|  |  |
| --- | --- |
|  | Practical No: 08 |
|  |  |
| Explanation/ Stepwise-Proced ure/ Algorithm | Write a program to insert and delete nodes in the graph using and adjacency list. |
| Theory: | A graph is a data structure that consist a sets of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:   * Sequential representation (or, Adjacency matrix representation) * Linked list representation (or, Adjacency list representation)     In sequential representation, an adjacency matrix is used to store the graph. Whereas in linked list representation, there is a use of an adjacency list to store the graph. Sequential representation  In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph. We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.  If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.  An entry Aijin the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between Viand Vj. If an Undirected Graph G consists of n vertices, then the adjacency matrix for that graph is n x n, and the matrix A = [aij] can be defined as - aij= 1 {if there is a path exists from Vito Vj} aij= 0 {Otherwise}  It means that, in an adjacency matrix, 0 represents that there is no association exists between the nodes, whereas 1 represents the existence of a path between two edges.  If there is no self-loop present in the graph, it means that the diagonal entries of the adjacency matrix will be 0.  Now, let's see the adjacency matrix representation of an undirected graph. |

|  |  |
| --- | --- |
|  | In the above figure, an image shows the mapping among the vertices (A, B, C, D, E), and this mapping is represented by using the adjacency matrix. There exist different adjacency matrices for the directed and undirected graph. In a directed graph, an entry Aijwill be 1 only when there is an edge directed from Vito Vj.  Adjacency matrix for a directed graph  In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node. Consider the below-directed graph and try to construct the adjacency matrix of it.    In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix are 0.  Adjacency matrix for a weighted directed graph  It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path, here we have to use the weight associated with the edge. The weights on the graph edges will be represented as the entries of the adjacency matrix. We can understand it with the help of an example. Consider the below graph and its adjacency matrix representation. In the representation, we can see that the weight associated with the edges is represented as the entries in the adjacency matrix. |

|  |  |
| --- | --- |
|  | In the above image, we can see that the adjacency matrix representation of the weighted directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.  Adjacency matrix is easier to implement and follow. An adjacency matrix can be used when the graph is dense and a number of edges are large. Though, it is advantageous to use an adjacency matrix, but it consumes more space. Even if the graph is sparse, the matrix still consumes the same space.  Linked list representation  An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.  Let's see the adjacency list representation of an undirected graph.    In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list. An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.  The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.  Now, consider the directed graph, and let's see the adjacency list representation of that graph. |

|  |  |
| --- | --- |
|  | For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.  Now, consider the weighted directed graph, and let's see the adjacency list representation of that graph.    In the case of a weighted directed graph, each node contains an extra field that is called the weight of the node.  In an adjacency list, it is easy to add a vertex. Because of using the linked list, it also saves space. |
| Source  Code/Algorithm/ Flow Chart: | #include <stdio.h>  #include <stdlib.h>    #define MAX\_NODES 100      typedef struct Node { int vertex;  struct Node\* next;  } Node;      typedef struct Graph { |

|  |  |
| --- | --- |
|  | Node\* adjList[MAX\_NODES];  int numVertices;  } Graph;      // Function to create a new node  Node\* createNode(int vertex) {  Node\* newNode = (Node\*)malloc(sizeof(Node)); newNode->vertex = vertex; newNode->next = NULL;  return newNode;  }      // Function to create a graph  Graph\* createGraph(int vertices) {  Graph\* graph = (Graph\*)malloc(sizeof(Graph)); graph->numVertices = vertices;  for (int i = 0; i < vertices; i++) {  graph->adjList[i] = NULL;  }  return graph;  }      // Function to add an edge to the graph  void addEdge(Graph\* graph, int src, int dest) { Node\* newNode = createNode(dest); newNode->next = graph->adjList[src]; graph->adjList[src] = newNode; |

|  |  |
| --- | --- |
|  | newNode = createNode(src); newNode->next = graph->adjList[dest]; graph->adjList[dest] = newNode; // For undirected graph  }      // Function to delete an edge from the graph  void deleteEdge(Graph\* graph, int src, int dest) { Node\*\* curr = &graph->adjList[src];  while (\*curr) {  if ((\*curr)->vertex == dest) {  Node\* temp = \*curr; \*curr = (\*curr)->next; free(temp);  break;  }  curr = &(\*curr)->next;  }      curr = &graph->adjList[dest];  while (\*curr) { if ((\*curr)->vertex == src) {  Node\* temp = \*curr; \*curr = (\*curr)->next; free(temp); break;  }  curr = &(\*curr)->next;  } |

|  |  |
| --- | --- |
|  | }      // Function to delete a node from the graph  void deleteNode(Graph\* graph, int node) {  for (int i = 0; i < graph->numVertices; i++) {  if (i != node) { deleteEdge(graph, i, node);  }  }  graph->adjList[node] = NULL; // Remove the adjacency list of the node  }      // Function to display the graph  void displayGraph(Graph\* graph) {  for (int i = 0; i < graph->numVertices; i++) { Node\* temp = graph->adjList[i];  printf("%d -> ", i); while (temp) { printf("%d -> ", temp->vertex);  temp = temp->next;  }  printf("NULL\n");  }  }      // Main function to demonstrate the graph operations  int main() {  Graph\* graph = createGraph(4); |

