Graph Theory Problem Set - Solutions

Comprehensive Solutions with Explanations

Section A: Multiple Choice Questions - Solutions

Q1) Select all statements that are TRUE about adjacency matrix representation:

Correct Answers: A, B, E

> ***A) TRUE:*** *An adjacency matrix for V vertices requires V×V space, hence $O(V^2)$.*
>
> ***B) TRUE:*** *To check if edge (i,j) exists, we simply access matrix[i][j], which is O(1).*
>
> ***C) FALSE:*** *Adding a new vertex requires resizing the matrix from V×V to (V+1)×(V+1), which takes $O(V^2)$ time to copy all elements.*
>
> ***D) FALSE:*** *For sparse graphs (few edges), adjacency list O(V+E) is better than adjacency matrix $O(V^2)$ when $E << V^2$.*
>
> ***E) TRUE:*** *In an undirected graph, if there's an edge between i and j, then matrix[i][j] = matrix[j][i] = 1, making it symmetric.*

Q2) What is the time complexity of BFS traversal on a graph with V vertices and E edges represented using an adjacency list?

Correct Answer: c) O(V + E)

> *BFS visits each vertex once (O(V)) and explores each edge once in undirected graphs or exactly once in directed graphs (O(E)). Total: O(V + E).*

Q3) For the adjacency matrix of an undirected graph, what can be said about the diagonal elements?

Correct Answer: a) They are always 0 unless there are self-loops

> *Diagonal elements matrix[i][i] represent edges from vertex i to itself. In graphs without self-loops, these are 0. If self-loops exist, they would be 1 (or the weight of the self-loop).*

Q4) In DFS traversal of a graph, which data structure is implicitly used?

Correct Answer: b) Stack

> *DFS uses a stack either explicitly (iterative implementation) or implicitly through recursion (call stack). It explores as deep as possible before backtracking.*

Q5) Which of the following statements about Dijkstra's algorithm are TRUE?

Correct Answers: B, C, D

> ***A) FALSE:*** *Dijkstra's algorithm fails with negative weights as it assumes once a vertex is finalized, no shorter path exists.*
>
> ***B) TRUE:*** *Dijkstra finds shortest paths from source to all reachable vertices.*
>
> ***C) TRUE:*** *A min-heap/priority queue efficiently extracts the vertex with minimum distance.*
>
> ***D) TRUE:*** *With binary heap: O(V) extract-min operations at O(log V) each, and O(E) decrease-key operations at O(log V) each = O((V + E) log V) = O(E log V) for connected graphs.*
>
> ***E) FALSE:*** *Dijkstra cannot detect negative cycles; Bellman-Ford is needed for that.*

Q6) What is the maximum number of edges in a complete undirected graph with n vertices?

Correct Answer: c) n(n-1)/2

> *In a complete undirected graph, every vertex connects to every other vertex. Using combination formula: C(n,2) = n!/(2!(n-2)!) = n(n-1)/2.*

Q7) Which algorithm(s) can be used to find a Minimum Spanning Tree?

Correct Answers: A, B

> ***A) TRUE:*** *Kruskal's algorithm sorts edges and adds them if they don't create cycles.*
>
> ***B) TRUE:*** *Prim's algorithm grows the MST from a starting vertex.*
>
> ***C) FALSE:*** *Dijkstra finds shortest paths, not MST.*
>
> ***D) FALSE:*** *Floyd-Warshall finds all-pairs shortest paths.*
>
> ***E) FALSE:*** *Bellman-Ford finds shortest paths with negative weights.*

Q8) In BFS traversal, nodes are visited in what order relative to the source?

Correct Answer: c) Increasing order of their distance from source

> *BFS explores level by level, visiting all vertices at distance k before any vertex at distance k+1 from the source.*

Q9) A Minimum Spanning Tree of a graph with V vertices must have how many edges?

