**Roll No…………….. Total No. of Pages:……**

**ST-5 (SET-VI)**

**6th SEMESTER 2023-24**

**CS192- Advanced Data Structures**

**Time allowed: 90 Minutes Max. Marks: 40**

**General Instructions:**

* **Follow the instructions given in each section.**
* **Make sure that you attempt the questions in order.**

**SECTION-A (10\*1 mark=10 marks)**

***(All questions are compulsory)***

1. Which algorithm can detect cycles in both directed and undirected graphs?
   1. Bellman-Ford algorithm
   2. Floyd-Warshall algorithm
   3. **Union-Find algorithm**
   4. Kruskal's algorithm
2. In a directed graph, if there is a cycle, then there is at least one:
   1. Sink vertex
   2. **Source vertex**
   3. Leaf vertex
   4. Root vertex
3. Which algorithm can be used to perform topological sorting of a directed acyclic graph (DAG)?
   1. Dijkstra's algorithm
   2. Prim's algorithm
   3. **Depth-First Search (DFS)**
   4. Breadth-First Search (BFS)
4. The algorithm used to find the shortest path in an unweighted graph is:
   1. Dijkstra's algorithm
   2. Prim's algorithm
   3. Bellman-Ford algorithm
   4. **Breadth-First Search (BFS)**
5. Which algorithm is more suitable for dense graphs?

**a) Prim's algorithm.**

b) Kruskal's algorithm.

c) Both have the same suitability.

d) It depends on the specific graph.

1. When would Kruskal's algorithm not work correctly?

**a) When the graph contains negative-weight edges.**

b) When the graph is not connected.

c) When the graph contains cycles.

d) Kruskal's algorithm always works correctly.

1. A Hamiltonian cycle in a graph is a cycle that:

**a) Visits every vertex exactly once and returns to the starting vertex**

b) Visits every edge exactly once

c) Visits some vertices more than once

d) Visits some edges more than once

1. Which type of Trie represents a compact form of a Trie, specifically designed for character-based operations?
   1. **Radix Trie**
   2. Compressed Trie
   3. Suffix Trie
   4. Patricia Trie
2. What is the primary advantage of using a Compressed Trie over a regular Trie?
   1. Faster insertion
   2. **Reduced memory usage**
   3. Constant-time lookup
   4. Improved search performance
3. A graph with no cycles is called a:

a) Bipartite graph

**b) Tree**

c) Eulerian graph

d) Complete graph

**SECTION-B (5\*2 mark=10 marks)**

***(All questions are compulsory)***

11) Which of the following represent the correct pseudo code for non recursive DFS algorithm?

**a)**

**procedure DFS-non\_recursive(G,v):**

**//let St be a stack**

**St.push(v)**

**while St is not empty**

**v = St.pop()**

**if v is not discovered:**

**label v as discovered**

**for all adjacent vertices of v do**

**St.push(a) //a being the adjacent vertex**

b)

procedure DFS-non\_recursive(G,v):

//let St be a stack

St.pop()

while St is not empty

v = St.push(v)

if v is not discovered:

label v as discovered

for all adjacent vertices of v do

St.push(a) //a being the adjacent vertex

c)

procedure DFS-non\_recursive(G,v):

//let St be a stack

St.push(v)

while St is not empty

v = St.pop()

if v is not discovered:

label v as discovered

for all adjacent vertices of v do

St.push(v)

d)

procedure DFS-non\_recursive(G,v):

//let St be a stack

St.pop(v)

while St is not empty

v = St.pop()

if v is not discovered:

label v as discovered

for all adjacent vertices of v do

St.push(a) //a being the adjacent vertex

12) #include <iostream>

#include <vector>

using namespace std;

void FloydWarshall(vector<vector<int>>& graph) {

int n = graph.size();

for (int k = 0; k < n; ++k) {

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

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

if (graph[i][k] != INT\_MAX && graph[k][j] != INT\_MAX && graph[i][k] + graph[k][j] < graph[i][j]) {

graph[i][j] = graph[i][k] + graph[k][j];

}

}

}

}

}

int main() {

int n = 4;

vector<vector<int>> graph(n, vector<int>(n, INT\_MAX));

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

graph[i][i] = 0;

}

graph[0][1] = 3;

graph[1][2] = -2;

graph[2][0] = 7;

graph[2][3] = 1;

graph[3][0] = 2;

FloydWarshall(graph);

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

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

cout << graph[i][j] << " ";

}

cout << endl;

}

return 0;

}

What will be the output of the program?

