check and copy them straight across. The diagonal stays 0 unless there is a negative cycle.
- Anything plus ∞ is ∞. Do not let a sum through a missing edge produce a number.
- Floyd-Warshall handles negative edges; Dijkstra does not. Neither handles a negative cycle: Floyd-Warshall reports it with a negative diagonal entry.
- Favourite question: "Apply Floyd-Warshall to the given 4-vertex graph showing $D^{(0)}$ to $D^{(4)}$." Write the recurrence first, then the matrices, then name the changed entries in each round.

CHECK YOURSELF

– REMEMBER **Write the Floyd-Warshall recurrence.** 2 marks

$d^{(k)}[i][j] = \min(d^{(k-1)}[i][j], d^{(k-1)}[i][k] + d^{(k-1)}[k][j])$, with $d^{(0)}$ the weight matrix.

– REMEMBER **State the time and space complexity of Floyd-Warshall.** 1 mark

Time $\Theta(V^3)$, space $\Theta(V^2)$ (one matrix, updated in place).

– UNDERSTAND **What does d (k) [i][j] mean, and why does the final matrix give true shortest paths?** 3 marks

The shortest i-to-j distance using only vertices from {1..k} as intermediates. When k = V every vertex is allowed, so no restriction remains and the value is the unrestricted shortest distance.

– UNDERSTAND **How does the algorithm reveal a negative cycle?** 2 marks

A negative cycle through vertex i makes a path from i back to i with negative total, so d[i][i] becomes negative; checking the diagonal at the end detects it.

– APPLY **Three vertices, edges 1→2 (4), 2→3 (−2), 1→3 (5). Run Floyd-Warshall and give d[1][3].** 4 marks

k = 1: nothing (no edges into 1). k = 2: d[1][3] = d[1][2] + d[2][3] = 4 − 2 = 2 < 5. k = 3: nothing. d[1][3] = 2 via 1→2→3. Dijkstra happens to agree here, because 2 (distance 4) settles before 3 (distance 5) and relaxes 2→3 in time; with a negative edge that is luck, not a guarantee (Worked example 2 of 5.7).

– APPLY **For V = 400, compare the operation counts of Floyd-Warshall and of 400 runs of heap Dijkstra on a graph with E = 2000.** 3 marks

Floyd-Warshall: $400^3 = 6.4 \times 10^7$. Dijkstra × V: $V \cdot E \log V \approx 400 \times 2000 \times 9 \approx 7.2 \times 10^6$, about nine times fewer; the sparse graph favours repeated Dijkstra, provided weights are non-negative.

– ANALYZE **Explain why updating the matrix in place during round k does not corrupt the result.** 4 marks

Round k only changes d[i][j] using d[i][k] and d[k][j]. Those two entries lie in column k and row k, and in round k they cannot improve: d[i][k] via k would be d[i][k] + d[k][k] = d[i][k] + 0. So every value read in round k is still its round-(k−1) value, and the in-place update equals the two-matrix version.

BEYOND THE SYLLABUS

Not for the exam. Robert Floyd published the three-loop form in 1962, building on Stephen Warshall's transitive-closure algorithm of the same year. The same triple loop with other operators computes the transitive closure (OR, AND), the widest path (max, min) and the number of paths; this family is the "algebraic path problem" or semiring framework. Some syllabi end this unit with complexity classes: shortest paths and MSTs are in P, while the closely related longest simple path and the travelling salesman problem are NP-hard, so no known algorithm solves them in polynomial time; that distinction is the opening theme of the Design and Analysis of Algorithms course. Search: "semiring shortest paths", "Johnson's algorithm", "P versus NP".

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Relaxation	if dist[u] + w(u, v) < dist[v] then update dist[v], parent[v] = u	5.7
Dijkstra	settle smallest dist; weights ≥ 0; $O(V^2)$ or $O((V + E) \log V)$	5.7
Bellman-Ford	V − 1 passes over all edges; negative weights OK; O(VE); V-th pass detects negative cycle	5.7
Floyd-Warshall	d(k)[i][j] = min(d(k−1)[i][j], d(k−1)[i][k] + d(k−1)[k][j]); k outer; $\Theta(V^3)$	5.8

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Remember and understand

- **5. Explain the relaxation step and prove that Dijkstra's algorithm settles each vertex with its correct shortest distance when all weights are non-negative. Why does the proof fail with negative edges? (10 marks)**

  Relaxation definition; invariant that settled vertices have final distances; the closest unsettled vertex u cannot be improved since any other path leaves the settled set through x with dist[x] ≥ dist[u] and non-negative continuation; negative edge breaks the "continuation cannot reduce" step; give the 3-vertex counterexample. (5.7)

Apply

- **10. Trace Dijkstra's algorithm from S on the digraph S→A 4, S→B 1, B→A 2, A→C 5, B→C 8, C→D 3, B→D 10, showing the distance table after each settled vertex, the shortest-path tree and the path to D. (12 marks)**

  Settle S (A4 B1); B (A3 C9 D11); A (C8); C (D11 stays, 8 + 3 = 11 not < 11); D. Distances S0 A3 B1 C8 D11; path S→B→A→C→D or S→B→D, both 11 (state the tie). (5.7)

- **11. Apply Floyd-Warshall to the digraph 1→2 (2), 1→3 (6), 2→3 (3), 3→1 (1), 3→4 (2), 4→2 (4), showing $D^{(0)}$ to $D^{(4)}$. Give the shortest path from 4 to 3. (16 marks)**

