

Assignment 3 - Set C: Theoretical Questions and Answers

Graph Theory - Data Structures and Representations

Question 1

Which data structure is used to implement Depth First Search?

Answer

Stack data structure is used to implement Depth First Search (DFS).

Explanation

DFS can be implemented in two ways:

- 1. Recursive approach** - Uses implicit call stack
- 2. Iterative approach** - Uses explicit stack data structure

Why Stack?

- Stack follows Last-In-First-Out (LIFO) principle
- DFS explores as deep as possible along each branch before backtracking
- When a vertex is visited, its adjacent unvisited vertices are pushed onto stack
- We process the most recently discovered vertex first (LIFO behavior)

Recursive DFS (Using Implicit Stack)

```
void DFS(int vertex, int visited[]) {
    visited[vertex] = 1;
    printf("%d ", vertex);

    for each adjacent vertex v of vertex {
        if (!visited[v]) {
            DFS(v, visited); // Recursive call uses call stack
        }
    }
}
```

Iterative DFS (Using Explicit Stack)

```
void DFS(int start) {  
    Stack s;  
    int visited[MAX] = {0};  
  
    push(&s, start);  
  
    while (!isEmpty(&s)) {  
        int vertex = pop(&s);  
  
        if (!visited[vertex]) {  
            visited[vertex] = 1;  
            printf("%d ", vertex);  
  
            for each adjacent vertex v of vertex {  
                if (!visited[v]) {  
                    push(&s, v);  
                }  
            }  
        }  
    }  
}
```

Example

Graph: 0 -- 1
| |
2 -- 3

DFS from 0: 0 → 2 → 3 → 1

Stack Operations:

```
Initial: [0]
Pop 0, push 2,1 → [2, 1]
Pop 2, push 3 → [3, 1]
Pop 3 → [1]
Pop 1 → []
```

Time Complexity: $O(V + E)$ where V = vertices, E = edges

Space Complexity: $O(V)$ for stack

Question 2

Where is the new node appended in Breadth-first search of OPEN list?

Answer

In Breadth-First Search (BFS), new nodes are appended at the **REAR (end)** of the OPEN list.

Explanation

The OPEN list in BFS is implemented as a **Queue**, which follows FIFO (First-In-First-Out) principle.

Queue Operations in BFS

1. **ENQUEUE (Insert at rear)**: Add newly discovered nodes
2. **DEQUEUE (Remove from front)**: Process nodes in order of discovery

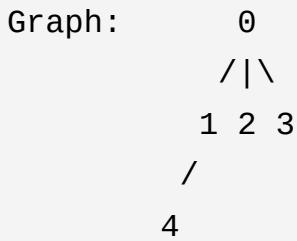
BFS Algorithm with OPEN List

1. Start with source vertex in OPEN list
2. While OPEN list is not empty:
 - Remove vertex from FRONT of OPEN list
 - Process/visit this vertex
 - Add all unvisited adjacent vertices to REAR of OPEN list

Why Rear?

- Ensures level-by-level traversal
- Nodes discovered first are processed first
- Maintains the breadth-first property
- All nodes at level L are processed before level L+1

Example



BFS from 0:

Step-by-step:

Step 1: OPEN = [0]
 Step 2: Remove 0 from front, add 1,2,3 to rear → OPEN = [1,2,3]
 Step 3: Remove 1 from front, add 4 to rear → OPEN = [2,3,4]
 Step 4: Remove 2 from front → OPEN = [3,4]
 Step 5: Remove 3 from front → OPEN = [4]
 Step 6: Remove 4 from front → OPEN = []

BFS Order: 0 1 2 3 4

Summary

- **New nodes:** Added at REAR (enqueue)
- **Processed nodes:** Removed from FRONT (dequeue)
- **Data Structure:** Queue
- **Property:** FIFO ensures level-order traversal

Question 3

What is a complete graph?

Answer

A **Complete Graph** is a graph in which every pair of distinct vertices is connected by a unique edge.

Definition

A complete graph with n vertices has exactly $\frac{n(n-1)}{2}$ edges (for undirected graph) and $n(n-1)$ edges (for directed graph).

Notation: K_n (where n = number of vertices)

Properties

1. Every vertex is connected to every other vertex
2. Maximum number of edges possible for n vertices
3. Degree of each vertex = $n-1$

4. It is a regular graph
5. It is always connected
6. Diameter = 1 (shortest path between any two vertices is 1)

Examples

K_3 (Complete graph with 3 vertices)



Edges: (0,1), (0,2), (1,2)

$$\text{Total edges} = 3(3-1)/2 = 3$$

K_4 (Complete graph with 4 vertices)



Edges: (0,1), (0,2), (0,3), (1,2), (1,3), (2,3)

Total edges = $4(4-1)/2 = 6$

K₅ (Complete graph with 5 vertices)

Total edges = $5(5-1)/2 = 10$

Formula for Edges

- **Undirected:** $E = n(n-1)/2$
- **Directed:** $E = n(n-1)$

Where:

- n = number of vertices
- E = number of edges

Applications

1. Network topology where every node communicates with every other node
2. Modeling fully connected systems
3. Clique problems in graph theory
4. Tournament scheduling
5. Social networks (everyone knows everyone)

Key Difference

- **Complete Graph:** ALL possible edges exist
 - **Connected Graph:** At least one path exists between any two vertices (may not have all edges)
-

Question 4

Which data structure is used to implement adjacency list method?

