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

**ST-5 (SET-III)**

**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 property is essential for finding the shortest cycle using the Floyd-Warshall algorithm?
   1. **Negative edge weights**
   2. Positive edge weights
   3. Acyclic graph
   4. Complete graph
2. The length of the shortest cycle in a graph is also known as the:
   1. Diameter of the graph
   2. Radius of the graph
   3. **Girth of the graph**
   4. Span of the graph
3. The algorithm used to find the shortest path in a graph with non-negative weights is:
   1. **Dijkstra's algorithm**
   2. Prim's algorithm
   3. Bellman-Ford algorithm
   4. Kruskal's algorithm
4. Which algorithm is used to find the transitive closure of a directed graph?
   1. Dijkstra's algorithm
   2. **Floyd-Warshall algorithm**
   3. Kruskal's algorithm
   4. Bellman-Ford algorithm
5. DFS is implemented using which data structure?

**a) Stack**

b) Queue

c) Priority Queue

d) Linked List

1. Which data structure is commonly used to implement the priority queue in Dijkstra's algorithm?

a) Stack

b) Queue

**c) Min-Heap**

d) Max-Heap

1. Dijkstra's algorithm is based on the principle of:

**a) Greedy Strategy**

b) Divide and Conquer

c) Depth-First Search

d) Breadth-First Search

1. Kruskal's algorithm is used for:

a) Finding shortest paths in a graph

**b) Finding minimum spanning tree of a graph**

c) Detecting cycles in a graph

d) Finding strongly connected components

1. In topological sorting of a directed acyclic graph (DAG), which vertex is processed last?

a) Source vertex (a vertex with no incoming edges)

**b) Sink vertex (a vertex with no outgoing edges)**

c) Random vertex

d) Vertex with the maximum number of incoming edges

1. In a Trie, what is the purpose of a null pointer in a child array?
   1. **It indicates the end of the Trie.**
   2. It represents a space character.
   3. It marks an invalid node.
   4. It is used for padding.

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

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

11) #include <iostream>

#include <vector>

#include <queue>

using namespace std;

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

queue<int> q;

q.push(start);

while (!q.empty()) {

int node = q.front();

q.pop();

cout << node << " ";

for (int neighbor : graph[node]) {

q.push(neighbor);

}

}

}

int main() {

int n = 5;

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

graph[1] = {3};

graph[2] = {5};

graph[3] = {4,2};

graph[4] = {};

graph[5] = {};

BFS(graph, 1);

return 0;

}

What will be the output of the program?

**a) 1 3 4 2 5**

b) 1 2 4 3 5

c) 1 2 3

d) 1 2

12) What is the major missing in BFS snippet given below?

while (!queue.empty()) {

s = queue.front();

cout << s << " ";

for (auto adjacent : adj[s]) {

if (!visited[adjacent]) {

visited[adjacent] = true;

queue.push\_back(adjacent);

}

}

}

a) Adjacent vertex is not visited

**b) front from queue is not popped**

c) Nothing

d) Adjacent vertex is not pushed to queue

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 = 5;

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

graph[1].push\_back({2, 6});

graph[1].push\_back({3, 5});

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

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

graph[2].push\_back({3, 7});

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

graph[4].push\_back({5, 1});

BellmanFord(graph, 1);

return 0;

}

What will be the output of the program?

a) 0 3 5 -2 0

**b) 0 3 5 1 2**

c) 0 5 3 2 5

d) 0 6 7 2 5

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

a b c

a 0 2 ∞

b ∞ 0 -4

c 5 ∞ 0

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

a) 0

b) -4

**c) 1**

d) ∞

15) Consider an undirected unweighted graph G. Let a breadth-first traversal of G be done starting from a node r. Let d(r, u) and d(r, v) be the lengths of the shortest paths from r to u and v respectively, in G. lf u is visited before v during the breadth-first traversal, which of the following statements is correct?

a) d(r, u) < d (r, v)

b) d(r, u) > d(r, v)

**c) d(r, u) <= d (r, v)**

d) None of the above

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

Q16) Given a set of strings, find the longest common prefix.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | {"apple", "ape", "april"} | {"zee", "zebra","zomato", "zen"} | {"mango", "pizza","chole", "rabdi"} |
| **Output** | Ap | z | There is no common prefix |