  D(0) rows: [0 2 6 ∞], [∞ 0 3 ∞], [1 ∞ 0 2], [∞ 4 ∞ 0]. k=1: d32 = 3. k=2: d13 = 5, d43 = 7. k=3: d14 = 7, d21 = 4, d24 = 5, d41 = 8. k=4: no change. Final rows [0 2 5 7], [4 0 3 5], [1 3 0 2], [8 4 7 0]. Path 4→2→3 = 7. (5.8)

Analyze and evaluate

- **12. A city has 2×10^5 junctions and 5×10^5 road segments. Choose a representation and algorithms for (a) checking whether the map is connected, (b) finding the fewest-turn route, (c) finding the fastest route with travel times, (d) planning the cheapest set of roads to keep open after a flood. Justify each with complexity. (16 marks)**

  Adjacency list (sparse: average degree 5). (a) one DFS/BFS, O(V + E), count restarts. (b) BFS, O(V + E). (c) Dijkstra with a heap, O(E log V); not Floyd-Warshall ($V^3 \approx 8 \times 10^{15}$). (d) MST: Kruskal O(E log E) or Prim with heap. (spans 5.2, 5.3, 5.5, 5.6, 5.7)

- **13. Compare Dijkstra's algorithm, Bellman-Ford and Floyd-Warshall on: problem solved, allowed weights, complexity, data structure, and when each is the right choice. (12 marks)**

  Single-source vs single-source vs all-pairs; weights ≥ 0 vs any (no negative cycle) vs any (no negative cycle); O(E log V) vs O(VE) vs $\Theta(V^3)$; heap vs edge list vs matrix; choose Dijkstra for non-negative sparse graphs, Bellman-Ford for negative edges or distributed routing, Floyd-Warshall for small dense graphs needing every pair. (5.7, 5.8)

- **14. Prim's algorithm and Dijkstra's algorithm have the same loop structure. Identify the single difference, show a 3-vertex graph on which they produce different trees, and explain why the MST need not contain shortest paths. (10 marks)**

  Key = w(u, v) vs dist[u] + w(u, v). Graph: A–B 2, B–C 2, A–C 3. MST from A: AB, BC (weight 4; path A to C in the tree is 4). Shortest-path tree from A: AB, AC (C at 3). The MST minimises total weight, not each path. (5.6, 5.7)

Glossary (terms used in this lesson)

Adjacency matrix	V × V array with M[i][j] = 1 or the weight when edge i→j exists; $\Theta(V^2)$ space.
Bellman-Ford	Shortest paths by V − 1 passes of relaxing all edges; O(VE); allows negative weights.
BFS	Breadth-first search: queue-based traversal by increasing distance in edges.
Degree	Number of edges at a vertex; in-degree and out-degree for digraphs.
Dijkstra's algorithm	Single-source shortest paths for non-negative weights by settling the closest unsettled vertex.
Floyd-Warshall	All-pairs shortest paths by dynamic programming over intermediate vertices; $\Theta(V^3)$.
Negative cycle	A cycle whose weights sum below zero; makes shortest paths undefined.
Path	Walk with no repeated vertex; its length is its edge count or weight sum.
Prim's algorithm	MST by growing one tree, always adding the cheapest edge leaving it.
Relaxation	Updating dist[v] when dist[u] + w(u, v) is smaller.
Shortest-path tree	Tree formed by the parent pointers after Dijkstra or Bellman-Ford.
Transitive closure	Matrix saying which vertices can reach which; Warshall's algorithm.
Weighted graph	Graph with a number w(u, v) on each edge.

Sources and further reading

Textbooks

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, Part VI "Graph Algorithms" (elementary graph algorithms, minimum spanning trees, single-source and all-pairs shortest paths). The standard reference for the proofs in 5.6 to 5.8.
- Mark Allen Weiss, *Data Structures and Algorithm Analysis in C*, chapter "Graph Algorithms" (topological sort, shortest paths, minimum spanning tree, DFS applications). Closest to the way many syllabi phrase this unit.
- Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed, *Fundamentals of Data Structures in C*, chapter on graphs (representations, traversals, spanning trees, shortest paths).
- Aaron M. Tenenbaum, Yedidyah Langsam and Moshe J. Augenstein, *Data Structures Using C*, chapter on graphs.
- Robert Sedgewick and Kevin Wayne, *Algorithms*, 4th edition, chapter 4 "Graphs" (undirected graphs, directed graphs, MSTs, shortest paths), with free online companion material.

Open web resources (opened while writing)

- VisuAlgo, "Graph Traversal (DFS/BFS)": animated BFS, DFS, topological sort and component finding on your own graphs, free. https://visualgo.net/en/dfsbfs
- CP-Algorithms, "Floyd-Warshall algorithm": loop order, negative-cycle handling and path reconstruction, with code. https://cp-algorithms.com/graph/all-pair-shortest-path-floyd-warshall.html
- Wikipedia, "Dijkstra's algorithm": history (conceived 1956, published 1959), the heap-based complexities quoted in 5.7, and pointers to A* and Johnson's algorithm. https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Syllabi: the universal syllabus and a range of university course documents.

Built for the University free n easy YouTube channel from the universal syllabus of Data Structures (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.