Correct Answer: b) V - 1

> *A spanning tree connects all V vertices with the minimum number of edges without cycles. A tree with V vertices always has exactly V-1 edges.*

Q10) Which property must a graph satisfy for Kruskal's algorithm to work?

Correct Answer: b) Graph must be connected

> *For an MST to exist, the graph must be connected (or we find a minimum spanning forest). The graph can be directed/undirected, have any weights (positive/negative), and edges need not be unique. However, for practical MST, we typically work with connected, undirected, weighted graphs.*

Section B: Short Answer Questions - Solutions

Q11) Graph representation:

a) Adjacency Matrix:

	A	B	C	D	E
A	0	4	2	0	0
B	4	0	1	5	0
C	2	1	0	8	10
D	0	5	8	0	2
E	0	0	10	2	0

b) Adjacency List:

```
A: [(B, 4), (C, 2)]
B: [(A, 4), (C, 1), (D, 5)]
C: [(A, 2), (B, 1), (D, 8), (E, 10)]
D: [(B, 5), (C, 8), (E, 2)]
E: [(C, 10), (D, 2)]
```

c) Difference if directed:

> *If the graph were directed (from first to second vertex): • The adjacency matrix would NOT be symmetric • Each edge would appear only once in the adjacency list*
>
> *Example: Edge A B (weight 4) would mean: • Matrix: matrix[A][B] = 4, but matrix[B][A] = 0 • List: A would have (B,4) but B would NOT have (A,4)*

Q12) Adjacency matrix analysis:

a) Graph drawing:

> *Edges: A-B, A-C, B-D, C-D (forming a square/cycle)*

b) Directed or Undirected:

> *The graph is **UNDIRECTED**. We can tell because the matrix is symmetric: • matrix[A][B] = matrix[B][A] = 1 • matrix[A][C] = matrix[C][A] = 1 • matrix[B][D] = matrix[D][B] = 1 • matrix[C][D] = matrix[D][C] = 1*

c) Degree of each vertex:

> *Degree = sum of row (or column, since symmetric) • deg(A) = 1 + 1 = 2 • deg(B) = 1 + 1 = 2 • deg(C) = 1 + 1 = 2 • deg(D) = 1 + 1 = 2*

Q13) BFS and DFS traversals:

a) BFS starting from A:

Order: A B C D E F

> *Level 0: A*
> *Level 1: B, C (neighbors of A)*
> *Level 2: D, E (from B), F (from C)*
> *F is already discovered via C, so not repeated*

b) DFS starting from A:

Order: A B D E F C

> *A B (first neighbor of A)*
> *B D (first unvisited neighbor of B)*
> *D has no unvisited neighbors, backtrack*
> *B E (next unvisited neighbor of B)*

E F (only unvisited neighbor of E)
Backtrack to A C (last unvisited neighbor of A)

Q14) Dijkstra's algorithm:

a) Order of finalized vertices:

S A B C D T

Step	Finalized	Distances [S,A,B,C,D,T]
0	-	[0, ∞, ∞, ∞, ∞, ∞]
1	S	[0, 2, 4, ∞, ∞, ∞]
2	A	[0, 2, 3, 9, ∞, ∞]
3	B	[0, 2, 3, 6, 8, ∞]
4	C	[0, 2, 3, 6, 8, 8]
5	D	[0, 2, 3, 6, 8, 8]
6	T	[0, 2, 3, 6, 8, 8]

b) Shortest distances from S:

S: 0, A: 2, B: 3, C: 6, D: 8, T: 8

c) Shortest path S to T:

Path: S A B C T
Total distance: 2 + 1 + 3 + 2 = 8

Alternative path with same length: S A B D T
Total distance: 2 + 1 + 5 + 1 = 9 (longer, so not chosen)

Q15) Kruskal's algorithm for MST:

a) Edges considered in order (sorted by weight):

1. B-C: 2
2. A-B: 3
3. D-E: 3
4. B-D: 4
5. E-F: 4
6. A-C: 5
7. C-D: 6
8. D-F: 7
9. C-E: 8

b) Edges included in MST:

