Acyclic Coloring of Graphs

Submitted in partial fulfilment of the requirements for the degree of

Master of Technology

in

Computer Science(Information Security)

by

Lavish Kothari

15IS12F

Lavishkothari004@gmail.com

Department of Computer Science

NATIONAL INSTITUTE OF TECHNOLOGY

SURATHKAL, INDIA

CERTIFICATE

It is certified that the work contained in this Report, titled "***Acyclic Edge Coloring of Graphs***" by Lavish Kothari, has been carried out under my supervision and is not submitted elsewhere for a degree.

_____ _____

Date Guide : Dr. Manu Basavaraju

Acknowledgement

I am grateful to my advisor Dr. Manu Basavaraju for introducing me to research and believing that I could do good research.

I would also like to thank for all the facilities that I worked with and students including my class-mates from whom I have learnt immensely.

I would specially like to thank my guide for all the support, encouragement and intellectual capital he provided me with.

I also acknowledge the support of Department of Computer Science and Technology for their constant support.

Abstract

Graph coloring is a fundamental problem in Computer Science. Despite its status as a computationally hard problem, it is still an active area of research. Depending on the specific area of requirement or an application, there are several variants of coloring.

One such variant of graph coloring is Acyclic Graph Coloring. This requires that after a graph has been colored there should not exist any cycle that runs on vertices that are colored by only two colors. Such a cycle is called a bichromatic cycle. When a coloring does not present itself with a bichromatic cycle, it is said to have acyclically colored the given graph.

As with the case of proper coloring of graphs, acyclic coloring can either be on vertex set or the edge set of a graph. When such a coloring is performed on the edges of a graph, a bichromatic cycle is said to be one that runs on the edges that are colored by only two colors. In absence of such a bichromatic cycle, an edge coloring is said to be an acyclic edge coloring of that graph.

The smallest number of colors required to acyclically vertex color a graph is called its acyclic chromatic number and is denoted by $a(G)$. Similarly, the smallest number of colors required to acyclically edge colors a graph is called its acyclic edge chromatic number and is denoted by $a'(G)$.

A Proper Edge Coloring of G = (V,E) is a map c : E→C (where C is the set of available colors) with $c(e) \neq c(f)$ for any adjacent edges e, f. A proper coloring of the edges of a graph G is called acyclic if there is no 2-colored cycle in G. The minimum number of colors needed to properly color the edges of G, is called the Edge Chromatic Index of G and is denoted by $\chi'(G)$. The acyclic edge chromatic number of G, denoted by $a'(G)$,

is the least number of colors in an acyclic edge coloring of G. It is known that *a*'(G) ≤ 16 Δ(G) for any graph G.

Acyclic Edge coloring Conjecture by Alon, Sudakov, and Zaks (and independently by Fiamcik): **For any Graph G, *a*' ≤ Δ (G) + 2** where Δ represents the maximum degree amongst all the vertices.

This report basically focuses on the Acyclic Edge coloring of non-regular sub-cubic graphs, here it proves that the acyclic edge chromatic index of such graphs a' ≤ 4. Also it is obvious that If Δ(G) < 3 then ***a*'(G) ≤ 3.**

There are several applications of acyclic coloring of a graph which include Hessian computation, applications in coding theory as well as other theoretical problems such as Minimim Feedback Vertex Set. In addition to this, acyclic chromatic numbers are closely related to several other types to colorings of a graph each of which are applicable in several other fields of interest. A notable example among these is the star chromatic number of a graph and star arboricity. Star graphs are used to model star network topologies that have applications in computer networks and distributed computing.

In this report I have basically discussed the acyclic edge coloring of non-regular sub cubic graphs. For proving the stated theorem various graph operations to assign a valid acyclic edge coloring. I have extended this same concept to the graphs of maximum degree 5 and I was able to prove that graphs with maximum degree 5 can be acyclically edge colored using at most 15 colors.

CONTENTS

Chapter 1

INTRODUCTION

Notations

- Let **G = (V , E)** be a graph such that **E ⊆ V X V**
  where E is the Edge set and V is the Vertex Set.
  We usually denote
  |V| = n
  |E| = m

- **δ(G) = min {deg_G(v) | v ∈ V(G)}** – this is minimum degree in the graph

- **Δ(G) = max {deg_G(v) | v ∈ V(G)}** – this is the maximum degree in the graph

- **N_G(u)** = neighbors of vertex u in G.

- For e ∈ E
  G-e denotes the graph obtained by the deletion of edge e.

- A coloring c, of a graph is denoted by **c : E → {1,2,3….k}**
  c(e) denotes the color given to the edge e with respect to the coloring c.

