GUJARAT TECHNOLOGICAL UNIVERSITY

BE-5 SEMESTER – OLD PAPER – S22 TO W25 – QUESTION BANK ANSWER

Subject Name & Code:

ANALYSIS AND DESIGN OF ALGORITHMS (3150703)

Unit 6: Graph Algorithms & Traversals

Note for Q4, Q5, Q9, Q11: The CQB does not provide the specific graphs. The answers below include the **algorithm/method** plus a generic example. Replace with the actual graph data from your exam.

Q1 – Explain DFS traversal with example and write its time complexity. (4 marks)
Ans:
Depth-First Search (DFS) – Graph traversal that explores as far as possible along each branch before backtracking. Uses a stack (explicit or recursion).
Algorithm:

```
DFS(vertex v):
   mark v as visited
   for each neighbor u of v:
      if u not visited:
         DFS(u)
```

Example graph:
Vertices: A, B, C, D, E
Edges: A-B, A-C, B-D, B-E, C-E
DFS starting from A:
Visit A → go to B (neighbor) → visit B → go to D → visit D (no unvisited neighbors) → backtrack to B → go to E → visit E → backtrack to B → backtrack to A → go to C → visit C → done.
Traversal order: A, B, D, E, C
DFS tree: Edges used: A-B, B-D, B-E, A-C (tree edges). Back edges: C-E (if undirected, also A-C already tree).
Time complexity:

- Adjacency list: O(V + E)
- Adjacency matrix: $O(V^2)$

Space complexity: O(V) for stack and visited array.
Real-world application: Maze solving, cycle detection, topological sorting, strongly connected components.

Q2 – Explain BFS traversal with example and differentiate from DFS. (4 marks)
Ans:
Breadth-First Search (BFS) – Graph traversal that explores all neighbors at the current depth before moving to next level. Uses a queue.

Algorithm:

```
BFS(start):
   queue = empty
   mark start visited, enqueue start
   while queue not empty:
      v = dequeue
      for each neighbor u of v:
         if u not visited:
            mark u visited, enqueue u
```

Example graph (same as Q1):
Edges: A-B, A-C, B-D, B-E, C-E
BFS from A:
Queue: [A] → dequeue A, visit neighbors B,C → queue [B,C]
Dequeue B, visit D,E (C already in queue but not visited? C is visited from A) → add D,E → queue [C,D,E]
Dequeue C, neighbor E already visited → skip → queue [D,E]
Dequeue D → no new → queue [E]
Dequeue E → done.
Traversal order: A, B, C, D, E
Differentiate BFS vs DFS:

Parameter	BFS	DFS
Data structure	Queue	Stack (or recursion)
Traversal pattern	Level by level	Depth first, then backtrack
Shortest path	Finds shortest path in unweighted graph	Does not guarantee shortest path
Space complexity	O(V) (queue size)	O(V) (recursion stack)
Use cases	Shortest path, web crawling	Cycle detection, topological sort, SCC
Completeness	Complete	Complete (for finite graphs)

Real-world application: Shortest path in GPS, social network friend suggestions (BFS), puzzle solving.

Q3 – Define articulation point, graph, tree. (3 marks)
Ans:

Term	Definition
Graph	A non-linear data structure consisting of a set of vertices (nodes) and a set of edges (connections) between them. Denoted as G(V, E).

Term	Definition
Tree	A connected acyclic undirected graph. Any two vertices have exactly one path. A tree with n vertices has exactly n-1 edges.
Articulation point (cut vertex)	A vertex in a graph whose removal increases the number of connected components. Removing it disconnects the graph.

Example of articulation point: In a graph A-B-C (path), removing B disconnects A and C, so B is articulation point. In a cycle, no articulation points.
Real-world application: Network reliability – identifying critical routers or bridges whose failure would split the network.

Q4 – Find topological sort of given graph. (4 marks)

⚠️ **No graph provided.** Below is the **method** with a generic example.
Topological sort – Linear ordering of vertices in a directed acyclic graph (DAG) such that for every directed edge u→v, u comes before v.
Algorithm (Kahn's – BFS based):

1. Compute indegree of all vertices.
2. Enqueue all vertices with indegree 0.
3. While queue not empty:
    - Dequeue vertex v, add to order.
    - For each neighbor u of v, decrement indegree[u]; if becomes 0, enqueue u.
4. If order size < V → graph has cycle (no topological sort).