Solution :

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

**using namespace std;**

**// Alphabet size (# of symbols)**

**#define ALPHABET\_SIZE (26)**

**// Converts key current character into index use only 'a' through 'z' and lower case**

**#define CHAR\_TO\_INDEX(c) ((int)c - (int)'a')**

**// Trie node**

**struct TrieNode**

**{**

**struct TrieNode \*children[ALPHABET\_SIZE];**

**// isLeaf is true if the node represents end of a word**

**bool isLeaf;**

**};**

**// Returns new trie node (initialized to NULLs)**

**struct TrieNode \*getNode(void)**

**{**

**struct TrieNode \*pNode = new TrieNode;**

**if (pNode)**

**{**

**int i;**

**pNode->isLeaf = false;**

**for (i = 0; i < ALPHABET\_SIZE; i++)**

**pNode->children[i] = NULL;**

**}**

**return pNode;**

**}**

**// If not present, inserts the key into the trie**

**// If the key is a prefix of trie node, just marks leaf node**

**void insert(struct TrieNode \*root, string key)**

**{**

**int length = key.length();**

**int index;**

**struct TrieNode \*pCrawl = root;**

**for (int level = 0; level < length; level++)**

**{**

**index = CHAR\_TO\_INDEX(key[level]);**

**if (!pCrawl->children[index])**

**pCrawl->children[index] = getNode();**

**pCrawl = pCrawl->children[index];**

**}**

**// mark last node as leaf**

**pCrawl->isLeaf = true;**

**}**

**// Counts and returns the number of children of the current node**

**int countChildren(struct TrieNode \*node, int \*index)**

**{**

**int count = 0;**

**for (int i=0; i<ALPHABET\_SIZE; i++)**

**{**

**if (node->children[i] != NULL)**

**{**

**count++;**

**\*index = i;**

**}**

**}**

**return (count);**

**}**

**// Perform a walk on the trie and return the longest common prefix string**

**string walkTrie(struct TrieNode \*root)**

**{**

**struct TrieNode \*pCrawl = root;**

**int index;**

**string prefix;**

**while (countChildren(pCrawl, &index) == 1 &&**

**pCrawl->isLeaf == false)**

**{**

**pCrawl = pCrawl->children[index];**

**prefix.push\_back('a'+index);**

**}**

**return (prefix);**

**}**

**// A Function to construct trie**

**void constructTrie(string arr[], int n, struct TrieNode \*root)**

**{**

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

**insert (root, arr[i]);**

**return;**

**}**

**// A Function that returns the longest common prefix from the array of strings**

**string commonPrefix(string arr[], int n)**

**{**

**struct TrieNode \*root = getNode();**

**constructTrie(arr, n, root);**

**// Perform a walk on the trie**

**return walkTrie(root);**

**}**

**int main()**

**{**

**string arr[] = {"zee", "zebra",**

**"zomato", "zen"};**

**int n = sizeof (arr) / sizeof (arr[0]);**

**string ans = commonPrefix(arr, n);**

**if (ans.length())**

**cout << "The longest common prefix is "**

**<< ans;**

**else**

**cout << "There is no common prefix";**

**return (0);**

**}**

Q17) Write a C++ program to find maximum edges after adding which graph still remains a DAG.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | graph=[(5, 2),(5, 0),(4, 0),(4, 1),(2, 3),(3, 1)] | graph=[(0,1),(1,2)] | graph=[(0,1),(1,2),(1,4)] |
| **Output** | 4-5 4-2 4-3 5-3 5-1 2-0 2-1 0-3 0-1 | 0-2 | 0-3 0-2 3-1 3-2 |

Solution :