- For any vertex u, **F_u(c) = { c(u , z) | z ∈ N_G(u) }**

- For a, b ∈ V, **S_{ab}(c) = F_b(c)-{c(a , b)}**

Basics

(α, β ,a , b) Maximal Bichromatic Path

Let a,b ∈ V, The (α, β ,a , b) Maximal Bichromatic Path is a maximal path that starts at vertex a with an edge color α, and ends at b.

Note:

- The edge of (α, β ,a , b) Maximal Bichromatic Path incident on a is colored α.
- The edge of (α, β ,a , b) Maximal Bichromatic Path incident on b can be either coloured as α or β.
- (α, β ,a , b) and (α, β ,a , b) Maximal Bichromatic Paths have different meaning.
- Maximal Bichromatic Path have at least 2 edges.

(α, β ,ab) Critical Path

Let a, b ∈ V, ab ∈ E and c be the partial coloring of G, then (α, β, a, b) Maximal Bichromatic Path which starts out from the vertex a via an edge colored α and ends at vertex b via an edge colored α, is called (α, β, ab) Critical Path.

Note:

- Every Critical Path will be of odd length.
- For a critical path, the smallest length possible is 3.

Candidate Color

A color α ≠ c(e) is a candidate color for an edge e in G with respect to parital coloring c of G if none of the adjacent edges of e are colored α**.**

Valid Color

A candidate color α is valid for an edge e if assigning the color α to e does not results in any bichromatic cycles in G.

Operations

- ColorExchange
- Recolor

1. Color Exchange

Let c be a partial coloring of G. Let u, i, j ∈ V(G) and ui, uj ∈ E(G).

We define Color Exchange with respect to the edge ui and uj as the modification of the current partial coloring c by exchanging the colors of ui and uj to get a partial coloring c'.

c' (u , i) = c (u , j)

c' (u , j) = c (u , i)

c' (e) = c (e) for all other edges e in G.

The above color exchange is denoted by

c' = ColorExchange (c, ui, uj)

2. Recolor

We define c' = Recolor(c, e, γ) as the recoloring of the edge e with a candidate color γ to get a modified coloring c' from c, i.e., c'(e) = γ and c'(f) = c(f), for all other edges f in G.

- The recoloring is said to be proper, if the coloring c' is proper.
- The recoloring is said to be acyclic (valid), if in coloring c' there exists no bichromatic cycle.

Some Important Observations and Lemma

- Given a pair of colors α and β of proper coloring c of G, there can be at most one maximal (α, β) bichromatic path containing a vertex v with respect to c.
- A candidate color of an edge e = uv is valid if
  $F_u \cap F_v - \{ c (u, v)\} = (S_{uv} \cap S_{vu}) = \emptyset$
- Let c be a partial coloring of G, a candidate color β is not valid for the edge e = (a , b) if and only if $\exists \alpha \in S_{ab} \cap S_{ba}$ such that there is a (α, β, ab) critical path in G with respect to coloring c.

Lemma

Let c' be the partial coloring obtained from a valid partial coloring c by the color exchange with respect to the edges ui and uj.

The partial coloring c' will be proper if and only if the following two conditions are true.

- $c(u , i) \notin S_{uj}$
- $c(u , j) \notin S_{ui}$

Chapter 2

ACYCLIC EDGE COLORING OF SUB-CUBIC GRAPHS

A graph is called subcubic if the maximum degree of the graph is three. Here we will look at the acyclic edge coloring of subcubic graphs.

Previous results on Sub-cubic Graphs:

1. Any sub-cubic graph can be acyclically edge colored using at most 5 colors.
2. Skulrattankulchai gave a polynomial time algorithm to color a subcubic graph using $\Delta + 2 = 5$ colors.

The Theorem:

Let G be a non-regular connected graph of maximum degree 3, then $a'(G) \leq 4$

Note:

- If $\Delta(G) < 3$ then $a'(G) \leq 3$

Proof:

Induction on number of Edges:

Base Case : Smallest possible edges on a non regular connected graph G of maximum degree 3 on n vertices is n-1. Then G is a tree and is acyclically colorable using 3 colors. (As there will be no cycles, we don't need to worry about the acyclicity of edges. Every proper edge coloring of a tree will be acyclic.)

Induction Hypothesis :

Let G be a

- Connected graph
- Non-regular Graph
- $\Delta(G) = 3$
- $m \geq n$

m = number of edges

n = number of vertices

Let the theorem be true for all non regular connected graphs with maximum degree 3

with at most m-1 edges.

Assumption:

Without the loss of generality we can assume that G is 2-connected.

