Graphs

Indtroduction

Creating a presentation on Graph Data Structures with C can be highly informative and engaging if structured properly. Here's a comprehensive list of topics you should consider including, along with brief explanations to guide your content:

1. **Introduction to Graphs**
   * Definition: Explain what a graph is, including key terms like vertices (nodes) and edges (links).
   * Types of Graphs: Differentiate between undirected and directed graphs.

**2. Graph Representation**

* Adjacency Matrix
* Adjacency List
* Comparison of adjacency matrix and adjacency list

When discussing types of graphs in C data structures, it is essential to categorize them based on various criteria such as directionality, weight, and structure. Here’s an overview of the different types of graphs you should cover:

**1. Based on Directionality**

* **Undirected Graph**
  + Edges have no direction. If there is an edge between vertices A and B, you can traverse from A to B and from B to A.
  + Example: Social networks where friendships are mutual.
* **Directed Graph (Digraph)**
  + Edges have a direction. If there is an edge from vertex A to vertex B, you can traverse from A to B, but not necessarily from B to A.
  + Example: Twitter following where one user can follow another without reciprocation.

**2. Based on Weight**

* **Unweighted Graph**
  + All edges are equal; there are no weights associated with the edges.
  + Example: Simple social networks where all connections are treated equally.
* **Weighted Graph**
  + Each edge has a weight or cost associated with it, representing the cost to traverse that edge.
  + Example: Road networks where edges represent roads and weights represent distances or travel times.

**3. Based on Cyclicity**

* **Acyclic Graph**
  + A graph without cycles. If directed, it's called a Directed Acyclic Graph (DAG).
  + Example: Task scheduling where tasks must be completed in a certain order.
* **Cyclic Graph**
  + A graph that contains at least one cycle.
  + Example: Feedback systems in electrical circuits.

**4. Based on Connectivity**

* **Connected Graph**
  + There is a path between any two vertices. In

an undirected graph, this means every vertex is reachable from any other vertex. - Example: A single connected component of a network, like a local area network.

* **Disconnected Graph**
  + The graph consists of two or more components, where there is no path between at least one pair of vertices.
  + Example: A network of isolated subnetworks.

**5. Based on Completeness**

* **Complete Graph**
  + Every pair of vertices is connected by a unique edge. For an undirected graph with nnn vertices, there are n(n−1)2\frac{n(n-1)}{2}2n(n−1)​ edges.
  + Example: A fully connected peer-to-peer network.
* **Incomplete Graph**
  + Not all pairs of vertices are connected by an edge.
  + Example: Most real-world networks, like social networks or road maps.

**6. Based on Bipartiteness**

* **Bipartite Graph**
  + The vertex set can be divided into two disjoint sets such that no two graph vertices within the same set are adjacent.
  + Example: Job assignment problems where jobs and workers are in separate sets.
* **Non-Bipartite Graph**
  + The vertex set cannot be divided into two disjoint sets as defined above.
  + Example: Most social networks.

**7. Special Graphs**

* **Tree**
  + An acyclic connected graph. Trees are a subset of graphs and an essential data structure in computer science.
  + Example: File system hierarchy.
* **Forest**
  + A collection of disjoint trees.
  + Example: Multiple disconnected hierarchies within an organization.
* **Subgraph**
  + A graph formed from a subset of the vertices and edges of a larger graph.
  + Example: A specific module within a larger software dependency graph.
* **Multigraph**
  + A graph where multiple edges (parallel edges) between the same set of vertices are allowed.
  + Example: Transport networks with multiple routes between two cities.

**8. Planar Graph**

* A graph that can be drawn on a plane without any edges crossing.
* Example: Circuit layouts on a PCB (Printed Circuit Board).

**Implementation Considerations in C**

For each type of graph, you may want to show how they can be represented in C using adjacency matrices, adjacency lists, or other data structures. For example:

* **Adjacency Matrix**: Suitable for dense graphs.
* **Adjacency List**: More space-efficient for sparse graphs.

Here are simple representations in C:

{

These examples provide a foundation to demonstrate how different types of graphs can be represented and manipulated in C. You can expand on these with more complex operations and algorithms as needed for your presentation.