Generic example graph:
Vertices: 1,2,3,4,5
Edges: 1→2, 1→3, 2→4, 3→4, 4→5
Step-by-step:

- Indegrees: 1:0, 2:1, 3:1, 4:2, 5:1
- Queue: [1]
- Dequeue 1 → order [1]; reduce indegrees of 2,3 → indegree2=0, indegree3=0 → enqueue 2,3
- Dequeue 2 → order [1,2]; reduce indegree of 4 → indegree4=1
- Dequeue 3 → order [1,2,3]; reduce indegree of 4 → indegree4=0 → enqueue 4
- Dequeue 4 → order [1,2,3,4]; reduce indegree of 5 → indegree5=0 → enqueue 5
- Dequeue 5 → order [1,2,3,4,5]

Topological order: 1,2,3,4,5 (other valid orders exist, e.g., 1,3,2,4,5)
Time complexity: O(V+E)
Real-world application: Course prerequisite scheduling, task dependency resolution (makefiles), build systems.

Q5 – Write DFS traversal and draw DFS tree for given graph. (4 marks)

⚠️ **No graph provided.** Below is the method with a generic example.
DFS traversal – as explained in Q1. To draw DFS tree:

- Mark tree edges (edges used to discover new vertices).
- Mark back edges (edges to ancestors), forward edges (to descendants not direct child), cross edges (in directed graphs).

Generic example graph (undirected):
Vertices: A, B, C, D, E
Edges: A-B, A-C, B-D, B-E, C-E

DFS from A (adjacency list order: A: B,C; B: A,D,E; C: A,E; D: B; E: B,C):

- Visit A (discovery time 1)
  - Go to B (time 2)
    - Go to D (time 3) – D finished (time 4)
    - Back to B, go to E (time 5) – E finished (time 6)
  - Back to B, B finished (time 7)
  - Back to A, go to C (time 8) – C finished (time 9)
- A finished (time 10)

DFS tree edges: A-B, B-D, B-E, A-C (all tree edges). No back edges in undirected graph (all non-tree edges are back edges to ancestors: C-E is a back edge because E is already visited? Actually E visited before C, so C-E connects C to E which is not ancestor? In undirected DFS, any edge not in tree is a back edge.)
Drawing instructions: Root at A, children B and C. B has children D and E.
Time complexity: O(V+E)

Q6 – Explain in-order, pre-order, post-order traversals of graph. (3 marks)
Important clarification: In-order, pre-order, post-order are **tree traversals**, not general graph traversals (graphs can have cycles). For a tree (a special graph), these are defined.
For a binary tree (or general tree with ordered children):

Traversal	Order of visiting	Use case
Pre-order	Visit root → left subtree → right subtree	Copying tree, prefix expression
In-order	Left subtree → root → right subtree	BST sorted order
Post-order	Left subtree → right subtree → root	Deleting tree, postfix expression

Example tree:
Root A, left child B (with left D, right E), right child C (with left F)

- **Pre-order:** A, B, D, E, C, F
- **In-order:** D, B, E, A, F, C
- **Post-order:** D, E, B, F, C, A

Note for exam: If question says "of graph", specify that graph must be a tree or forest. For cyclic graphs, these traversals are not uniquely defined.

Q7 – State and explain methods to represent graph. (4 marks)
Ans:
Two standard graph representation methods:

Method	Description	Space	Edge existence check	Neighbor iteration
Adjacency Matrix	2D array A[i][j]=1 if edge i→j exists	$O(V^2)$	O(1)	O(V)
Adjacency List	Array of lists; each list stores neighbors of a vertex	O(V+E)	O(degree)	O(degree)

Adjacency Matrix example:
Vertices {0,1,2,3} with edges: 0-1, 0-2, 1-3
Matrix (4×4):
Row0: [0,1,1,0]
Row1: [1,0,0,1]
Row2: [1,0,0,0]
Row3: [0,1,0,0]
For undirected, symmetric; for directed, may not be symmetric.
Adjacency List example (same graph):
0 → [1,2]
1 → [0,3]
2 → [0]
3 → [1]
Weighted graph: Matrix stores weights (or ∞ if no edge). List stores (neighbor, weight) pairs.
Choice: Use adjacency list for sparse graphs (most real-world graphs). Use matrix for dense graphs or when frequent edge existence checks needed.
Real-world application: Social networks (adjacency list), road maps (weighted list), circuit design (matrix).