|  |  |
| --- | --- |
|  | // Adding edges addEdge(graph, 0, 1); addEdge(graph, 0, 2); addEdge(graph, 1, 3);  addEdge(graph, 2, 3);      printf("Graph after adding edges:\n");  displayGraph(graph);    // Deleting an edge deleteEdge(graph, 0, 1); printf("\nGraph after deleting edge 0-1:\n");  displayGraph(graph);    // Deleting a node deleteNode(graph, 3); printf("\nGraph after deleting node 3:\n");  displayGraph(graph);    // Freeing the graph memory  for (int i = 0; i < MAX\_NODES; i++) { Node\* temp = graph->adjList[i];  while (temp) {  Node\* toDelete = temp; temp = temp->next;  free(toDelete);  } |
|  | }  free(graph);      return 0;  } |
| Output  Screenshots (if applicable) |  |
| Conclusion | Thus, we have studied and implemented Graph with adjacency list. |
| Post Lab Questions: | 1. What are ways of representing graph 2. Given the adjacency list and number of vertices and edges of a graph, the task is to represent the adjacency list for a directed graph.   Input: V = 3, edges[][]= {{0, 1}, {1, 2} {2, 0}} |

Practical No: 09

|  |  |
| --- | --- |
|  | |
| Explanation/  Stepwise-Procedure  / Algorithm | Write a program in C to implement Depth First Search. |
| Theory: | Now, let's understand the working of the DFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.    Now, let's start examining the graph starting from Node H. Step 1 - First, push H onto the stack.    1. STACK: H    Step 2 - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.    1. Print: H]STACK: A    Step 3 - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.     1. Print: A 2. STACK: B, D     Step 4 - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.     1. Print: D 2. STACK: B, F     Step 5 - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.    1. Print: F |

|  |  |
| --- | --- |
|  | 2. STACK: B    Step 6 - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.     1. Print: B 2. STACK: C     Step 7 - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.     1. Print: C 2. STACK: E, G     Step 8 - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.     1. Print: G 2. STACK: E     Step 9 - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.     1. Print: E 2. STACK:     Now, all the graph nodes have been traversed, and the stack is empty. The step-by-step process to implement the DFS traversal is given as follows -     1. First, create a stack with the total number of vertices in the graph. 2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack. 3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack. 4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top. 5. If no vertex is left, go back and pop a vertex from the stack. 6. Repeat steps 2, 3, and 4 until the stack is empty.     Applications of DFS algorithm  The applications of using the DFS algorithm are given as follows - |

|  |  |
| --- | --- |
|  | * DFS algorithm can be used to implement the topological sorting. * It can be used to find the paths between two vertices. * It can also be used to detect cycles in the graph. * DFS algorithm is also used for one solution puzzles. * DFS is used to determine if a graph is bipartite or not.     Algorithm  Step 1: SET STATUS = 1 (ready state) for each node in G  Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)  Step 3: Repeat Steps 4 and 5 until STACK is empty  Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)  Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)  [END OF LOOP]  Step 6: EXIT |
| Source  Code/Algorithm/Flo w Chart: | #include <stdio.h>  #include <stdlib.h>  #define MAX\_NODES 100  typedef struct Node { int vertex;  struct Node\* next;  } Node;    typedef struct Graph {  Node\* adjList[MAX\_NODES];  int numVertices;  } Graph;    // Function to create a new node  Node\* createNode(int vertex) {  Node\* newNode = (Node\*)malloc(sizeof(Node)); newNode->vertex = vertex; newNode->next = NULL; return newNode;  }  // Function to create a graph  Graph\* createGraph(int vertices) { |

|  |  |
| --- | --- |
|  | Graph\* graph = (Graph\*)malloc(sizeof(Graph)); graph->numVertices = vertices;  for (int i = 0; i < vertices; i++) { graph->adjList[i] = NULL;  }  return graph;  }  // Function to add an edge to the graph void addEdge(Graph\* graph, int src, int dest) { Node\* newNode = createNode(dest); newNode->next = graph->adjList[src]; graph->adjList[src] = newNode;    newNode = createNode(src); newNode->next = graph->adjList[dest];  graph->adjList[dest] = newNode; // For undirected graph  }  // Function to perform DFS  void DFS(Graph\* graph, int vertex, int visited[]) { visited[vertex] = 1; // Mark the current node as visited printf("%d ", vertex); // Print the current node    Node\* temp = graph->adjList[vertex]; while (temp) { int adjVertex = temp->vertex; if (!visited[adjVertex]) {  DFS(graph, adjVertex, visited);  }  temp = temp->next;  }  }  // Function to initiate DFS traversal  void depthFirstSearch(Graph\* graph, int startVertex) { int visited[MAX\_NODES] = {0}; // Array to track visited nodes DFS(graph, startVertex, visited);  }    int main() {  Graph\* graph = createGraph(MAX\_NODES); |