So **$\delta(G) \geq 2$**

If there are cut vertices in G, then the acyclic coloring of blocks of G can be easily extended to G.

Since G is not 3 regular and **$\delta(G) \geq 2$.**

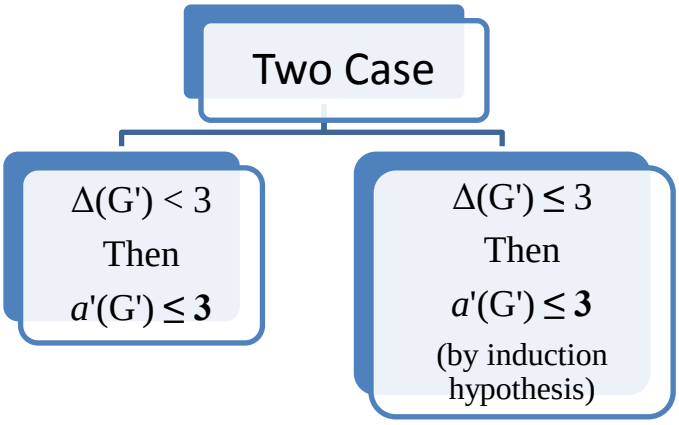
Then there is definitely a vertex of degree 2, let this vertex be x.

$\deg_G(x) = 2$

Let $y \in N_G(x)$

$G' = G - \{ xy \}$

G' is connected, since G is 2-connected.

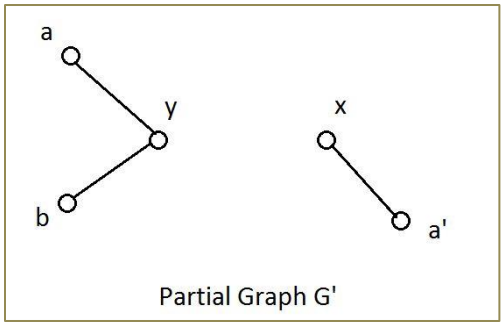


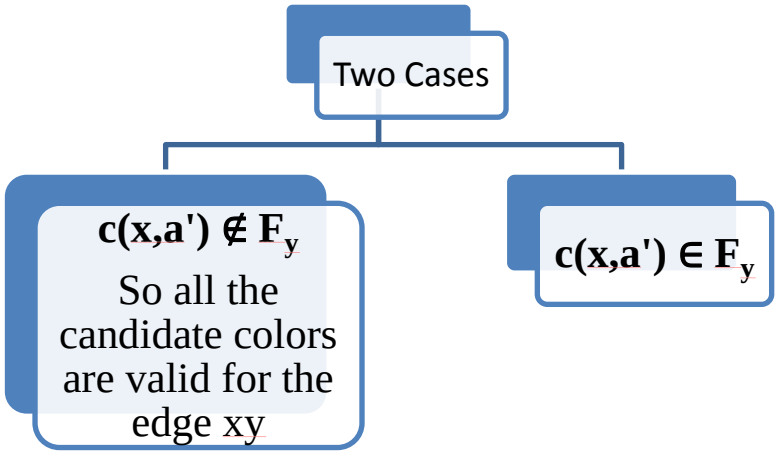
Approach towards Proving:

We will try to extend the edge coloring of G' to G by giving a color to the edge xy from available 4 colors.