Q8 – Differentiate BFS vs DFS. (4 marks)
Ans: (Already covered in Q2 difference table. Provide expanded table.)

Feature	BFS	DFS
Data structure	Queue	Stack (implicit recursion)
Traversal order	Level order (breadth-wise)	Depth-wise
Memory	O(V) in worst case (queue)	O(V) recursion stack (worst for skewed)
Shortest path (unweighted)	Yes	No
Cycle detection	Yes (if back edge to same level)	Yes (back edge)
Connected components	Yes	Yes
Topological sort	Kahn's algorithm (BFS)	Using finishing times (DFS)
SCC (Kosaraju)	Uses DFS	Used for first pass
Implementation	Iterative	Recursive or iterative
Example use	GPS shortest path, web crawling	Maze solving, topological sort

Conceptual difference: BFS explores all neighbors first; DFS goes to the deepest node and

backtracks.

Q9 – Write algorithm for Prim's method and find MST for given graph. (7 marks)

⚠️ **No graph provided.** Below is the algorithm and a generic example.

Prim's Algorithm (greedy) – finds Minimum Spanning Tree (MST):

```
Prim(Graph, start):
   visited = set()
   MST = empty list
   minHeap = priority queue (edge weight, vertex, parent)
   visited.add(start)
   for each neighbor v of start with weight w:
      push (w, v, start) into minHeap
   while minHeap not empty and visited size < V:
      (weight, v, parent) = pop min
      if v not visited:
         visited.add(v)
         MST.add( (parent, v, weight) )
         for each neighbor u of v with weight w_u:
            if u not visited:
               push (w_u, u, v) into minHeap
   return MST
```

Generic example graph: (same as in Q4 of Unit 5)

Vertices: A,B,C,D,E

Edges: A-B(4), A-C(3), A-E(5), B-C(1), B-D(2), C-D(4), C-E(3), D-E(2)

Trace from A:

- visited={A}, heap: (3,C,A), (4,B,A), (5,E,A)
- pop (3,C,A): C not visited → add edge A-C, visited={A,C}, add edges from C: (1,B,C), (4,D,C), (3,E,C) → heap: (1,B,C), (4,B,A), (3,E,C), (5,E,A), (4,D,C)
- pop (1,B,C): B not visited → add edge C-B, visited={A,B,C}, add edges from B: (2,D,B), (4,A,B but A visited) → heap: (2,D,B), (3,E,C), (4,D,C), (5,E,A)
- pop (2,D,B): D not visited → add edge B-D, visited={A,B,C,D}, add edges from D: (2,E,D), (4,C,D visited) → heap: (2,E,D), (3,E,C), (5,E,A)
- pop (2,E,D): E not visited → add edge D-E, visited all → stop.

MST edges: A-C(3), C-B(1), B-D(2), D-E(2) → total = 8

Complexity: O(E log V) with binary heap; $O(V^2)$ with adjacency matrix.

Real-world application: Network design, cluster analysis, approximation for TSP.

Q10 – Explain connected components. (3 marks)

Ans:

Connected components – In an undirected graph, a connected component is a maximal set of vertices where any two vertices are connected by a path. Components are disjoint and partition the graph.

Finding components: Run BFS/DFS from each unvisited vertex; each search identifies one component.

Example: Graph with vertices 1..6, edges: (1-2), (2-3), (4-5). Components: C1={1,2,3}, C2={4,5}, C3={6} (isolated vertex).

For directed graphs: Use **strongly connected components** (SCC) – maximal sets where every vertex reaches every other via directed path.