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

**using namespace std;**

**class Graph {**

**int V; // No. of vertices**

**// Pointer to a list containing adjacency list**

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

**// Vector to store indegree of vertices**

**vector<int> indegree;**

**// function returns a topological sort**

**vector<int> topologicalSort();**

**public:**

**Graph(int V); // Constructor**

**// function to add an edge to graph**

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

**// Prints all edges that can be added without making any**

**// cycle**

**void maximumEdgeAddition();**

**};**

**// Constructor of graph**

**Graph::Graph(int V)**

**{**

**this->V = V;**

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

**// Initialising all indegree with 0**

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

**indegree.push\_back(0);**

**}**

**// Utility function to add edge**

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

**{**

**adj[v].push\_back(w); // Add w to v's list.**

**// increasing inner degree of w by 1**

**indegree[w]++;**

**}**

**// Main function to print maximum edges that can be added**

**vector<int> Graph::topologicalSort()**

**{**

**vector<int> topological;**

**queue<int> q;**

**// In starting push all node with indegree 0**

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

**if (indegree[i] == 0)**

**q.push(i);**

**while (!q.empty()) {**

**int t = q.front();**

**q.pop();**

**// push the node into topological vector**

**topological.push\_back(t);**

**// reducing indegree of adjacent vertices**

**for (list<int>::iterator j = adj[t].begin();**

**j != adj[t].end(); j++) {**

**indegree[\*j]--;**

**// if indegree becomes 0, just push**

**// into queue**

**if (indegree[\*j] == 0)**

**q.push(\*j);**

**}**

**}**

**return topological;**

**}**

**// The function prints all edges that can be**

**// added without making any cycle**

**// It uses recursive topologicalSort()**

**void Graph::maximumEdgeAddition()**

**{**

**bool\* visited = new bool[V];**

**vector<int> topo = topologicalSort();**

**// looping for all nodes**

**for (int i = 0; i < topo.size(); i++) {**

**int t = topo[i];**

**// In below loop we mark the adjacent node of t**

**for (list<int>::iterator j = adj[t].begin();**

**j != adj[t].end(); j++)**

**visited[\*j] = true;**

**// In below loop unmarked nodes are printed**

**for (int j = i + 1; j < topo.size(); j++) {**

**// if not marked, then we can make an edge**

**// between t and j**

**if (!visited[topo[j]])**

**cout << t << "-" << topo[j] << " ";**

**visited[topo[j]] = false;**

**}**

**}**

**}**

**int main()**

**{**

**Graph g(6);**

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

**g.addEdge(5, 0);**

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

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

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

**g.addEdge(3, 1);**

**g.maximumEdgeAddition();**

**return 0;**

**}**

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

Q18) Write a C++ program to find strongly connected components in a given directed graph using Tarjan's algorithm (single DFS).

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | graph=[(0, 1),(1, 2),(2, 3)] | graph= [(1, 0),(0, 2),(2, 1),(0, 3),(3, 4)] | graph= [(0, 1),(1, 2),(2, 3),(2, 4),(3, 0),(4,2)] |
| **Output** | 3  2  1  0 | 4  3  1 2 0 | 4 3 2 1 0 |

Solution :

