Data Structures

BFS, DFS and topological sort

The notes for the lesson “BFS, DFS and topological sort”: chapters 5.3 to 5.4, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

`2 chapters` `Chapters 5.3 to 5.4`

About these notes

These are the notes for the University free n easy video lesson **BFS, DFS and topological sort** on Data Structures: chapters 5.3 to 5.4, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

5.3 Breadth-first and depth-first traversal

A traversal visits every vertex reachable from a start vertex exactly once. There are two natural orders. Breadth-first search (BFS) spreads out in rings: all vertices one edge away, then two edges away, and so on, using a queue. Depth-first search (DFS) follows one path as far as it can, backs up, and tries the next branch, using a stack or recursion. Both run in O(V + E) with adjacency lists, and almost every later algorithm in this unit is one of them with extra bookkeeping: shortest paths in unweighted graphs come from BFS; topological order, cycle detection and components come from DFS.

You need: queues and stacks from an earlier lesson; adjacency lists from 5.2.

Breadth-first search

Keep a `visited` flag per vertex and a FIFO queue. Mark the start s visited and enqueue it. Then repeat: dequeue a vertex u, "visit" it (print it), and for each neighbour v of u that is not yet visited, mark v visited and enqueue it. Stop when the queue is empty.

1. visited[s] = true; enqueue(s); dist[s] = 0.
2. While the queue is not empty: u = dequeue(); output u.

3. For each v in adj[u]: if not visited[v], set visited[v] = true, dist[v] = dist[u] + 1, parent[v] = u, enqueue(v).

Marking a vertex when it is *enqueued*, not when it is dequeued, is what stops the same vertex entering the queue twice. The `dist` values are the number of edges on the shortest path from s, because a vertex at distance d is enqueued only after every vertex at distance d − 1 has been dequeued; the `parent` pointers form the BFS tree. This is the whole shortest-path algorithm for unweighted graphs, and the reason BFS matters in networking (fewest hops), social networks (degrees of separation) and puzzle solving (fewest moves).

```
/* C: BFS from s on adjacency lists adj[], V vertices */
void bfs(int s) {
    int queue[100], front = 0, rear = 0, visited[100] = {0};
    visited[s] = 1; queue[rear++] = s;
    while (front < rear) {
        int u = queue[front++];
        printf("%d ", u);
        for (Node *p = adj[u]; p != NULL; p = p->next)
            if (!visited[p->v]) { visited[p->v] = 1; queue[rear++] = p->v; }
    }
}
```

Each vertex enters the queue at most once (V operations) and each list is scanned once (2E node visits for undirected graphs): O(V + E). With an adjacency matrix the neighbour scan is V per vertex, so O(V²). Space is O(V) for the queue and flags.

Depth-first search

DFS is simplest as recursion: visit u, then for each unvisited neighbour v, call DFS(v). The call stack remembers where to come back to. Iteratively, replace the queue of BFS with a stack, with one subtlety: push neighbours without marking them, and mark a vertex when it is popped, skipping it if already marked; otherwise the order differs from the recursive one.

1. DFS(u): visited[u] = true; output u; record discovery time.
2. For each v in adj[u]: if not visited[v], parent[v] = u, DFS(v).
3. Record the finishing time of u when the loop ends; return.

Edges fall into classes. In the DFS tree a **tree edge** leads to a newly discovered vertex. In an undirected graph every other edge is a **back edge** to an ancestor, and a back edge proves a cycle (5.5). In a digraph there are also **forward edges** (to a descendant already finished) and **cross edges** (between subtrees); only a back edge, one that reaches a vertex still on the stack, means

a cycle. Finishing times give topological order (5.4). Time is again O(V + E); the recursion depth can reach V, which crashes on a deep graph unless you switch to the explicit stack.

If the graph is disconnected, wrap either search in a loop: for every vertex v, if not visited, start a new search from v. The number of restarts is the number of connected components (5.5).