Properties:

- Number of components = number of times BFS/DFS is called.
- A tree has 1 component.
- Empty graph has V components.

Real-world application: Image segmentation (connected pixels), social network clusters, network resilience.

Q11 – Find strongly connected components from given graph. (4 marks)

⚠️ **No graph provided.** Below is the method (Kosaraju's algorithm) with a generic example.

Strongly Connected Components (SCC) – Maximal subsets where every vertex is reachable from every other in the subset (directed graph).

Kosaraju's Algorithm:

1. Perform DFS on original graph, push vertices to stack in order of completion (post-order).
2. Reverse the graph (reverse all edge directions).
3. Pop vertices from stack, run DFS on reversed graph; each DFS tree gives one SCC.

Generic example graph:

Vertices: 1,2,3,4

Edges: 1→2, 2→3, 3→1, 2→4

Step 1 – DFS (original):

Start from 1: 1→2→3→(back to 1) → 2→4. Finishing times: push 4, then 3, then 2, then 1? Order depends on implementation. Stack (bottom to top): [4,3,2,1] (1 is last finished).

Step 2 – Reverse graph edges: 2→1, 3→2, 1→3, 4→2

Step 3 – DFS on reverse graph using stack order (pop 1):

DFS from 1 on reverse: 1→3→2→(2 also has 1) → set {1,3,2} is one SCC. Next pop 4 (not visited): DFS from 4→2 (but 2 already in SCC) → {4} alone is SCC.

SCCs: {1,2,3} and {4}

Time complexity: O(V+E)

Alternative: Tarjan's algorithm (single DFS) also O(V+E).

Real-world application: Finding cycles in dependencies, compilers (optimization), social network strongly connected groups.

Q12 – Explain depth-first traversal using suitable example. (4 marks)

Ans: (Similar to Q1, but emphasis on example.)

Depth-First Traversal (DFT) – A graph traversal that explores as far as possible along each branch before backtracking. Recursive or stack-based.

Example graph (directed):

Vertices: P, Q, R, S

Edges: P→Q, P→R, Q→S, R→S

DFS starting from P (adjacency list order: Q then R):

- Visit P (mark visited)
- Go to Q (unvisited)
  - Go to S (unvisited)
    - S has no outgoing → backtrack to Q
  - Q done, backtrack to P
- Go to R (unvisited)
  - R→S, but S already visited → skip
  - R done

DFS traversal order: P, Q, S, R

DFS tree edges: P-Q, Q-S, P-R (tree edges). R-S is a cross/forward edge.

Applications: Cycle detection, topological sort, solving puzzles (N-Queens, Sudoku).

Q13 – Explain breadth-first traversal method for graph with algorithm and example. (4 marks)
Ans: (Similar to Q2, with algorithm and example.)
Breadth-First Traversal (BFT) – Level-order traversal using a queue.
Algorithm:

```
BFT(start):
   create empty queue Q
   mark start visited, enqueue start
   while Q not empty:
      v = dequeue Q
      process v
      for each neighbor u of v:
         if u not visited:
            mark u visited, enqueue u
```

Example graph (same as Q12 directed):
Edges: P→Q, P→R, Q→S, R→S
BFS from P:

- Queue: [P] → dequeue P, process P, enqueue Q and R → queue [Q,R]
- Dequeue Q, process Q, enqueue S → queue [R,S]
- Dequeue R, process R, neighbor S already visited/enqueued? Not visited yet? Actually S is in queue but not processed – if we check visited before enqueue, S will not be enqueued again. Enqueue S only once. Queue [S]
- Dequeue S, process S, no new → done.

BFS order: P, Q, R, S
BFS tree: Edges P-Q, P-R, Q-S (tree edges). R-S not used because S already discovered.
Applications: Shortest path in unweighted graph, web crawling, social network "friends of friends".

Q14 – Define acyclic directed graph, back edge. (3 marks)
Appeared in: W25 (Q4a, 03 marks)
Ans:

Term	Definition
Acyclic directed graph (DAG)	A directed graph that contains **no directed cycles**. Equivalently, it is a directed graph where it is impossible to start at a vertex and follow a sequence of directed edges to return to the same vertex. DAGs always have at least one topological ordering.
Back edge	In the context of DFS traversal of a graph, a back edge is an edge (u, v) such that v is an ancestor of u in the DFS tree (i.e., already in the recursion stack). In undirected graphs, every non-tree edge is a back edge. In directed graphs, back edges indicate the presence of a cycle.