Top of Form

Bottom of Fo

**3. Graph Traversal Algorithms**

* Comparison of DFS and BFSGraph and tree traversal algorithms are essential for exploring the nodes and edges of these data structures systematically. In C, the most common traversal algorithms for graphs and trees include Depth-First Search (DFS) and Breadth-First Search (BFS). Additionally, for trees specifically, you have various depth-first traversals like Preorder, Inorder, and Postorder. Here’s a detailed explanation of each with implementations in C:
* Depth-First Search (DFS)
* Breadth-First Search (BFS)

### 1. ****Depth-First Search (DFS)****

#### For Graphs:

DFS explores as far as possible along each branch before backtracking. It can be implemented using recursion (implicit stack) or an explicit stack.

#### Graph Representation Using Adjacency List and DFS Implementation

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

typedef struct Node {

int vertex;

struct Node\* next;

} Node;

typedef struct Graph {

int numVertices;

Node\*\* adjLists;

int\* visited;

} Graph;

// Function to create a node

Node\* createNode(int v) {

Node\* newNode = malloc(sizeof(Node));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

// Function to create a graph

Graph\* createGraph(int vertices) {

Graph\* graph = malloc(sizeof(Graph));

graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(Node\*));

graph->visited = malloc(vertices \* sizeof(int));

for (int i = 0; i < vertices; i++) {

graph->adjLists[i] = NULL;

graph->visited[i] = 0;

}

return graph;

}

// Function to add edge

void addEdge(Graph\* graph, int src, int dest) {

// Add edge from src to dest

Node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

// Add edge from dest to src (for undirected graph)

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

// Recursive function for DFS

void DFS(Graph\* graph, int vertex) {

Node\* adjList = graph->adjLists[vertex];

Node\* temp = adjList;

graph->visited[vertex] = 1;

printf("Visited %d\n", vertex);

while (temp != NULL) {

int connectedVertex = temp->vertex;

if (graph->visited[connectedVertex] == 0) {

DFS(graph, connectedVertex);

}

temp = temp->next;

}

}

int main() {

int vertices = 5;

Graph\* graph = createGraph(vertices);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 1, 3);

addEdge(graph, 2, 4);

printf("Depth First Search starting from vertex 0:\n");

DFS(graph, 0);

return 0;

}

### Explanation

1. **Graph Creation**:
   * The createGraph function initializes the graph with the given number of vertices. It creates an adjacency list for each vertex and initializes the visited array to track visited nodes.
2. **Adding Edges**:
   * The addEdge function adds edges to the graph. Since it’s an undirected graph, edges are added in both directions (src to dest and dest to src).
3. **DFS Function**:
   * The DFS function recursively visits nodes. It marks the current node as visited, prints it, and recursively visits all its unvisited adjacent nodes.
4. **Main Function**:
   * The main function sets up the graph with 5 vertices, adds edges, and starts the DFS traversal from vertex 0.

### DFS for Trees

For trees, DFS can be performed using Preorder, Inorder, or Postorder traversal. Here’s an example of Preorder traversal (Root, Left, Right):

#### Tree Representation and Preorder DFS Implementation

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* left;

struct Node\* right;

} Node;

// Function to create a new node

Node\* createNode(int data) {

Node\* newNode = (Node\*) malloc(sizeof(Node));

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function for Preorder DFS

void preorderTraversal(Node\* node) {

if (node == NULL) return;

printf("%d ", node->data); // Visit the root

preorderTraversal(node->left); // Traverse the left subtree

preorderTraversal(node->right); // Traverse the right subtree

}

int main() {

Node\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

root->left->left = createNode(4);

root->left->right = createNode(5);

printf("Preorder traversal of the binary tree:\n");

preorderTraversal(root);

return 0;

}

### Explanation

1. **Tree Node Creation**:
   * The createNode function initializes a tree node with the given data and sets its left and right pointers to NULL.
2. **Preorder Traversal Function**:
   * The preorderTraversal function performs a Preorder DFS traversal. It visits the root, then recursively traverses the left subtree, followed by the right subtree.
3. **Main Function**:
   * The main function sets up a simple binary tree and calls the preorderTraversal function to traverse the tree starting from the root.