**a) 0 3 1 2**

**1 0 -2 -1**

**3 6 0 1**

**2 5 3 0**

b) 0 3 -2 1

5 0 -4 1

7 10 0 1

2 5 3 0

c) 0 3 1 1

7 0 -2 1

5 8 0 1

2 5 3 0

d) 0 3 -2 1

7 0 1 1

5 8 0 1

2 5 3 0

13) #include <iostream>

#include <vector>

#include <climits>

using namespace std;

void BellmanFord(vector<vector<pair<int, int>>>& graph, int start) {

int n = graph.size();

vector<int> dist(n, INT\_MAX);

dist[start] = 0;

for (int i = 0; i < n - 1; ++i) {

for (int u = 0; u < n; ++u) {

for (auto edge : graph[u]) {

int v = edge.first;

int weight = edge.second;

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

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

}

}

}

}

for (int i = 1; i < n; ++i) {

cout << dist[i] << " ";

}

}

int main() {

int n = 4;

vector<vector<pair<int, int>>> graph(n + 1);

graph[1].push\_back({2, 3});

graph[1].push\_back({3, 2});

graph[2].push\_back({3, -5});

graph[3].push\_back({2, 1});

graph[3].push\_back({4, 2});

graph[4].push\_back({2, 2});

BellmanFord(graph, 1);

return 0;

}

What will be the output of the program?

a) 0 3 -3 1

b) 0 3 -5 1

c) 0 3 -3 2

**d) 0 -13 -14 -12**

14) Consider the following adjacency matrix for a directed graph:

a b c

a 0 ∞ 3

b ∞ 0 -2

c ∞ 1 0

After applying the Floyd-Warshall algorithm, what will be the distance from vertex 'a' to vertex 'b'?

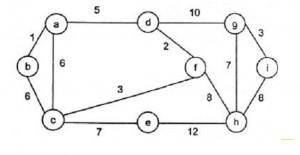
a) 0

b) 1

**c) 4**

d) ∞

15) For the undirected, weighted graph given below, which of the following sequences of edges represents a correct execution of Prim's algorithm to construct a Minimum Span­ning Tree?



a) (a, b), (d, f), (f, c), (g, i), (d, a), (g, h), (c, e), (f, h)

b) (c, e), (c, f), (f, d), (d, a), (a, b), (g, h), (h, f), (g, i)

**c) (d, f), (f, c), (d, a), (a, b), (c, e), (f, h), (g, h), (g, i)**

d) (h, g), (g, i), (h, f), (f, c), (f, d), (d, a), (a, b), (c, e)

**SECTION-C(Coding Question) (2x5 marks=5 marks)**

Q16) Given a positive weighted undirected graph, find the minimum weight cycle in it.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | [(0, 1, 4),(0, 7, 8),(1, 2, 8),(1, 7, 11),(2, 3, 7),(2, 8, 2),(2, 5, 4),(3, 4, 9),(3, 5, 14),(4, 5, 10),(5, 6, 2),(6, 7, 1),(6, 8, 6),(7, 8, 7)] | [(0, 1, 4),(0, 2, 8),(1, 2, 3)] | [(0, 1, 1), (0, 2, 3), (1, 3, 7),( 2, 3, 3), (2, 4, 1), (3, 4, 2)] |
| **Output** | 14 | 15 | 6 |

Solution :