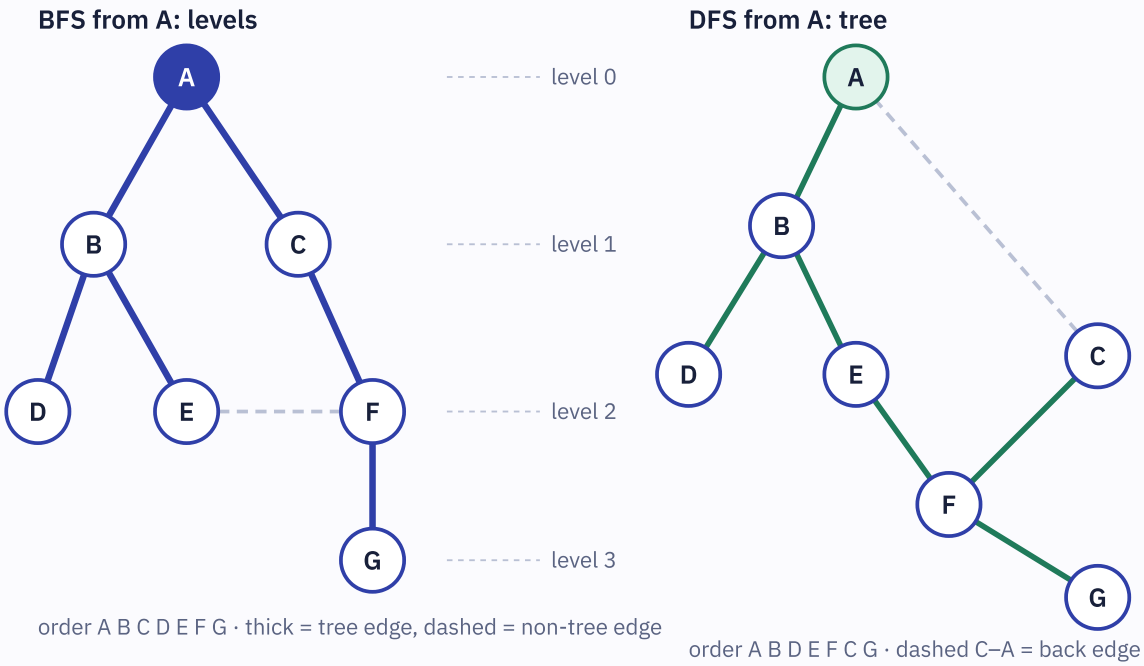


Figure 5.3.1 The same graph (edges A–B, A–C, B–D, B–E, C–F, E–F, F–G, lists in alphabetical order) searched both ways from A. BFS discovers in rings: level 1 is B, C; level 2 is D, E, F; G is at level 3, so the shortest path A to G has three edges. DFS plunges A→B→D, backs up to B, goes B→E→F and from F reaches C and G; the edge C–A then meets an already visited ancestor and is a back edge, the sign of the cycle A–B–E–F–C–A.

WORKED EXAMPLE

Question. Trace BFS and DFS from A on the graph of Figure 5.3.1, with each adjacency list in alphabetical order. Show the queue after every dequeue, give both visiting orders, and state the BFS distance of G from A.

Solution. Adjacency lists: A: B, C · B: A, D, E · C: A, F · D: B · E: B, F · F: C, E, G · G: F.

BFS. Start: visited {A}, queue [A].

STEP	DEQUEUED (OUTPUT)	NEIGHBOURS ENQUEUED	QUEUE AFTER	DIST
1	A	B, C	[B, C]	B 1, C 1
2	B	D, E (A visited)	[C, D, E]	D 2, E 2
3	C	F (A visited)	[D, E, F]	F 2
4	D	none	[E, F]	
5	E	none (B, F visited)	[F]	
6	F	G (C, E visited)	[G]	G 3
7	G	none	[]	

BFS order: **A B C D E F G**. dist(G) = 3, path A → C → F → G from the parent pointers (parent[G] = F, parent[F] = C, parent[C] = A).

DFS (recursive). DFS(A): output A; first unvisited neighbour B → DFS(B): output B; A visited; D → DFS(D): output D; B visited, return. Back in B: E → DFS(E): output E; B visited; F → DFS(F): output F; C → DFS(C): output C; A and F visited, return. Back in F: E visited; G → DFS(G): output G; return. All calls unwind. DFS order: **A B D E F C G**; finishing order D, C, G, F, E, B, A.

Both orders visit all seven vertices once. BFS puts C second because it is one edge from A; DFS reaches C sixth because it went deep through B first. The queue never held more than 3 vertices; the recursion depth reached 5 (A, B, E, F, C).