By understanding these examples, you can implement DFS for both graphs and trees in C,

### 2. ****Breadth-First Search (BFS)****

#### BFS explores all neighbors at the present depth level before moving on to nodes at the next depth level. It uses a queue to keep track of the next location to visit. Graph Representation Using Adjacency List and BFS Implementation

#### Graph Representation Using Adjacency List and BFS Implementation

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

typedef struct Node {

int vertex;

struct Node\* next;

} Node;

typedef struct Graph {

int numVertices;

Node\*\* adjLists;

int\* visited;

} Graph;

typedef struct Queue {

int items[MAX];

int front;

int rear;

} Queue;

// Function to create a node

Node\* createNode(int v) {

Node\* newNode = malloc(sizeof(Node));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

// Function to create a graph

Graph\* createGraph(int vertices) {

Graph\* graph = malloc(sizeof(Graph));

graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(Node\*));

graph->visited = malloc(vertices \* sizeof(int));

for (int i = 0; i < vertices; i++) {

graph->adjLists[i] = NULL;

graph->visited[i] = 0;

}

return graph;

}

// Function to add edge

void addEdge(Graph\* graph, int src, int dest) {

// Add edge from src to dest

Node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

// Add edge from dest to src (for undirected graph)

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

// Function to create a queue

Queue\* createQueue() {

Queue\* q = malloc(sizeof(Queue));

q->front = -1;

q->rear = -1;

return q;

}

// Check if the queue is empty

int isEmpty(Queue\* q) {

if (q->rear == -1)

return 1;

else

return 0;

}

// Enqueue an element

void enqueue(Queue\* q, int value) {

if (q->rear == MAX - 1)

printf("\nQueue is Full!!");

else {

if (q->front == -1)

q->front = 0;

q->rear++;

q->items[q->rear] = value;

}

}

// Dequeue an element

int dequeue(Queue\* q) {

int item;

if (isEmpty(q)) {

printf("Queue is empty");

item = -1;

} else {

item = q->items[q->front];

q->front++;

if (q->front > q->rear) {

q->front = q->rear = -1;

}

}

return item;

}

// BFS algorithm

void BFS(Graph\* graph, int startVertex) {

Queue\* q = createQueue();

graph->visited[startVertex] = 1;

enqueue(q, startVertex);

while (!isEmpty(q)) {

int currentVertex = dequeue(q);

printf("Visited %d\n", currentVertex);

Node\* temp = graph->adjLists[currentVertex];

while (temp) {

int adjVertex = temp->vertex;

if (graph->visited[adjVertex] == 0) {

graph->visited[adjVertex] = 1;

enqueue(q, adjVertex);

}

temp = temp->next;

}

}

}

int main() {

int vertices = 6;

Graph\* graph = createGraph(vertices);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 1, 3);

addEdge(graph, 2, 4);

addEdge(graph, 3, 4);

addEdge(graph, 3, 5);

printf("Breadth First Search starting from vertex 0:\n");

BFS(graph, 0);

return 0;

}

**Explanation**  **Graph Creation**:

* The createGraph function initializes a graph with the given number of vertices. It creates an adjacency list for each vertex and initializes the visited array to track visited nodes.

 **Adding Edges**:

* The addEdge function adds edges to the graph. Since it’s an undirected graph, edges are added in both directions (src to dest and dest to src).

 **Queue Operations**:

* The createQueue function initializes a queue.
* The isEmpty function checks if the queue is empty.
* The enqueue function adds an element to the queue.
* The dequeue function removes an element from the queue.

 **BFS Function**:

* The BFS function performs a Breadth-First Search starting from a given vertex. It uses a queue to manage the order of nodes to be visited. It marks the current node as visited, prints it, and enqueues all its unvisited adjacent nodes.

 **Main Function**:

* The main function sets up the graph with 6 vertices, adds edges, and starts the BFS traversal from vertex 0.

**4. Shortest Path Algorithms**

* Dijkstra’s Algorithm
* Bellman-Ford Algorithm
* Floyd-Warshall Algorithm

### C implementation Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path from a single source node to all other nodes in a graph with non-negative edge weights. Here’s a C implementation using an adjacency matrix:

#### Dijkstra's Algorithm Implementation

#include <stdio.h>

#include <limits.h>

#define V 9 // Number of vertices in the graph

int minDistance(int dist[], int sptSet[]) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

if (sptSet[v] == 0 && dist[v] <= min)

min = dist[v], min\_index = v;

return min\_index;

}

void printSolution(int dist[]) {

printf("Vertex \t\t Distance from Source\n");

for (int i = 0; i < V; i++)

printf("%d \t\t %d\n", i, dist[i]);