**#include<bits/stdc++.h>**

**using namespace std;**

**# define INF 0x3f3f3f3f**

**struct Edge**

**{**

**int u;**

**int v;**

**int weight;**

**};**

**// weighted undirected Graph**

**class Graph**

**{**

**int V ;**

**list < pair <int, int > >\*adj;**

**// used to store all edge information**

**vector < Edge > edge;**

**public :**

**Graph( int V )**

**{**

**this->V = V ;**

**adj = new list < pair <int, int > >[V];**

**}**

**void addEdge ( int u, int v, int w );**

**void removeEdge( int u, int v, int w );**

**int ShortestPath (int u, int v );**

**void RemoveEdge( int u, int v );**

**int FindMinimumCycle ();**

**};**

**//function add edge to graph**

**void Graph :: addEdge ( int u, int v, int w )**

**{**

**adj[u].push\_back( make\_pair( v, w ));**

**adj[v].push\_back( make\_pair( u, w ));**

**// add Edge to edge list**

**Edge e { u, v, w };**

**edge.push\_back ( e );**

**}**

**// function remove edge from undirected graph**

**void Graph :: removeEdge ( int u, int v, int w )**

**{**

**adj[u].remove(make\_pair( v, w ));**

**adj[v].remove(make\_pair(u, w ));**

**}**

**// find the shortest path from source to sink using Dijkstra’s shortest path algorithm [ Time complexity O(E logV )]**

**int Graph :: ShortestPath ( int u, int v )**

**{**

**// Create a set to store vertices that are being preprocessed**

**set< pair<int, int> > setds;**

**// Create a vector for distances and initialize all**

**// distances as infinite (INF)**

**vector<int> dist(V, INF);**

**// Insert source itself in Set and initialize its distance as 0.**

**setds.insert(make\_pair(0, u));**

**dist[u] = 0;**

**/\* Looping till all shortest distance are finalized**

**then setds will become empty \*/**

**while (!setds.empty())**

**{**

**// The first vertex in Set is the minimum distance**

**// vertex, extract it from set.**

**pair<int, int> tmp = \*(setds.begin());**

**setds.erase(setds.begin());**

**// vertex label is stored in second of pair (it has to be done this way to keep the vertices**

**// sorted distance (distance must be first item in pair)**

**int u = tmp.second;**

**// 'i' is used to get all adjacent vertices of**

**// a vertex**

**list< pair<int, int> >::iterator i;**

**for (i = adj[u].begin(); i != adj[u].end(); ++i)**

**{**

**// Get vertex label and weight of current adjacent**

**// of u.**

**int v = (\*i).first;**

**int weight = (\*i).second;**

**// If there is shorter path to v through u.**

**if (dist[v] > dist[u] + weight)**

**{**

**/\* If the distance of v is not INF then it must be in our set, so removing it and inserting again**

**with updated less distance. \*/**

**if (dist[v] != INF)**

**setds.erase(setds.find(make\_pair(dist[v], v)));**

**// Updating distance of v**

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

**setds.insert(make\_pair(dist[v], v));**

**}**

**}**

**}**

**// return shortest path from current source to sink**

**return dist[v] ;**

**}**

**// function return minimum weighted cycle**

**int Graph :: FindMinimumCycle ( )**

**{**

**int min\_cycle = INT\_MAX;**

**int E = edge.size();**

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

**{**

**// current Edge information**

**Edge e = edge[i];**

**// get current edge vertices which we currently remove from graph and then find shortest path**

**// between these two vertex using Dijkstra’s shortest path algorithm .**

**removeEdge( e.u, e.v, e.weight ) ;**

**// minimum distance between these two vertices**

**int distance = ShortestPath( e.u, e.v );**

**// to make a cycle we have to add weight of currently removed edge if this is the shortest cycle then update min\_cycle**

**min\_cycle = min( min\_cycle, distance + e.weight );**

**// add current edge back to the graph**

**addEdge( e.u, e.v, e.weight );**

**}**

**// return shortest cycle**

**return min\_cycle ;**

**}**

**int main()**

**{**

**int V = 9;**

**Graph g(V);**

**// making above shown graph**

**g.addEdge(0, 1, 4);**

**g.addEdge(0, 7, 8);**

**g.addEdge(1, 2, 8);**

**g.addEdge(1, 7, 11);**

**g.addEdge(2, 3, 7);**

**g.addEdge(2, 8, 2);**

**g.addEdge(2, 5, 4);**

**g.addEdge(3, 4, 9);**

**g.addEdge(3, 5, 14);**

**g.addEdge(4, 5, 10);**

**g.addEdge(5, 6, 2);**

**g.addEdge(6, 7, 1);**

**g.addEdge(6, 8, 6);**

**g.addEdge(7, 8, 7);**

**cout << g.FindMinimumCycle() << endl;**

**return 0;**

**}**

Q17) A Hamiltonian path, is a path in an undirected graph that visits each vertex exactly once.

Given an undirected graph, the task is to check if a Hamiltonian path is present in it or not.

N (the number of vertices), M (Number of edges)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | N = 4, M = 4  Edges[][]= { {1,2}, {2,3}, {3,4}, {2,4} } | N = 3, M = 3  Edges[][]= { {1,2}, {2,3}, {3,1}} | N = 4, M = 3  Edges[][]= { {1,2}, {2,3}, {1,3}} |
| **Output** | 1 | 1 | 0 |

Solution :

