**GRAPH COLOURING**

**TASK - II(TEN/DS/146)**

[Graph coloring](http://en.wikipedia.org/wiki/Graph_coloring) problem is to assign colors to certain elements of a graph subject to certain constraints.

**Vertex coloring** is the most common graph coloring problem. The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using same color. The other graph coloring problems like **Edge Coloring** (No vertex is incident to two edges of same color) and **Face Coloring** (Geographical Map Coloring) can be transformed into vertex coloring.

**Chromatic Number**: The smallest number of colors needed to color a graph G is called its chromatic numb

**Applications of Graph Coloring:**

1) Making Schedule or Time Table: Suppose we want to make am exam schedule for a university. We have list different subjects and students enrolled in every subject. Many subjects would have common students (of same batch, some backlog students, etc). How do we schedule the exam so that no two exams with a common student are scheduled at same time? How many minimum time slots are needed to schedule all exams? This problem can be represented as a graph where every vertex is a subject and an edge between two vertices mean there is a common student. So this is a graph coloring problem where minimum number of time slots is equal to the chromatic number of the graph.

2) [Mobile Radio Frequency Assignment](http://www.zib.de/groetschel/teaching/SS2012/GraphCol%20and%20FrequAssignment.pdf): When frequencies are assigned to towers, frequencies assigned to all towers at the same location must be different. How to assign frequencies with this constraint? What is the minimum number of frequencies needed? This problem is also an instance of graph coloring problem where every tower represents a vertex and an edge between two towers represents that they are in range of each other.

3) Sudoku: Sudoku is also a variation of Graph coloring problem where every cell represents a vertex. There is an edge between two vertices if they are in same row or same column or same block.

4) [Register Allocation](http://en.wikipedia.org/wiki/Register_allocation): In compiler optimization, register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers. This problem is also a graph coloring problem.

5) Bipartite Graphs: We can check if a graph is Bipartite or not by coloring the graph using two colors. If a given graph is 2-colorable, then it is Bipartite, otherwise not.s.

6) Map Coloring: Geographical maps of countries or states where no two adjacent cities cannot be assigned same color. Four colors are sufficient to color any map

**IMPLEMENTATION USING GREEDY ALGO**

**Basic Greedy Coloring Algorithm:**

1. Color first vertex with first color.

2. Do following for remaining V-1 vertices.

a) Consider the currently picked vertex and color it with the

lowest numbered color that has not been used on any previously

colored vertices adjacent to it. If all previously used colors

appear on vertices adjacent to v, assign a new color to it.

**Time Complexity: O(V^2 + E) in worst case**

#include <bits/stdc++.h>

using namespace std;

class Graph{

int V;

list<int>\*adj;

public:

Graph(int V){

this->V=V;

adj=new list<int>[V];

}

void addEdge(int v,int w){

adj[v].push\_back(w);

adj[w].push\_back(v);

}

void graphColoring(){

int ans[V];

bool available[V];

ans[0]=0;

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

ans[i]=-1;

}

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

available[j]=false;

}

for(int u=1;u<V;u++){

list<int>::iterator x;

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

if(ans[\*x]!=-1){

available[ans[\*x]]=true;

}

}

int j;

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

if(available[j]==false){

break;

}

}

ans[u]=j;

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

if(ans[\*x]!=-1){

available[ans[\*x]]=false;

}

}

}

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

cout<<"Vertex "<<u<<" ---> Color "<<ans[u]<<endl;

}

}

~Graph(){

delete[] adj;

}

};

int main()

{

Graph g1(5);

g1.addEdge(0, 1);

g1.addEdge(0, 2);

g1.addEdge(1, 2);

g1.addEdge(1, 3);

g1.addEdge(2, 3);