Real-world application: DAGs model task dependencies (build systems, course prerequisites). Detecting back edges identifies cycles (e.g., deadlock detection).

Q14 – Find prefix code for given frequencies. (4 marks)
Appeared in: W25 (Q4b, 04 marks)

⚠️ **Note:** This is the **Huffman coding** problem (greedy algorithm). The specific frequencies are **not provided** in the image. Below is the **method** with a generic example. Replace with actual frequencies from your exam.

Prefix code – A code where no codeword is a prefix of another (prefix-free). Huffman's algorithm produces an optimal prefix code given character frequencies.

Method (Huffman algorithm):

1. Create a leaf node for each character with its frequency.
2. Repeatedly take two nodes with smallest frequencies, merge into a new internal node with frequency = sum, insert back.
3. Continue until one node remains (root).
4. Traverse tree: assign 0 to left edge, 1 to right edge. Codewords are the path from root to leaf.

Generic example frequencies: A:5, B:9, C:12, D:13, E:16, F:45

Huffman tree construction:

- Combine A(5)+B(9)=14 → list: 12(C),13(D),14,16(E),45(F)
- Combine C(12)+D(13)=25 → list: 14,16(E),25,45(F)
- Combine 14+16(E)=30 → list: 25,30,45(F)
- Combine 25+30=55 → list: 45(F),55
- Combine 45+55=100 (root)

Prefix codes (0=left, 1=right – assuming left child smaller frequency):

- F: 0
- C: 100, D: 101, E: 111
- A: 1100, B: 1101

Verification: No code is prefix of another → valid prefix code.

Real-world application: ZIP file compression, JPEG (Huffman tables), MP3.

Q15 – Write algorithm for Kruskal's method and find MST for given graph. (7 marks)

Appeared in: W25 (Q4c, 07 marks)

⚠️ **Note:** The specific graph is **not provided** in the image. Below is the **complete algorithm** plus a generic example. Replace with the actual graph from your exam.

Kruskal's Algorithm (greedy) – Minimum Spanning Tree (MST):

text

```
Kruskal(Graph):
   sort all edges by weight ascending
   Initialize disjoint sets: each vertex in its own set
   MST = empty list
   for each edge (u, v, weight) in sorted order:
      if find(u) != find(v):
         add edge to MST
         union(u, v)
      if |MST| == |V| - 1:
         break
   return MST
```

Time complexity: O(E log E) due to sorting, near O(E log V).

Generic example graph (undirected, weighted):

Vertices: A, B, C, D, E

Edge	Weight
A-B	4

Edge	Weight
A-C	3
A-E	5
B-C	1
B-D	2
C-D	4
C-E	3
D-E	2

Trace Kruskal's algorithm:

1. Sort edges by weight:
   B-C(1), B-D(2), D-E(2), A-C(3), C-E(3), A-B(4), C-D(4), A-E(5)
2. Initialize: each vertex separate set: {A},{B},{C},{D},{E}

Edge	Weight	Action	MST edges	Disjoint sets
B-C	1	Add	B-C	{A},{B,C},{D},{E}
B-D	2	Add	B-C, B-D	{A},{B,C,D},{E}
D-E	2	Add	B-C, B-D, D-E	{A},{B,C,D,E}
A-C	3	Add	+ A-C	{A,B,C,D,E} (all connected)
C-E	3	Skip (cycle)	-	-
A-B	4	Skip (cycle)	-	-
C-D	4	Skip (cycle)	-	-
A-E	5	Skip (cycle)	-	-

MST edges: B-C(1), B-D(2), D-E(2), A-C(3)
Total weight = 1+2+2+3 = 8
Drawing the MST:
Vertices A,B,C,D,E with edges: A–C, C–B, B–D, D–E.
Real-world application: Designing minimum cost network (roads, cables, pipelines).

****