}

void dijkstra(int graph[V][V], int src) {

int dist[V];

int sptSet[V];

for (int i = 0; i < V; i++)

dist[i] = INT\_MAX, sptSet[i] = 0;

dist[src] = 0;

for (int count = 0; count < V - 1; count++) {

int u = minDistance(dist, sptSet);

sptSet[u] = 1;

for (int v = 0; v < V; v++)

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v])

dist[v] = dist[u] + graph[u][v];

}

printSolution(dist);

}

int main() {

int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},

{4, 0, 8, 0, 0, 0, 0, 11, 0},

{0, 8, 0, 7, 0, 4, 0, 0, 2},

{0, 0, 7, 0, 9, 14, 0, 0, 0},

{0, 0, 0, 9, 0, 10, 0, 0, 0},

{0, 0, 4, 14, 10, 0, 2, 0, 0},

{0, 0, 0, 0, 0, 2, 0, 1, 6},

{8, 11, 0, 0, 0, 0, 1, 0, 7},

{0, 0, 2, 0, 0, 0, 6, 7, 0}};

dijkstra(graph, 0);

return 0;

}

### Explanation

1. **Graph Representation**:
   * The graph is represented using an adjacency matrix where graph[i][j] holds the weight of the edge from vertex i to vertex j.
2. **Utility Functions**:
   * minDistance: Finds the vertex with the minimum distance value from the set of vertices not yet included in the shortest path tree.
   * printSolution: Prints the calculated shortest distances from the source to each vertex.
3. **Dijkstra's Algorithm**:
   * Initializes distance values and the shortest path tree set.
   * Iteratively finds the vertex with the minimum distance, updates distance values of its adjacent vertices, and includes it in the shortest path tree.

### Bellman-Ford Algorithm

The Bellman-Ford algorithm finds the shortest paths from a single source vertex to all other vertices in a graph. It works even with graphs that have negative edge weights. Here’s a C implementation using an edge list:

#### Bellman-Ford Algorithm Implementation

#include <stdio.h>

#include <stdlib.h>

#define INF 10000

#define V 5

#define E 8

typedef struct Edge {

int src, dest, weight;

} Edge;

typedef struct Graph {

int V, E;

Edge\* edge;

} Graph;

Graph\* createGraph(int V, int E) {

Graph\* graph = (Graph\*)malloc(sizeof(Graph));

graph->V = V;

graph->E = E;

graph->edge = (Edge\*)malloc(E \* sizeof(Edge));

return graph;

}

void printArr(int dist[], int n) {

printf("Vertex Distance from Source\n");

for (int i = 0; i < n; ++i)

printf("%d \t\t %d\n", i, dist[i]);

}

void BellmanFord(Graph\* graph, int src) {

int V = graph->V;

int E = graph->E;

int dist[V];

for (int i = 0; i < V; i++)

dist[i] = INF;

dist[src] = 0;

for (int i = 1; i <= V - 1; i++) {

for (int j = 0; j < E; j++) {

int u = graph->edge[j].src;

int v = graph->edge[j].dest;

int weight = graph->edge[j].weight;

if (dist[u] != INF && dist[u] + weight < dist[v])

dist[v] = dist[u] + weight;

}

}

for (int i = 0; i < E; i++) {

int u = graph->edge[i].src;

int v = graph->edge[i].dest;

int weight = graph->edge[i].weight;

if (dist[u] != INF && dist[u] + weight < dist[v])

printf("Graph contains negative weight cycle\n");

}

printArr(dist, V);

}

int main() {

int V = 5;

int E = 8;

Graph\* graph = createGraph(V, E);

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = -1;

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 4;

graph->edge[2].src = 1;

graph->edge[2].dest = 2;

graph->edge[2].weight = 3;

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 2;

graph->edge[4].src = 1;

graph->edge[4].dest = 4;

graph->edge[4].weight = 2;

graph->edge[5].src = 3;

graph->edge[5].dest = 2;

graph->edge[5].weight = 5;

graph->edge[6].src = 3;

graph->edge[6].dest = 1;

graph->edge[6].weight = 1;

graph->edge[7].src = 4;

graph->edge[7].dest = 3;

graph->edge[7].weight = -3;

BellmanFord(graph, 0);

return 0;

}

### Explanation

1. **Graph Creation**:
   * The createGraph function initializes a graph with the given number of vertices and edges. It allocates memory for the edges.
2. **Utility Functions**:
   * printArr: Prints the calculated shortest distances from the source to each vertex.
3. **Bellman-Ford Algorithm**:
   * Initializes distance values.
   * Iteratively relaxes all edges V-1 times.
   * Checks for negative weight cycles by checking if further relaxation is possible.