Since **$| F_y \cup c(x, a') | \leq 3$,**

So there is at least one candidate color for the edge xy

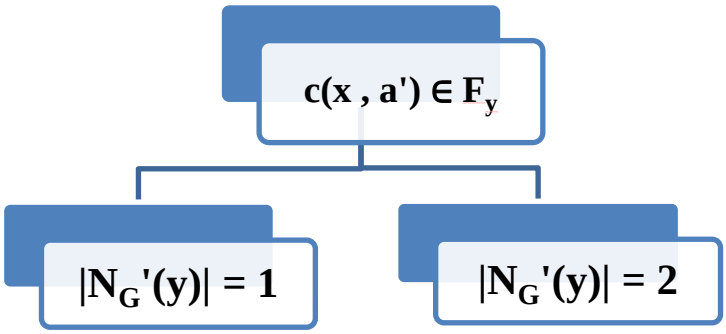


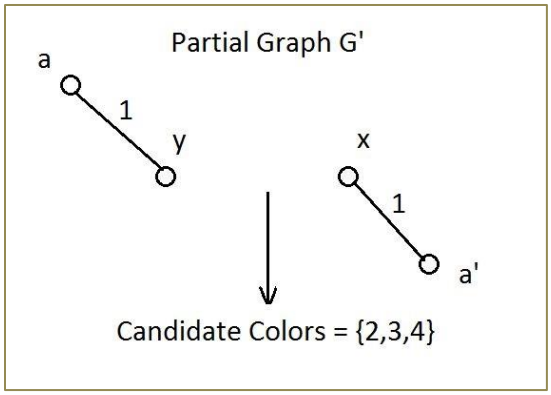


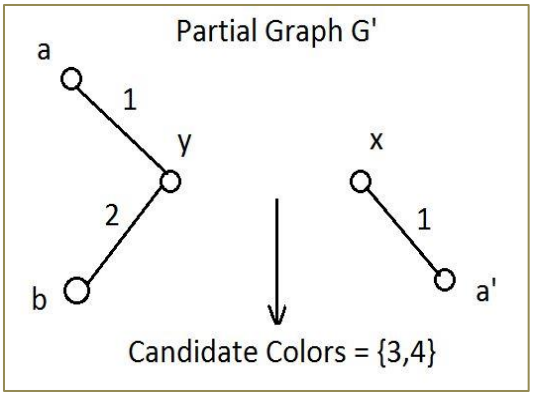
case 1: c(x, a') ∉ F_y

Then clearly all the candidate colors are valid for the edge *xy*, since any cycle involving the edge *xy* will contain the edge *xa'* as well as an edge incident on *y* in *G'* and thus the cycle will have at least 3 colors.

case 2: c(x, a') ∈ Fy







Without loss of generality let $a \in N_G(y)$ be the vertex such that $c(x, a') = c(y, a) = 1$. Suppose first $|NG'(y)| = 1$. Then we have 3 candidate colors $\{2, 3, 4\}$. Suppose $\alpha \in \{2, 3, 4\}$ is not valid, because if we assign color α to the edge xy, a bichromatic cycle is formed. It is easy to check that this has to be a (1, α) bichromatic cycle. It follows that if α is not valid there exists a (1, α) maximal bichromatic path with x and y as end vertices in G' with respect to the coloring c. Now if a color α is not valid, then it should be in Sa' to form a (1, α) maximal bichromatic path. But $|Sa'| \leq 2$ and hence $|\{2, 3, 4\} - Sa'| \geq 1$. Thus at least one color from $\{2, 3, 4\}$ is *valid* for the edge xy.

Now we can assume $|NG'(y)| = 2$. Let $NG'(y) = \{a, b\}$. Without loss of generality let $c(y, b) = 2$. Now colors $\{3, 4\}$ are candidates for the edge xy. If both the colors 3 and 4 are not valid for the edge xy, then there are (1,3) and (1,4) maximal bichromatic paths starting at the vertex y, passing through vertex a and ending at vertex x. Thus $Sa' = \{3, 4\}$ and $Sya = \{3, 4\}$.

Now recolor the edge xa' with color 2. Let the new coloring be called c'. Note that since $2 \notin S_{a'}$ and a is a pendant vertex in G', coloring c' is a valid acyclic edge coloring of the graph G'. Even with respect to the coloring c', colors $\{3, 4\}$ are candidates for the edge xy. If both the colors 3 and 4 remain invalid for the edge xy even now, it means that there are (2,3) and (2,4) maximal bichromatic paths starting at the vertex y, passing through vertex b and ending at the vertex x. Thus $S_{yb} = \{3, 4\}$, where $S_{yb} = \{ c(b, z) \mid z \in NG'(b) - \{y\} \}$. Let P be the above discussed (2,4)- maximal bichromatic path with respect to c'. Note that P does not contain a as $2 \notin Sya$.

Now we can exchange the colors of the edges *ya* and *yb* to get the coloring *c''*. That is, with respect to *c''*, we will have *c''*(*y, a*) = 2, *c''*(*y, b*) = 1 and *c''*(*e*) = *c'*(*e*), for all other edges *e* in *G'*. Note that the coloring *c''* is proper since 1 ∉ *Syb* and 2 ∉ *Sya*. Now suppose there is a bichromatic cycle with respect to *c''*. Then clearly this bichromatic cycle should contain both the edges *ya* and *yb* as *degG'*(*y*) = 2. Moreover, there has to be an edge colored 2 at vertex *b*. Recall that S_{yb} = {3, 4} and thus at vertex *b* only colors {1, 3, 4} are present. Thus the coloring *c''* is acyclic. Since *c''*(*y, b*) = 1, the path *P*, which was bichromatic in coloring *c'* has 3 colors in the coloring *c''*. Let *P'* = *P* − *y*. It is easy to verify that *P'* is a (2,4) maximal bichromatic path which starts from vertex *x* and ends at vertex *b* and does not contain vertex *a*. We have *c''*(*y, a*) = *c''*(*x, a'*) = 2. There can only be at most one (2,4)-maximal bichromatic path starting from the vertex *x*. We know *P'* is such a path and it does not include vertex *a*. Thus there cannot be a (2,4) maximal bichromatic path which starts at vertex *x* and ends at vertex *y*, passing through vertex *a*. Thus color 4 is valid for the edge *xy*.