**#include<bits/stdc++.h>**

**using namespace std;**

**class Solution**

**{**

**public:**

**// Function to recursively check for Hamiltonian Path**

**bool checkHamiltonian(int node, int currVis, int n, int m, vector<int>& vis, vector<int> adj[]) {**

**// Base case: If all nodes are visited, a Hamiltonian Path is found**

**if (currVis == n)**

**return true;**

**vis[node] = 1; // Mark the current node as visited**

**for (auto nodes : adj[node]) {**

**if (!vis[nodes]) {**

**if (checkHamiltonian(nodes, currVis + 1, n, m, vis, adj))**

**return true;**

**}**

**}**

**vis[node] = 0; // Backtrack: Mark the current node as unvisited**

**return false;**

**}**

**// Function to check for Hamiltonian Path in the graph**

**bool check(int N, int M, vector<vector<int>> Edges) {**

**vector<int> adj[N + 1]; // Adjacency list for the graph**

**// Creating the adjacency list from the given edges**

**for (auto vec : Edges) {**

**int u = vec[0], v = vec[1];**

**adj[u].push\_back(v);**

**adj[v].push\_back(u); // Since the graph is undirected**

**}**

**// For each node in the graph, check if there's a Hamiltonian Path**

**for (int i = 1; i <= N; i++) {**

**vector<int> vis(N + 1, 0); // Initialize the visited array**

**if (checkHamiltonian(i, 1, N, M, vis, adj))**

**return true; // If Hamiltonian Path exists, return true**

**}**

**return false; // If no Hamiltonian Path is found, return false**

**}**

**};**

**int main()**

**{**

**int N, M, X, Y;**

**cin >> N >> M; // Input: Number of nodes (N) and edges (M)**

**vector<vector<int>> Edges; // Stores the edges of the graph**

**// Input the edges from the user**

**for (int i = 0; i < M; i++) {**

**cin >> X >> Y;**

**Edges.push\_back({ X, Y });**

**}**

**Solution obj; // Create an instance of the Solution class**

**if (obj.check(N, M, Edges)) {**

**cout << "1" << endl; // Output: If Hamiltonian Path exists, print 1**

**}**

**else {**

**cout << "0" << endl; // Output: If no Hamiltonian Path, print 0**

**}**

**return 0;**

**}**

**SECTION-D (Coding Question)(1x10 mark=10 mark)**

Q18) Given a directed graph represented by an adjacency matrix, find the transitive closure of the graph using Warshall's algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | 1 1 0 1  0 1 1 0  0 0 1 1  0 0 0 1 | 0 1 0 1  0 1 0 0  0 0 1 1  0 0 0 1 | 0 1 0 1  0 1 0 0  1 0 1 1  1 0 0 1 |
| **Output** | Transitive Closure:  1 1 1 1  0 1 1 1  0 0 1 1  0 0 0 1 | Transitive Closure:  1 1 0 1  0 1 0 0  0 0 1 1  0 0 0 1 | Transitive Closure:  1 1 0 1  0 1 0 0  1 1 1 1  1 1 0 1 |

Solution :

**#include<stdio.h>**

**// Number of vertices in the graph**

**#define V 4**

**// A function to print the solution matrix**

**void printSolution(int reach[][V]);**

**// Prints transitive closure of graph[][]**

**// using Floyd Warshall algorithm**

**void transitiveClosure(int graph[][V])**

**{**

**/\* reach[][] will be the output matrix**

**// that will finally have the**

**shortest distances between**

**every pair of vertices \*/**

**int reach[V][V], i, j, k;**

**/\* Initialize the solution matrix same**

**as input graph matrix. Or**

**we can say the initial values of**

**shortest distances are based**

**on shortest paths considering**

**no intermediate vertex. \*/**

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

**for (j = 0; j < V; j++)**

**reach[i][j] = graph[i][j];**

**/\* Add all vertices one by one to the**

**set of intermediate vertices.**

**---> Before start of a iteration,**

**we have reachability values for**

**all pairs of vertices such that**

**the reachability values**

**consider only the vertices in**

**set {0, 1, 2, .. k-1} as**

**intermediate vertices.**

**----> After the end of a iteration,**

**vertex no. k is added to the**

**set of intermediate vertices**

**and the set becomes {0, 1, .. k} \*/**

**for (k = 0; k < V; k++)**

**{**

**// Pick all vertices as source one by one**

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

**{**

**// Pick all vertices as destination for the above picked source**

**for (j = 0; j < V; j++)**

**{**

**// If vertex k is on a path from i to j,**

**// then make sure that the value of reach[i][j] is 1**

**reach[i][j] = reach[i][j] ||**

**(reach[i][k] && reach[k][j]);**

**}**

**}**

**}**

**// Print the shortest distance matrix**

**printSolution(reach);**

**}**

**/\* A utility function to print solution \*/**

**void printSolution(int reach[][V])**

**{**

**printf ("Transitive Closure: \n");**

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

**{**

**for (int j = 0; j < V; j++)**

**{**

**/\* because "i==j means same vertex"**

**and we can reach same vertex**

**from same vertex. So, we print 1....**

**and we have not considered this in**

**Floyd Warshall Algo. so we need to**

**make this true by ourself**

**while printing transitive closure.\*/**

**if(i == j)**

**printf("1 ");**

**else**

**printf ("%d ", reach[i][j]);**

**}**

**printf("\n");**

**}**

**}**

**int main()**

**{**

**int graph[V][V] = { {1, 1, 0, 1},**

**{0, 1, 1, 0},**

**{0, 0, 1, 1},**

**{0, 0, 0, 1}**

**};**

**// Print the solution**

**transitiveClosure(graph);**

**return 0;**

**}**