EXAM TRAPS

- State the neighbour order (alphabetical, or the order of your list) before the trace. With a different order the output differs and the examiner cannot follow your answer.
- In BFS mark a vertex visited when it is enqueued. Marking on dequeue lets a vertex enter the queue twice and produces duplicate outputs.
- BFS gives shortest paths only when every edge counts 1. With weights use Dijkstra (5.7); a BFS trace on a weighted graph earns nothing.
- Iterative DFS with a stack can give a different order from recursive DFS if you mark on push. Say which you are doing; examiners usually expect the recursive order.
- Favourite question: "Trace BFS and DFS on the given graph and draw the BFS tree and DFS tree." The tree uses parent pointers; leave non-tree edges out or draw them dashed.

CHECK YOURSELF

– REMEMBER **Which data structure does BFS use and which does DFS use?** 2 marks

BFS uses a FIFO queue; DFS uses a LIFO stack, usually the implicit call stack of recursion.

– REMEMBER **State the time complexity of BFS with adjacency lists and with an adjacency matrix.** 2 marks

O(V + E) with lists, O(V²) with a matrix, because each neighbour scan costs deg(u) in a list but V in a matrix row.

– UNDERSTAND **Why does BFS find the fewest-edges path from the source?** 3 marks

The queue always holds vertices in non-decreasing distance order, and a vertex at distance d is enqueued only when some vertex at distance d − 1 is processed. So the first time a vertex is discovered is via a shortest path, and its dist is final.

– UNDERSTAND **What is a back edge, and what does finding one in an undirected DFS tell you?** 3 marks

A non-tree edge from a vertex to one of its ancestors in the DFS tree (an already visited vertex that is not its parent). It closes a cycle: the tree path from the ancestor plus the back edge.

– APPLY **Graph: vertices 1–6, edges 1–2, 1–3, 2–4, 3–4, 4–5, 5–6. Give the BFS order from 1 (ascending neighbours) and dist(6).** 4 marks

Queue trace: [1] → 2, 3 → 4 → 5 → 6. Order 1 2 3 4 5 6. dist: 2, 3 at 1; 4 at 2; 5 at 3; 6 at 4. So dist(6) = 4.

– APPLY **Same graph: give the recursive DFS order from 1 with ascending neighbours.** 3 marks

1 → 2 → 4 → 3 (3's neighbours 1, 4 visited, return) → back in 4: 5 → 6. Order: 1 2 4 3 5 6.

– ANALYZE **You must find whether a maze of 10^6 cells has an exit and, if so, the shortest route. Which search, and why not the other?** 4 marks

BFS: it finds the exit on the fewest moves because it explores by distance, and its queue uses O(V) memory. Recursive DFS finds some route, not the shortest, and its recursion depth could reach 10^6, overflowing the stack; iterative DFS avoids the overflow but still gives no shortest-path guarantee.

BEYOND THE SYLLABUS

Not for the exam. Web crawlers are BFS with a frontier queue and a visited set of URLs; garbage collectors (an earlier lesson mentioned them) are a DFS or BFS over the object graph from the root references, marking what is reachable. Bidirectional BFS searches from both ends and meets in the middle, cutting the explored region drastically; iterative deepening DFS and A* search (BFS with a heuristic on a priority queue) are what AI and games courses build on top of this chapter. On huge graphs the frontier is processed in parallel on GPUs (direction-optimising BFS). Search: "bidirectional BFS", "A* search", "direction-optimizing BFS".

5.4 Topological sorting of directed acyclic graphs

When tasks depend on each other (take C before Data Structures, pour the foundation before the walls, compile a library before the program that links it), the dependencies form a directed graph, and a schedule that respects them is a topological order: a line-up of all vertices in which every edge u→v has u before v. Such an order exists exactly when the graph has no directed cycle, that is, when it is a DAG. Two algorithms produce one in O(V + E): Kahn's algorithm, which repeatedly removes a vertex with no incoming edges, and the DFS method, which lists vertices in reverse order of finishing time. It is a favourite exam question because a six-vertex trace fits on one page.

You need: in-degree from 5.1, DFS finishing times from 5.3, a queue from an earlier lesson.

What a topological order is, and when it exists

A topological order of a digraph G = (V, E) is a sequence of all vertices such that for every edge (u, v), u appears before v. If G has a cycle a→b→c→a, no order can put a before b, b before c and c before a, so cyclic graphs have no topological order. Conversely every DAG has at least one, and usually many: the order is total but the dependencies are partial, so independent vertices may be swapped. "Find a topological order" therefore has several correct answers; "find all of them" is a different and much harder question.

