UNIT II · LESSON 06

Graph Algorithms

representation · BFS · DFS · topological sort · MST · shortest paths · Floyd–Warshall · network flow

20
questions in Unit II
Lists · Trees · Sorting · Hashing · Graphs

BASICS GRAPH TERMS

- G = (V, E); **directed** (digraph) or **undirected**; weighted or not
- Degree (undirected): Σ deg(v) = **2|E|** (handshaking); odd-degree vertices are even in number
- Digraph: in-degree, out-degree; Σ in = Σ out = |E|
- Max edges: undirected **n(n−1)/2**, directed **n(n−1)**
- **Connected**: path between every pair; **strongly connected** (digraph): directed path both ways
- **Tree** = connected acyclic, n − 1 edges; **forest** = acyclic
- **DAG** = directed acyclic graph; **complete** K_n; **bipartite**: no odd cycle
- Euler circuit: all degrees even; Hamiltonian cycle: visits each vertex once (NP-complete)

MATRIX VS LIST REPRESENTATIONS

Feature	Adjacency matrix	Adjacency list
Space	$O(V^2)$	O(V + E)
Edge (u,v)?	O(1)	O(deg u)
All neighbours of u	O(V)	O(deg u)
Best for	dense graphs	sparse graphs

- Undirected matrix is **symmetric**; list stores each edge **twice**
- Incidence matrix: V × E

STACK / RECURSION DFS

```
void dfs(int u) {
  vis[u] = 1; printf("%d ", u);
  for (v = 0; v < n; v++)
    if (adj[u][v] && !vis[v]) dfs(v);
}
```

- O(V + E); edges classified: **tree, back, forward, cross**
- **Back edge ⇔ cycle** (directed graph)
- Uses: cycle detection, topological sort, SCCs (Kosaraju/Tarjan), articulation points, bridges, maze solving

QUEUE BFS

```
void bfs(int s) {
  enqueue(s); vis[s] = 1;
  while (!empty()) {
    int u = dequeue(); printf("%d ", u);
    for (v = 0; v < n; v++)
      if (adj[u][v] && !vis[v]) {
        vis[v] = 1; enqueue(v);
      }
  }
}
```

- Level by level; gives **shortest path (edge count)** in unweighted graphs
- O(V + E) with lists, $O(V^2)$ with matrix
- Uses: connected components, bipartite test, web crawling

DAG ONLY TOPOLOGICAL SORT

- Linear order such that every edge u → v has u **before** v
- **Kahn**: repeatedly remove vertices of **in-degree 0** (queue)
- **DFS**: output vertices in **decreasing finish time**
- Not unique; exists **iff** graph is a DAG; O(V + E)
- Uses: course prerequisites, build/compile order, task scheduling

MST MINIMUM SPANNING TREE (CONTD.)

- Both are **greedy** algorithms; Borůvka is a third

MST MINIMUM SPANNING TREE

Feature	Prim	Kruskal
Grows	one tree from a start vertex	forest, joins components
Picks	cheapest edge leaving the tree	cheapest edge overall without cycle
Data structure	min-heap (priority queue)	sort edges + **union–find**
Time	O(E log V) (heap), $O(V^2)$ matrix	O(E log E)
Best for	dense graphs	sparse graphs

- MST has **V − 1** edges; unique if all weights distinct
- **Cut property**: lightest edge across any cut is in some MST
- **Cycle property**: heaviest edge on a cycle is not in the MST

SINGLE SOURCE SHORTEST PATHS

Algorithm	Weights	Time	Technique
BFS	unweighted	O(V + E)	queue
Dijkstra	**non-negative**	O((V + E) log V) heap; $O(V^2)$ array	greedy
Bellman–Ford	negative allowed	O(V·E)	DP, relax V−1 times
DAG shortest path	any (no cycles)	O(V + E)	topological order

- **Relax**(u, v): if d[v] > d[u] + w(u,v) then d[v] = d[u] + w, π[v] = u
- Bellman–Ford: if any edge still relaxes in pass V → **negative cycle**
- Dijkstra can give wrong answers with negative edges

SOURCE A DIJKSTRA TRACE

- Edges: A–B 4, A–C 1, C–B 2, B–D 1, C–D 5

Step	Picked	d(A)	d(B)	d(C)	d(D)
init	—	0	∞	∞	∞
1	A	0	4	1	∞
2	C	0	**3**	1	6
3	B	0	3	1	**4**
4	D	0	3	1	4

FLOYD–WARSHALL ALL-PAIRS

```
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (d[i][k] + d[k][j] < d[i][j])
        d[i][j] = d[i][k] + d[k][j];
```

- **Dynamic programming**; $O(V^3)$ time, $O(V^2)$ space
- Negative edges OK; negative cycle if some d[i][i] < 0
- Loop order **k outermost** is essential
- Transitive closure version = **Warshall** (OR/AND)
- Johnson's algorithm: O(V E log V) for sparse graphs

MAX-FLOW NETWORK FLOW

- Flow network: source s, sink t, capacity c(u,v); flow ≤ capacity; **conservation** at inner vertices
- **Residual capacity** $c_f(u,v) = c(u,v) - f(u,v)$, plus reverse edge f(u,v)
- **Augmenting path**: s → t path in the residual graph; push its bottleneck
- **Ford–Fulkerson**: repeat until no augmenting path; $O(E \cdot |f^*|)$ for integer capacities
- **Edmonds–Karp**: choose shortest path by **BFS** → $O(V E^2)$
- **Max-flow = min-cut** theorem: max flow value = capacity of the smallest s–t cut
- Uses: bipartite matching, transport, network reliability

TRAP EXAM TRAPS

- Dijkstra + negative edge → wrong result (even without a negative cycle)
- Floyd–Warshall with k as the inner loop is **wrong**
- MST is defined for **undirected** connected graphs
- Shortest-path tree ≠ MST in general
- Kruskal on a disconnected graph gives a minimum spanning **forest**
- BFS shortest path holds only when all edges have equal weight

TRACE MAX-FLOW EXAMPLE

- s→a 10, s→b 5, a→b 15, a→t 5, b→t 10

Augmenting path	Bottleneck	Total flow
s → a → t	5	5
s → b → t	5	10
s → a → b → t	5	**15**

- Min cut {s, a, b} | {t}: 5 + 10 = **15** ✓

Q → A RAPID RECALL

Edges in complete graph K_6	15
BFS data structure	queue
DFS data structure	stack
Shortest path, unweighted graph	BFS
Algorithm for negative edges	Bellman–Ford
Detects negative cycles	Bellman–Ford (pass V) / Floyd (d[i][i] < 0)
All-pairs shortest path complexity	$O(V^3)$
Edges in MST of 10 vertices	9
Kruskal needs	sorting + disjoint-set (union–find)
Topological order exists only for	DAG
Max-flow =	min-cut capacity
Edmonds–Karp augments along	shortest (BFS) paths