|  |  |
| --- | --- |
|  | // Adding edges to the graph addEdge(graph, 0, 1); addEdge(graph, 0, 2); addEdge(graph, 1, 3); addEdge(graph, 1, 4); addEdge(graph, 2, 5); addEdge(graph, 2, 6);    printf("Depth First Search starting from vertex 0:\n"); depthFirstSearch(graph, 0); printf("\n");    // Freeing the graph memory for (int i = 0; i < MAX\_NODES; i++) { Node\* temp = graph->adjList[i]; while (temp) {  Node\* toDelete = temp; temp = temp->next;  free(toDelete);  }  }  free(graph);    return 0;  } |
| Output Screenshots (if applicable) |  |
| Conclusion | Thus, we have studied and implemented Depth First Search. |
| Post Lab Questions: | * State the differences between DFS and BFS. * Find the DFS and BFS of following tree |

|  |  |
| --- | --- |
|  | Practical No: 10 |
|  |  |
| Explanation/  Stepwise-Procedure / Algorithm | Write a program in C to implement prims algorithm for a given directed Graph. |
| Theory: | Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.  Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.  How does the prim's algorithm work?  Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -     * First, we have to initialize an MST with the randomly chosen vertex. * Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree. * Repeat step 2 until the minimum spanning tree is formed.     The applications of prim's algorithm are -     * Prim's algorithm can be used in network designing. * It can be used to make network cycles. * It can also be used to lay down electrical wiring cables.     Example of prim's algorithm  Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.  Suppose, a weighted graph is - |

|  |  |
| --- | --- |
|  | Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.    Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.      Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.    Step 4 - Now, select the edge CD, and add it to the MST. |

|  |  |
| --- | --- |
|  | Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.      So, the graph produced in step 5 is the minimum spanning tree of the  given graph. The cost of the MST is given below - Cost of MST = 4 + 2 + 1 + 3 = 10 units.  Algorithm    Complexity of Prim's algorithm  Now, let's see the time complexity of Prim's algorithm. The running time of the prim's algorithm depends upon using the data structure for the graph and the ordering of edges. Below table shows some choices  -    Prim's algorithm can be simply implemented by using the adjacency matrix or adjacency list graph representation, and to add the edge with the minimum weight requires the linearly searching of an array of weights. It requires O(|V|2) running time. It can be improved further by using the implementation of heap to find the minimum weight edges in the inner loop of the algorithm.  The time complexity of the prim's algorithm is O(E logV) or O(V logV), where E is the no. of edges, and V is the no. of vertices. |

|  |  |
| --- | --- |
| Source | #include <stdio.h> |
| Code/Algorithm/Flo w Chart: | #include <limits.h>  #define MAX\_NODES 100    // Function to find the vertex with the minimum key value  int minKey(int key[], int visited[], int vertices) {  int min = INT\_MAX, minIndex;  for (int v = 0; v < vertices; v++) {  if (visited[v] == 0 && key[v] < min) { min = key[v];  minIndex = v;  }  }  return minIndex;  }    // Function to implement Prim's algorithm  void primMST(int graph[MAX\_NODES][MAX\_NODES], int vertices) { int parent[MAX\_NODES]; // Array to store the constructed MST int key[MAX\_NODES]; // Key values used to pick minimum weight edge  int visited[MAX\_NODES]; // Track vertices included in MST    // Initialize all keys as infinite and visited as false  for (int i = 0; i < vertices; i++) {  key[i] = INT\_MAX;  visited[i] = 0;  } |

|  |  |
| --- | --- |
|  | key[0] = 0; // Make the first vertex as the starting point  parent[0] = -1; // First node is always the root of MST    for (int count = 0; count < vertices - 1; count++) {  // Pick the minimum key vertex from the set of vertices not yet included in MST  int u = minKey(key, visited, vertices);      // Add the picked vertex to the MST set  visited[u] = 1;    // Update the key value and parent index of the adjacent vertices  for (int v = 0; v < vertices; v++) { if (graph[u][v] && visited[v] == 0 && graph[u][v] < key[v]) { parent[v] = u;  key[v] = graph[u][v];  }  }  }      // Print the constructed MST printf("Edge \tWeight\n");  for (int i = 1; i < vertices; i++) { printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);  }  } |

|  |  |
| --- | --- |
|  | int main() {  int graph[MAX\_NODES][MAX\_NODES] = { {0} };  int vertices;      // Number of vertices printf("Enter the number of vertices: ");  scanf("%d", &vertices);    // Input the adjacency matrix printf("Enter the adjacency matrix (enter 0 for no edge):\n"); for (int i = 0; i < vertices; i++) { for (int j = 0; j < vertices; j++) {  scanf("%d", &graph[i][j]);  }  }      primMST(graph, vertices);      return 0;  } |
| Output Screenshots (if applicable) |  |
| Conclusion | Thus, we have studied and implemented Prims algorithm for a given directed graph. |
| Post Lab Questions: | Use prims algorithm to find the minimum spanning tree for the following graph: |