Why does a DAG always have a vertex with in-degree 0? Walk backwards along incoming edges; in a finite graph with no cycle the walk must stop, and it stops only at a vertex with nothing coming in. That vertex can go first, and removing it leaves a smaller DAG. This argument is Kahn's algorithm.

Kahn's algorithm (in-degree method)

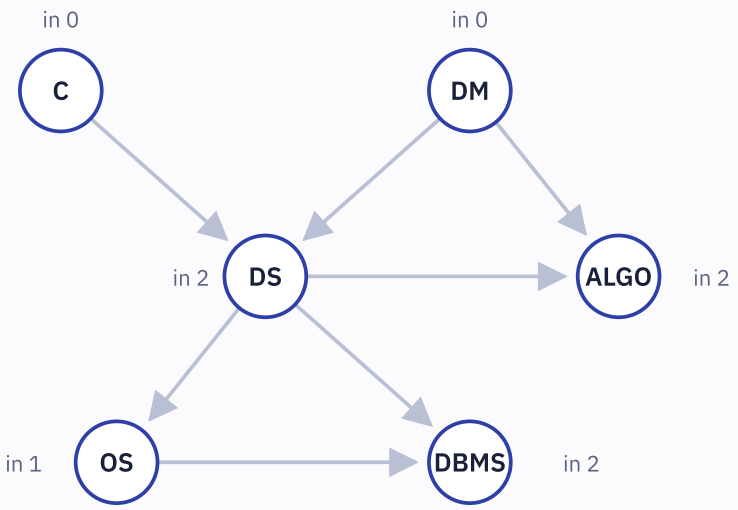
1. Compute indeg[v] for every vertex by scanning all adjacency lists: O(V + E).
2. Put every vertex with indeg 0 into a queue (any container works; a queue gives a deterministic answer).
3. While the queue is not empty: dequeue u, append u to the output, and for every edge (u, v) decrement indeg[v]; when indeg[v] reaches 0, enqueue v.
4. If the output has fewer than V vertices, the graph has a cycle: the vertices never output are on or downstream of it.

Each vertex is enqueued and dequeued once and each edge is examined once: O(V + E). The side effect in step 4 makes Kahn's algorithm the standard cycle test for digraphs (see 5.5). Using a min-priority queue instead of a FIFO queue gives the lexicographically smallest order, a common variant question.

DFS method (finishing times)

Run DFS from every unvisited vertex in turn. When a vertex finishes (all its out-neighbours are explored), push it on a stack. At the end, pop the stack: that is a topological order. The reason: for an edge (u, v) in a DAG, v always finishes before u. Either v is discovered from u, so v finishes inside u's call, or v was already finished when u looked at it (it cannot be still in progress, because that would make (u, v) a back edge and the graph cyclic). So reverse finishing order puts u before v. Cost O(V + E), one DFS.

The two methods generally give different valid orders; both are accepted. If an examiner asks for "the" order, follow their stated rule (usually alphabetical or numerical tie-breaking) and show the rule you used.



Kahn’s algorithm

queue [C, DM] → pop C

DS in 1

pop DM → DS 0, ALGO 1

pop DS → ALGO 0, OS 0

pop ALGO, pop OS → DBMS 0

pop DBMS → done

order: C DM DS ALGO OS DBMS

DFS method: DM C DS OS DBMS ALGO

Figure 5.4.1 A prerequisite DAG for six courses with each vertex's in-degree, and Kahn's algorithm run with a FIFO queue and ties broken in the order C, DM, DS, ALGO, OS, DBMS. C and DM start the queue; removing them frees DS; removing DS frees ALGO and OS; OS frees DBMS. The DFS method on the same graph gives a different but equally valid order.

WORKED EXAMPLE

Question. Courses C, DM, DS, ALGO, OS, DBMS have prerequisites C→DS, DM→DS, DM→ALGO, DS→ALGO, DS→OS, DS→DBMS, OS→DBMS. (a) Find a topological order by Kahn's algorithm with a FIFO queue, breaking ties in the listed order. (b) Find one by the DFS method, starting DFS in the listed order with out-neighbours in the listed order. (c) What happens to Kahn's algorithm if the edge DBMS→C is added?