Answer

Linked List data structure is used to implement the adjacency list method.

Structure

An adjacency list consists of:

1. An array of pointers/heads (size = number of vertices)
2. Each array element points to a linked list of adjacent vertices

Implementation

```

typedef struct Node {
    int vertex;          // Destination vertex
    struct Node* next;  // Pointer to next adjacent vertex
} Node;

// Array of linked list heads
Node* adjacencyList[MAX_VERTICES];

```

Why Linked List?

1. **Dynamic size** - Can add edges without knowing total count beforehand
2. **Space efficient** - Only stores existing edges
3. **Easy insertion/deletion** of edges
4. **Efficient traversal** of neighbors

Structure Example

Graph:

```

0 -- 1
|   |
2 -- 3

```

Adjacency List:

```
0: [1] -> [2] -> NULL
```

```

1: [0] -> [3] -> NULL
2: [0] -> [3] -> NULL
3: [1] -> [2] -> NULL

```

Memory Representation

Index	Linked List
0	$\rightarrow [1] \rightarrow [2] \rightarrow \text{NULL}$
1	$\rightarrow [0] \rightarrow [3] \rightarrow \text{NULL}$
2	$\rightarrow [0] \rightarrow [3] \rightarrow \text{NULL}$
3	$\rightarrow [1] \rightarrow [2] \rightarrow \text{NULL}$

Operations

Operation	Time Complexity
Add Edge	$O(1)$
Check Edge	$O(\text{degree})$
Remove Edge	$O(\text{degree})$
Space	$O(V + E)$

Where V = vertices, E = edges

Advantages

- Space efficient for sparse graphs
- Fast to iterate over neighbors
- Easy to add/remove edges
- Supports weighted graphs (add weight field to node)

Variations

1. Singly Linked List (most common)
2. Doubly Linked List (for easy deletion)
3. Sorted Linked List (for ordered traversal)
4. Array of vectors/lists (in C++)

Comparison with Adjacency Matrix

Aspect	Adjacency List	Adjacency Matrix
Space Complexity	$O(V + E)$	$O(V^2)$
Edge Check Time	$O(\text{degree})$	$O(1)$
Best for	Sparse graphs	Dense graphs

Best for: Sparse graphs where $E \ll V^2$

Question 5

Compare adjacency matrix and adjacency list.

Answer

Adjacency Matrix vs Adjacency List

1. Representation

Adjacency Matrix:

- 2D array of size $V \times V$
- $\text{Matrix}[i][j] = 1$ if edge exists from i to j , else 0
- For weighted graphs: $\text{Matrix}[i][j] = \text{weight}$

Adjacency List:

- Array of linked lists of size V
- Each index contains list of adjacent vertices
- For weighted graphs: Store (vertex, weight) in nodes

2. Space Complexity

Method	Space	Best for
Adjacency Matrix	$O(V^2)$	Dense graphs
Adjacency List	$O(V + E)$	Sparse graphs

3. Time Complexity

Operation	Adjacency Matrix	Adjacency List
Add Edge	$O(1)$	$O(1)$
Remove Edge	$O(1)$	$O(\text{degree})$
Check Edge (i,j)	$O(1)$	$O(\text{degree})$
Get All Neighbors	$O(V)$	$O(\text{degree})$

4. Advantages

Adjacency Matrix ✓

- Edge lookup is $O(1)$
- Simple to implement
- Easy to check if edge exists
- Good for dense graphs ($E \approx V^2$)
- Better for adjacency queries
- Can represent multi-graphs easily

Adjacency List ✓

- Space efficient for sparse graphs
- Fast to iterate over neighbors
- Easy to add edges dynamically
- Good for most graph algorithms (BFS, DFS)
- Less memory for sparse graphs
- Faster neighbor traversal

5. Disadvantages

Adjacency Matrix ✗

- Wastes space for sparse graphs
- Takes $O(V^2)$ space even with few edges
- Traversing all neighbors takes $O(V)$ time

Adjacency List ✗

- Edge lookup takes $O(\text{degree})$ time

- More complex implementation
- Extra space for pointers

6. When to Use

Use Adjacency Matrix when:

- Graph is dense ($E \approx V^2$)
- Need frequent edge existence queries
- Need quick access to any edge
- Working with small graphs

Use Adjacency List when:

- Graph is sparse ($E \ll V^2$)
- Need to frequently traverse neighbors
- Memory is a concern
- Graph size is large
- Most real-world graphs (social networks, roads, etc.)

7. Example Comparison

Graph: 0-1, 1-2 (3 vertices, 2 edges)

Adjacency Matrix:

	0	1	2
0	0	1	0
1	1	0	0
2	0	0	0

```
0[0 1 0]
1[1 0 1]
2[0 1 0]
```

Space: 9 units

Adjacency List:

```
0: [1]
1: [0, 2]
2: [1]
```

Space: 4 units (nodes)

Conclusion

- **Adjacency List** is preferred for most applications due to space efficiency
- **Adjacency Matrix** is useful when graph is dense or edge queries are frequent