### Summary

Both Dijkstra’s and Bellman-Ford algorithms are used to find the shortest path in graphs. Dijkstra’s algorithm is efficient for graphs with non-negative weights, while Bellman-Ford is suitable for graphs with negative weights and can detect negative weight cycles. Including these implementations in your presentation will give a comprehensive overview of shortest path algorithms in C.

Top of Form

Bottom of Form

**5. Minimum Spanning Tree Algorithms**

* Kruskal’s Algorithm
* Prim’s Algorithm
* 1. Krushkal,s algorithm
* Kruskal's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a connected, undirected graph. It starts with an empty set of edges and adds the smallest edge that connects two different components of the graph until all vertices are connected. Here’s an implementation of Kruskal’s algorithm in C using the disjoint-set data structure:
* Implementation of krushkals algorithm using c data structures
* Copy code
* #include <stdio.h>
* #include <stdlib.h>
* #define MAX\_EDGES 30
* #define MAX\_VERTICES 30
* // Structure to represent an edge
* typedef struct {
* int src, dest, weight;
* } Edge;
* // Structure to represent a graph
* typedef struct {
* int numVertices, numEdges;
* Edge edges[MAX\_EDGES];
* } Graph;
* // Structure to represent a subset for union-find
* typedef struct {
* int parent;
* int rank;
* } Subset;
* // Create a graph with V vertices and E edges
* Graph\* createGraph(int V, int E) {
* Graph\* graph = (Graph\*)malloc(sizeof(Graph));
* graph->numVertices = V;
* graph->numEdges = E;
* return graph;
* }
* // Function to find the subset of an element i
* int find(Subset subsets[], int i) {
* if (subsets[i].parent != i)
* subsets[i].parent = find(subsets, subsets[i].parent);
* return subsets[i].parent;
* }
* // Function to perform union of two subsets
* void Union(Subset subsets[], int x, int y) {
* int xroot = find(subsets, x);
* int yroot = find(subsets, y);
* if (subsets[xroot].rank < subsets[yroot].rank)
* subsets[xroot].parent = yroot;
* else if (subsets[xroot].rank > subsets[yroot].rank)
* subsets[yroot].parent = xroot;
* else {
* subsets[yroot].parent = xroot;
* subsets[xroot].rank++;
* }
* }
* // Function to compare two edges based on their weights
* int compareEdges(const void\* a, const void\* b) {
* Edge\* edge1 = (Edge\*)a;
* Edge\* edge2 = (Edge\*)b;
* return edge1->weight - edge2->weight;
* }
* // Function to find the minimum spanning tree of a graph using Kruskal's algorithm
* void KruskalMST(Graph\* graph) {
* int numVertices = graph->numVertices;
* Edge result[numVertices];
* int e = 0;
* int i = 0;
* qsort(graph->edges, graph->numEdges, sizeof(graph->edges[0]), compareEdges);
* Subset subsets[numVertices];
* for (int v = 0; v < numVertices; v++) {
* subsets[v].parent = v;
* subsets[v].rank = 0;
* }
* while (e < numVertices - 1 && i < graph->numEdges) {
* Edge next\_edge = graph->edges[i++];
* int x = find(subsets, next\_edge.src);
* int y = find(subsets, next\_edge.dest);
* if (x != y) {
* result[e++] = next\_edge;
* Union(subsets, x, y);
* }
* }
* printf("Minimum Spanning Tree:\n");
* for (i = 0; i < e; ++i)
* printf("(%d, %d) - weight %d\n", result[i].src, result[i].dest, result[i].weight);
* }
* int main() {
* int V = 4;
* int E = 5;
* Graph\* graph = createGraph(V, E);
* graph->edges[0].src = 0;
* graph->edges[0].dest = 1;
* graph->edges[0].weight = 10;
* graph->edges[1].src = 0;
* graph->edges[1].dest = 2;
* graph->edges[1].weight = 6;
* graph->edges[2].src = 0;
* graph->edges[2].dest = 3;
* graph->edges[2].weight = 5;
* graph->edges[3].src = 1;
* graph->edges[3].dest = 3;
* graph->edges[3].weight = 15;
* graph->edges[4].src = 2;
* graph->edges[4].dest = 3;
* graph->edges[4].weight = 4;
* KruskalMST(graph);
* return 0;
* }
* This code demonstrates Kruskal's algorithm in C, which finds the minimum spanning tree of a graph. It utilizes a disjoint-set data structure to efficiently determine whether adding an edge to the spanning tree will create a cycle. The algorithm sorts the edges by weight and iterates over them, adding the smallest edge that doesn't form a cycle until all vertices are connected.