Solution. (a) In-degrees: C 0, DM 0, DS 2, ALGO 2, OS 1, DBMS 2 (sum 7 = |E|).

STEP	DEQUEUE	DECREMENT	IN-DEGREES AFTER (DS, ALGO, OS, DBMS)	QUEUE AFTER	OUTPUT SO FAI
0	—	—	2, 2, 1, 2	[C, DM]	
1	C	DS → 1	1, 2, 1, 2	[DM]	C
2	DM	DS → 0 (enqueue), ALGO → 1	0, 1, 1, 2	[DS]	C DM
3	DS	ALGO → 0, OS → 0 (enqueue both), DBMS → 1	0, 0, 0, 1	[ALGO, OS]	C DM DS
4	ALGO	no out-edges	0, 0, 0, 1	[OS]	C DM DS ALGO
5	OS	DBMS → 0 (enqueue)	0, 0, 0, 0	[DBMS]	C DM DS ALGO OS
6	DBMS	no out-edges		[]	C DM DS ALGO OS DBMS

Six vertices output, so the graph is a DAG and the order is **C, DM, DS, ALGO, OS, DBMS**. Check each edge: every tail is left of its head.

(b) DFS from C: C → DS → ALGO (no out-edges, finish 1st) → back to DS → OS → DBMS (finish 2nd) → OS finishes 3rd → DS: DBMS already finished → DS finishes 4th → C finishes 5th. Next unvisited: DM; its neighbours DS and ALGO are finished, so DM finishes 6th. Finishing order ALGO, DBMS, OS, DS, C, DM; reversed: **DM, C, DS, OS, DBMS, ALGO**. Check: DM before DS and ALGO, C before DS, DS before ALGO, OS, DBMS, OS before DBMS. Valid, and different from (a).

(c) With DBMS→C, indeg[C] = 1. The initial queue is [DM] only. DM lowers DS to 1 and ALGO to 1; the queue empties with output just "DM". Output count 1 < 6, so Kahn's algorithm reports a cycle: C→DS→DBMS→C (through DS→DBMS). No schedule exists.

EXAM TRAPS

- Topological order exists only for DAGs. Any answer for a graph with a directed cycle is wrong; check for one first, or let Kahn's count tell you.
- In the DFS method the order is the *reverse* of finishing order, not the discovery (preorder) order. Discovery order C, DS, ALGO, OS, DBMS, DM puts DM after DS, which violates DM→DS.
- Decrement in-degrees of the *out-neighbours* of the removed vertex. Students sometimes decrement the removed vertex itself or its in-neighbours.
- More than one correct answer exists; the examiner checks that every edge points forward, not that your sequence matches the key. Write the edge check at the end.
- Favourite question: "Define topological sort; give Kahn's algorithm; apply it to the given DAG showing the in-degree table at each step."

CHECK YOURSELF

– REMEMBER **Define topological ordering.** 2 marks

A linear ordering of all vertices of a directed graph such that for every edge (u, v), u comes before v. It exists if and only if the graph is acyclic.

– REMEMBER **What is the time complexity of Kahn's algorithm with adjacency lists?** 1 mark

O(V + E): computing in-degrees scans all lists once, and each vertex is dequeued once and each edge decremented once.

– UNDERSTAND **Why must every DAG have at least one vertex of in-degree 0?** 3 marks

Start at any vertex and keep stepping backwards along an incoming edge. Since the graph is finite the walk must either repeat a vertex, which would be a cycle, or stop at a vertex with no incoming edge. Acyclic rules out the first, so the second happens.

– UNDERSTAND **How does the DFS method detect that no topological order exists?** 2 marks

If DFS meets an edge (u, v) where v is currently on the recursion stack (discovered but not finished), that is a back edge and the graph has a cycle; the algorithm stops and reports it.

– APPLY **Vertices 1–5, edges 1→2, 1→3, 2→4, 3→4, 4→5, 3→5. Give the Kahn order with a min-priority queue (smallest number first).** 4 marks

In-degrees 0, 1, 1, 2, 2. Remove 1 → 2, 3 free; take 2 → 4 to 1; take 3 → 4 free, 5 to 1; take 4 → 5 free; take 5. Order 1 2 3 4 5.

– APPLY **How many different topological orders does the DAG with edges a→c, b→c have? List them.** 3 marks

Two: a b c and b a c. c must be last; a and b are independent and may be swapped.