Edge	Weight	Action
B-C	2	Add (connects B, C)
A-B	3	Add (connects A)
D-E	3	Add (connects D, E)
B-D	4	Add (connects component ABC with DE)
E-F	4	Add (connects F)
A-C	5	Skip (creates cycle in ABC)
C-D	6	Skip (already connected)

MST Edges: B-C, A-B, D-E, B-D, E-F

c) Total weight of MST:

> *Total = 2 + 3 + 3 + 4 + 4 =* ***16***

Section C: Long Answer Questions - Solutions

Q16) True or False with justification:

1. TRUE - Sum of degrees equals 2E (each edge contributes 2 to the total degree sum).

2. FALSE - BFS finds shortest path only in *unweighted* graphs. For weighted graphs, use Dijkstra's or Bellman-Ford.

3. TRUE - If all edge weights are distinct, no two spanning trees can have the same total weight. The unique minimum weight determines a unique MST.

4. TRUE - DFS can detect cycles: if we encounter an already visited vertex (that's not the parent), a cycle exists.

5. FALSE - Dijkstra assumes all edges are non-negative. Negative weights can lead to incorrect results because the algorithm finalizes vertices assuming no shorter path exists.

6. FALSE - The graph must also be *connected* to be a tree. Example: Two separate edges (4 vertices, 2 edges forming two components) is not a tree.

7. TRUE - Prim's with binary heap: O(V) extract-min at O(log V) + O(E) decrease-key at O(log V) = O(E log V).

8. FALSE - BFS typically uses more memory because it stores all vertices at current level in the queue, which can be O(V) in worst case. DFS uses O(h) where h is height/depth.

9. TRUE - Every connected graph has at least one spanning tree (can be constructed using DFS or BFS).

10. FALSE - In adjacency list, checking if edge (u,v) exists requires traversing u's adjacency list, which takes O(degree(u)) time, or O(V) in worst case.

Q17) Connected Components Algorithm:

Algorithm:

```
function countConnectedComponents(n, edges):
    # Build adjacency list
    graph = [[] for _ in range(n)]
    for (u, v) in edges:
        graph[u].append(v)
        graph[v].append(u)

    visited = [False] * n
    count = 0

    # Run DFS/BFS from each unvisited vertex
    for vertex in range(n):
        if not visited[vertex]:
            DFS(vertex, graph, visited)
            count += 1

    return count

function DFS(vertex, graph, visited):
    visited[vertex] = True
    for neighbor in graph[vertex]:
        if not visited[neighbor]:
            DFS(neighbor, graph, visited)
```

Explanation:

> *1. Build adjacency list representation: O(E)*
> *2. Initialize visited array: O(V)*
> *3. For each unvisited vertex, perform DFS/BFS to mark all vertices in that component*
> *4. Each DFS call marks one connected component*
> *5. Count increments for each new component discovered*
>
> ***Time Complexity: O(V + E)***
> *• We visit each vertex once: O(V)*
> *• We examine each edge once: O(E)*
> *• Total: O(V + E)*

Space Complexity: O(V + E)
• *Adjacency list: O(V + E)*
• *Visited array: O(V)*
• *Recursion stack (worst case): O(V)*

Q18) Path Existence Algorithm:

Algorithm: Use BFS or DFS

```
function pathExists(graph, source, destination):
    if source == destination:
        return True

    visited = set()
    queue = [source]  # For BFS
    visited.add(source)

    while queue:
        current = queue.pop(0)

        if current == destination:
            return True

        for neighbor in graph[current]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

    return False
```

Why BFS/DFS?