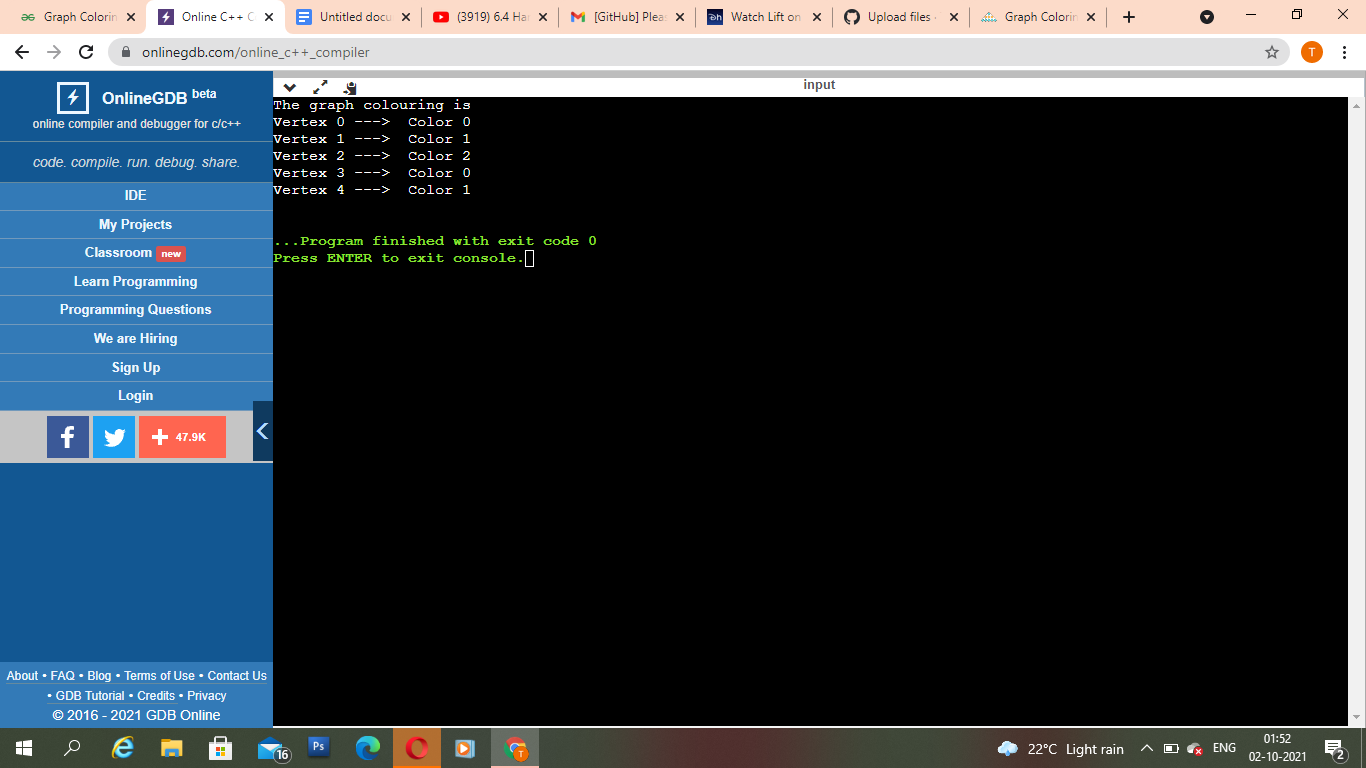
g1.addEdge(3, 4);

cout<<"The graph colouring is\n";

g1.graphColoring();

return 0;

}



**EDGE COLORING OF GRAPH**

Edge coloring of a graph is an assignment of “colors” to the edges of the graph so that no two adjacent edges have the same color with an optimal number of colors. Two edges are said to be adjacent if they are connected to the same vertex. There is no known polynomial time algorithm for edge-coloring every graph with an optimal number of colors. But,a number of algorithms have been developed that relax one or more of these criteria, they only work on a subset of graphs, or they do not always use an optimal number of colors, or they do not always run in polynomial time.

**IMPLEMENTATION OF EDGE COLORING USING BFS**

**Algorithm:**

1.Use BFS traversal to start traversing the graph.

2.Pick any vertex and give different colors to all of the edges connected to it, and mark those edges as colored.

3.Traverse one of it’s edges.

4.Repeat step to with a new vertexd until all edges are colored.

**Time Complexity: O(N) Where N is the number of nodes in the graph.**

**Auxiliary Space: O(N)**

#include <bits/stdc++.h>

using namespace std;

void edgeColouring(int ptr,vector<vector<pair<int,int>>>&graph,vector<int>&edgeColors,bool visited[]){

queue<int>q;

set<int> colored;

if(visited[ptr]){

return;

}

visited[ptr]=1;

int c=0;

for(int i=0;i<graph[ptr].size();i++){

if(edgeColors[graph[ptr][i].second]!=-1){

colored.insert(edgeColors[graph[ptr][i].second]);

}

}

for(int i=0;i<graph[ptr].size();i++){

if(!visited[graph[ptr][i].first]){

q.push(graph[ptr][i].first);

}

if(edgeColors[graph[ptr][i].second]==-1){

while(colored.find(c)!=colored.end()){

c++;

}

edgeColors[graph[ptr][i].second]=c;

colored.insert(c);

c++;

}

}

while(!q.empty()){

int temp=q.front();

q.pop();

edgeColouring(temp,graph,edgeColors,visited);

}

return;

}

int main()

{

vector<vector<pair<int,int>>>graph;

vector<int>edgeColors;

bool visited[1000000]={0};

int vertex=4,edge=4;

graph.resize(vertex);

edgeColors.resize(edge,-1);

graph[0].push\_back(make\_pair(1, 0));

graph[1].push\_back(make\_pair(0, 0));

graph[1].push\_back(make\_pair(2, 1));

graph[2].push\_back(make\_pair(1, 1));

graph[2].push\_back(make\_pair(3, 2));

graph[3].push\_back(make\_pair(2, 2));

graph[0].push\_back(make\_pair(3, 3));

graph[3].push\_back(make\_pair(0, 3));

edgeColouring(0,graph,edgeColors,visited);

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

cout << "Edge " << i + 1 << " is of color "<< edgeColors[i] + 1 << "\n";

}

return 0;

}