* 2.prims algorithm using c datastructures
* Prim's algorithm is another popular method for finding the minimum spanning tree (MST) of a connected, undirected graph. Unlike Kruskal's algorithm, which selects edges based on their weight, Prim's algorithm grows the MST from an arbitrary starting vertex by always choosing the smallest edge that connects a vertex in the MST to a vertex outside the MST. Here's an implementation of Prim's algorithm in C:
* Iimplementation of prims algorithm using c datastructures
* #include <stdio.h>
* #include <stdlib.h>
* #include <stdbool.h>
* #include <limits.h>
* #define V 5 // Number of vertices in the graph
* // Function to find the vertex with minimum key value,
* // from the set of vertices not yet included in MST
* int minKey(int key[], bool mstSet[]) {
* int min = INT\_MAX, min\_index;
* for (int v = 0; v < V; v++)
* if (mstSet[v] == false && key[v] < min)
* min = key[v], min\_index = v;
* return min\_index;
* }
* // Function to print the constructed MST stored in parent[]
* void printMST(int parent[], int graph[V][V]) {
* printf("Edge \tWeight\n");
* for (int i = 1; i < V; i++)
* printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
* }
* // Function to construct and print MST for a graph represented using adjacency matrix
* void primMST(int graph[V][V]) {
* int parent[V]; // Array to store constructed MST
* int key[V]; // Key values used to pick minimum weight edge in cut
* bool mstSet[V]; // To represent set of vertices not yet included in MST
* // Initialize all keys as INFINITE
* for (int i = 0; i < V; i++)
* key[i] = INT\_MAX, mstSet[i] = false;
* // Always include first vertex in MST
* key[0] = 0; // Make key 0 so that this vertex is picked as first vertex
* parent[0] = -1; // First node is always root of MST
* // The MST will have V vertices
* for (int count = 0; count < V - 1; count++) {
* // Pick the minimum key vertex from the set of vertices not yet included in MST
* int u = minKey(key, mstSet);
* // Add the picked vertex to the MST set
* mstSet[u] = true;
* // Update key value and parent index of the adjacent vertices of the picked vertex
* // Consider only those vertices which are not yet included in MST
* for (int v = 0; v < V; v++)
* if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
* parent[v] = u, key[v] = graph[u][v];
* }
* // Print the constructed MST
* printMST(parent, graph);
* }
* // Driver program to test above functions
* int main() {
* // Sample graph represented using adjacency matrix
* int graph[V][V] = {{0, 2, 0, 6, 0},
* {2, 0, 3, 8, 5},
* {0, 3, 0, 0, 7},
* {6, 8, 0, 0, 9},
* {0, 5, 7, 9, 0}};
* // Print the MST
* primMST(graph);
* return 0;
* }
* This C program demonstrates Prim's algorithm for finding the minimum spanning tree (MST) of a graph represented using an adjacency matrix. The algorithm starts from an arbitrary vertex and repeatedly adds the edge with the smallest weight that connects a vertex in the MST to a vertex outside the MST until all vertices are included in the MST. Finally, it prints the edges of the MST along with their weights.
* Top of Form
* Bottom of Form
* **6. Graph Applications**
* Real-world applications of graphs (e.g., social networks, navigation systems, etc.)
* Practical examples and case studies

**7. Advanced Topics (Optional)**

* Topological Sorting
* Strongly Connected Components (SCC)
* Graph Coloring
* Network Flow Algorithms (e.g., Ford-Fulkerson)

**8. Common Graph Problems and Solutions**

* Finding cycles in a graph
* Connectivity in graphs
* Bipartite graphs
* C implementations for each problem

**9. Conclusion**

Graphs are a critical data structure in computer science, widely used to model relationships and solve complex problems. In C programming, graphs can be represented and manipulated using various techniques such as adjacency matrices, adjacency lists, and edge lists. Understanding and implementing graph algorithms is essential for solving problems related to networking, pathfinding, scheduling, and more.

Bottom of Form