Either BFS or DFS works equally well:

BFS Advantages:
• *Finds shortest path (in terms of edges) if needed*
• *Better for finding nearby destinations*

DFS Advantages:

• Uses less memory (O(h) vs O(w) where h=height, w=width)

• Easier to implement recursively

Time Complexity: O(V + E)

• We may need to explore all vertices and edges in worst case

Space Complexity:

• BFS: O(V) for queue

• DFS: O(h) for recursion stack, where h is maximum depth

Q19) Comparison Questions:

a) Adjacency Matrix vs Adjacency List:

Aspect	Adjacency Matrix	Adjacency List
Space Complexity	$O(V^2)$	O(V + E)
Add Edge	O(1)	O(1)
Remove Edge	O(1)	O(V) worst case
Check if edge exists	O(1)	O(degree) = O(V) worst
Find all neighbors	O(V)	O(degree)

When to use Adjacency Matrix:

• Dense graphs ($E \approx V^2$)

• Need O(1) edge existence checks

• Fixed size graph

• Memory is not a constraint

When to use Adjacency List:

• Sparse graphs ($E << V^2$)

• Need to iterate over neighbors efficiently

• Dynamic graphs (adding/removing vertices)

• Memory is limited

b) BFS vs DFS:

Aspect	BFS	DFS
Data Structure	Queue	Stack (or recursion)
Traversal Order	Level by level	Depth-first, then backtrack
Space Complexity	O(w) w=max width	O(h) h=max depth
Shortest Path	Yes (unweighted)	No
Memory Usage	Higher (wide graphs)	Lower (deep graphs)

> ***BFS Applications:***
> • *Finding shortest path in unweighted graphs*
> • *Level-order traversal*
> • *Finding all nodes within k distance*
> • *Testing bipartiteness*
>
> ***DFS Applications:***
> • *Topological sorting*
> • *Cycle detection*
> • *Finding connected components*
> • *Path finding (any path, not shortest)*
> • *Solving mazes/puzzles*
>
> ***When to prefer BFS:***
> • *Need shortest path*
> • *Target is likely near source*
> • *Graph is wide and shallow*
>
> ***When to prefer DFS:***
> • *Don't need shortest path*
> • *Graph is deep and narrow*
> • *Want to use less memory*
> • *Need to explore all paths*

Q20) Prim's Algorithm Step-by-Step:

Given Graph Edges: A-B:7, A-D:5, B-C:8, B-D:9, B-E:7, C-E:5, D-E:15, D-F:6, E-F:8, E-G:9, F-G:11

Step 1: Start from A
MST = {}, Priority Queue = [(B,7), (D,5)]
Choose D (minimum weight 5)
Add edge: A-D (weight 5)

Step 2: From D, add neighbors
MST = {A-D}, PQ = [(B,7), (B,9), (E,15), (F,6)]
Choose F (minimum weight 6)
Add edge: D-F (weight 6)

Step 3: From F, add neighbors
MST = {A-D, D-F}, PQ = [(B,7), (B,9), (E,15), (E,8), (G,11)]
Choose B (minimum weight 7)
Add edge: A-B (weight 7)

Step 4: From B, add neighbors
MST = {A-D, D-F, A-B}, PQ = [(B,9), (E,15), (E,8), (G,11), (C,8), (E,7)]
Choose E via B (minimum weight 7)
Add edge: B-E (weight 7)

Step 5: From E, add neighbors
MST = {A-D, D-F, A-B, B-E}, PQ = [(B,9), (E,15), (E,8), (G,11), (C,8), (C,5), (F,8), (G,9)]
Choose C (minimum weight 5)
Add edge: C-E (weight 5)

Step 6: From C, no new neighbors
MST = {A-D, D-F, A-B, B-E, C-E}, PQ = [(B,9), (E,15), (E,8), (G,11), (F,8), (G,9)]
Choose G via E (minimum weight 9)
Add edge: E-G (weight 9)

Final MST Edges:

> *1. A-D (5)*
> *2. D-F (6)*
> *3. A-B (7)*
> *4. B-E (7)*

> *5. C-E (5)*
> *6. E-G (9)*
>
> ***Total Weight: 5 + 6 + 7 + 7 + 5 + 9 = 39***
>
> *The MST connects all 7 vertices (A, B, C, D, E, F, G) with 6 edges.*

End of Solutions