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

**#define NIL -1**

**using namespace std;**

**// A class that represents an directed graph**

**class Graph {**

**int V; // No. of vertices**

**list<int>\* adj; // A dynamic array of adjacency lists**

**// A Recursive DFS based function used by SCC()**

**void SCCUtil(int u, int disc[], int low[],**

**stack<int>\* st, bool stackMember[]);**

**public:**

**Graph(int V); // Constructor**

**void addEdge(int v,**

**int w); // function to add an edge to graph**

**void SCC(); // prints strongly connected components**

**};**

**Graph::Graph(int V)**

**{**

**this->V = V;**

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

**}**

**void Graph::addEdge(int v, int w) { adj[v].push\_back(w); }**

**// A recursive function that finds and prints strongly connected components using DFS traversal u --> The vertex**

**// to be visited next disc[] --> Stores discovery times of visited vertices low[] -- >> earliest visited vertex (the**

**// vertex with minimum discovery time) that can be reached from subtree rooted with current vertex**

**// \*st -- >> To store all the connected ancestors (could be partof SCC)**

**// stackMember[] --> bit/index array for faster check whether a node is in stack**

**void Graph::SCCUtil(int u, int disc[], int low[],**

**stack<int>\* st, bool stackMember[])**

**{**

**// A static variable is used for simplicity, we can avoid use of static variable by passing a pointer.**

**static int time = 0;**

**// Initialize discovery time and low value**

**disc[u] = low[u] = ++time;**

**st->push(u);**

**stackMember[u] = true;**

**// Go through all vertices adjacent to this**

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

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

**int v = \*i; // v is current adjacent of 'u'**

**// If v is not visited yet, then recur for it**

**if (disc[v] == -1) {**

**SCCUtil(v, disc, low, st, stackMember);**

**// Check if the subtree rooted with 'v' has a connection to one of the ancestors of 'u'**

**// Case 1 (per above discussion on Disc and Low value)**

**low[u] = min(low[u], low[v]);**

**}**

**// Update low value of 'u' only of 'v' is still in stack (i.e. it's a back edge, not cross edge).**

**// Case 2 (per above discussion on Disc and Low value)**

**else if (stackMember[v] == true)**

**low[u] = min(low[u], disc[v]);**

**}**

**// head node found, pop the stack and print an SCC**

**int w = 0; // To store stack extracted vertices**

**if (low[u] == disc[u]) {**

**while (st->top() != u) {**

**w = (int)st->top();**

**cout << w << " ";**

**stackMember[w] = false;**

**st->pop();**

**}**

**w = (int)st->top();**

**cout << w << "\n";**

**stackMember[w] = false;**

**st->pop();**

**}**

**}**

**// The function to do DFS traversal. It uses SCCUtil()**

**void Graph::SCC()**

**{**

**int\* disc = new int[V];**

**int\* low = new int[V];**

**bool\* stackMember = new bool[V];**

**stack<int>\* st = new stack<int>();**

**// Initialize disc and low, and stackMember arrays**

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

**disc[i] = NIL;**

**low[i] = NIL;**

**stackMember[i] = false;**

**}**

**// Call the recursive helper function to find strongly**

**// connected components in DFS tree with vertex 'i'**

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

**if (disc[i] == NIL)**

**SCCUtil(i, disc, low, st, stackMember);**

**}**

**int main()**

**{**

**cout << "\nSCCs in first graph \n";**

**Graph g1(5);**

**g1.addEdge(1, 0);**

**g1.addEdge(0, 2);**

**g1.addEdge(2, 1);**

**g1.addEdge(0, 3);**

**g1.addEdge(3, 4);**

**g1.SCC();**

**cout << "\nSCCs in second graph \n";**

**Graph g2(4);**

**g2.addEdge(0, 1);**

**g2.addEdge(1, 2);**

**g2.addEdge(2, 3);**

**g2.SCC();**

**cout << "\nSCCs in third graph \n";**

**Graph g3(7);**

**g3.addEdge(0, 1);**

**g3.addEdge(1, 2);**

**g3.addEdge(2, 0);**

**g3.addEdge(1, 3);**

**g3.addEdge(1, 4);**

**g3.addEdge(1, 6);**

**g3.addEdge(3, 5);**

**g3.addEdge(4, 5);**

**g3.SCC();**

**cout << "\nSCCs in fourth graph \n";**

**Graph g4(11);**

**g4.addEdge(0, 1);**

**g4.addEdge(0, 3);**

**g4.addEdge(1, 2);**

**g4.addEdge(1, 4);**

**g4.addEdge(2, 0);**

**g4.addEdge(2, 6);**

**g4.addEdge(3, 2);**

**g4.addEdge(4, 5);**

**g4.addEdge(4, 6);**

**g4.addEdge(5, 6);**

**g4.addEdge(5, 7);**

**g4.addEdge(5, 8);**

**g4.addEdge(5, 9);**

**g4.addEdge(6, 4);**

**g4.addEdge(7, 9);**

**g4.addEdge(8, 9);**

**g4.addEdge(9, 8);**

**g4.SCC();**

**cout << "\nSCCs in fifth graph \n";**

**Graph g5(5);**

**g5.addEdge(0, 1);**

**g5.addEdge(1, 2);**

**g5.addEdge(2, 3);**

**g5.addEdge(2, 4);**

**g5.addEdge(3, 0);**

**g5.addEdge(4, 2);**

**g5.SCC();**

**return 0;**

**}**