– ANALYZE **A build tool lists 12 modules whose dependencies form a DAG. Explain how topological order lets it also compute the minimum number of build "rounds" when independent modules build in parallel.** 4 marks

Process vertices in topological order and set level[v] = 1 + max level of its in-neighbours (0 for sources). Because all in-neighbours come earlier in the order, their levels are known. The number of rounds is the maximum level + 1, the number of vertices on the longest path; modules with equal level build together. Cost O(V + E).

BEYOND THE SYLLABUS

Not for the exam. Topological order is the heart of every build system (make, Gradle, Bazel), of package managers resolving install order (apt, pip, npm), of spreadsheet recalculation, and of deep-learning frameworks, which run the nodes of a computation graph in topological order forward and in reverse order for backpropagation. The longest path in a DAG, computed in the same order, gives the critical path of a project schedule (PERT/CPM) and the minimum rounds of a parallel build. Search: "critical path method", "DAG scheduling", "Kahn 1962".

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
BFS	queue; levels = shortest edge counts; O(V + E)	5.3
DFS	stack / recursion; tree, back, forward, cross edges; O(V + E)	5.3
Topological order	u before v for every edge u→v; exists iff DAG; Kahn or reverse DFS finish; O(V + E)	5.4

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

- **3. Write the BFS and DFS algorithms in pseudocode, state their complexities with both representations, and explain the roles of the queue and the stack. (12 marks)**

  BFS: mark on enqueue, dequeue-process-enqueue-neighbours loop. DFS: recursive visit with visited flag. O(V + E) with lists, O(V²) with matrix; queue gives level order, stack gives depth-first backtracking; mention parent pointers and BFS distances. (5.3)

Apply

- **6. For the undirected graph with vertices 0–5 and edges 0–1, 0–2, 1–3, 2–3, 2–4, 3–5, 4–5: draw the adjacency matrix and list, then trace BFS and DFS from 0 with ascending neighbour order, showing the queue or stack at every step and the two trees. (16 marks)**

  Matrix 6 × 6 with 14 ones; lists 0: 1 2; 1: 0 3; 2: 0 3 4; 3: 1 2 5; 4: 2 5; 5: 3 4. BFS order 0 1 2 3 4 5 (levels 0; 1, 2; 3, 4; 5). DFS order 0 1 3 2 4 5 (0→1→3→2→4→5). Draw both trees; non-tree edges dashed. (5.2, 5.3; spans chapters)

- **7. Find a topological order of the DAG with edges A→B, A→C, B→D, C→D, C→E, D→F, E→F by Kahn's algorithm, showing in-degrees after every removal. Then give the order from the DFS method and verify both. (12 marks)**

  In-degrees A0 B1 C1 D2 E1 F2. Kahn (FIFO, alphabetical): A B C D E F. DFS from A (alphabetical): A→B→D→F; F finishes first, then D, then B; back at A, C→D (visited), C→E→F (visited), so E finishes, then C, then A. Finishing order F D B E C A; reversed: A C E B D F. Check every edge in both. (5.4)

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)

- **15. Explain how DFS alone yields (a) a topological order, (b) a cycle test for digraphs and (c) the connected components of an undirected graph, and give the cost of each. (12 marks)**

  (a) reverse finishing order; (b) grey-vertex back edge; (c) restart count and labelling. Each O(V + E). Explain why (a) fails if (b) finds a back edge. (5.3, 5.4, 5.5)

Glossary (terms used in this lesson)

Adjacency list	One list of neighbours per vertex; Θ(V + E) space.
Adjacency matrix	V × V array with M[i][j] = 1 or the weight when edge i→j exists; $\Theta(V^2)$ space.
Back edge	A DFS edge to an ancestor (undirected: a visited non-parent; directed: a grey vertex); proves a cycle.
BFS	Breadth-first search: queue-based traversal by increasing distance in edges.
Connected component	Maximal set of mutually reachable vertices in an undirected graph.
DAG	Directed acyclic graph; the only digraphs with a topological order.
Degree	Number of edges at a vertex; in-degree and out-degree for digraphs.
DFS	Depth-first search: stack or recursion based traversal that goes deep before wide.

Kahn's algorithm	Topological sort by repeatedly removing in-degree-0 vertices.
Path	Walk with no repeated vertex; its length is its edge count or weight sum.
Topological order	Vertex sequence with every edge pointing forward.
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.