Chapter 3

IMPLEMENTATION DETAILS

Algorithm

- **Pick the edges in certain order**.
- The initial ordering of edges is necessary because if the edges are picked arbitrarily, then the edge may not fall at all into any case discussed before.
- So the edges should be picked in such a way that we always have a vertex (x) of degree 1 in G'.
- **How preferable edge ordering is decided?**
  - Pick an edge with at least one of it's endpoint with degree 2 or less, let us call this end point x.
  - Now give numbering to all the edges incident on x and then delete this vertex.
  - Repeat the above two steps and keep enumerating the edges incrementally.
- **How to use the current Edge Ordering?**
  - Pick the edges with decreasing edge number, so that each time you have the edge (to be colored) that falls in one of the discussed case.

Code Details

For implementation of this theorem I have used object oriented features of C++ and it's Standard Library. The standard library of C++ is really a powerful tool as it has already defined functions in a very optimal way. To quote a few examples set_intersection and set_difference are used for taking intersection and difference of two sets. These functions are implemented in optimal way in the algorithm header file.

The code is as follows:

```
#include<stdio.h>
#include<set>
#include<algorithm>
#include<list>

using namespace std;
#define MAX_COLORS 4 // This specifies the maximum number of colors that we
will be using to color the graph.
```

```
#define MAX_DEGREE 3 // This specifies the maximum degree of the graph.
#define IS_REGULAR 0 // 0 specifies that the graph is not regular and 1
specifies that the graph is regular.

class Vertex
{
	public:
	int vertexNumber;
	list<int>adjacencyList;
};

class Edge
{
	public:
	int start,end,color;
	Edge();
	Edge(Edge const &);
	Edge(int,int);
};
Edge::Edge()
{
}
Edge::Edge(int start,int end)
{
	this->start=start;
	this->end=end;
	this->color=-1;
}

Edge::Edge(Edge const&e)
{
	this->start=e.start;
	this->end=e.end;
	this->color=-1;
}

class Graph
{
	public:
	int numberOfVertices,numberOfEdges;
	Vertex* vertexArray;
	list<Edge> edgeList;
	set<int>C;

	Graph(int,int);
	void addEdge(int,int);
	void print();
	void deleteGraph();
	void arrangeEdges(); // very important thing. for description see the
function definition.
	Vertex& findVertexWithDegreeAtMost2(bool*);
	void colorEdge(Edge&);
	Edge findEdge(int,int); // given vertexNumber number of two vertices,
this function returns the edge between them.
	int findColor(int,int); // given two vertices of graph this function
returns its color.
	set<int> findCandidateColors(Edge const&); // finds the candidate
colors for a given edge.
	set<int> S(int,int); // for S(x,a) this method returns the set of
colors {c : c=color(a,b) for every b belongs to N(a)-{x} } where N(a) is the
neighbor set of vertex a.
	bool isCriticalPath(Edge const&e,int a,int b); // this method checks
where there exists a (e,a,b) critical path whith colors a and b
	void recolor(Edge &e,int c); // recolors the edge e with color c.
```

```
	void colorExchange(Edge &e1,Edge &e2); // exchanges the color of edge
e1 and e2.
	bool isInConfigurationA(Vertex const&,Vertex const&,Vertex
const&,set<int>const&,set<int>const&); // checks whether the five tuples are
in configuration A or not.
};

Graph::Graph(int numberOfVertices,int numberOfEdges)
{
	this->numberOfVertices=numberOfVertices;
	this->numberOfEdges=numberOfEdges;
	vertexArray=new Vertex[numberOfVertices];
	for(int i=0;i<numberOfVertices;i++)
		vertexArray[i].vertexNumber=i;
	for(int i=1;i<=MAX_COLORS;i++)
		C.insert(i);
}
void Graph::addEdge(int start,int end)
{
	Edge*e=new Edge(start,end);
	(this->edgeList).push_back(*e);

	vertexArray[start].adjacencyList.push_back(end);
	vertexArray[end].adjacencyList.push_back(start);
}
int Graph::findColor(int a,int b)
{
	for(list<Edge>::iterator i=edgeList.begin();i!=edgeList.end();i++)
	{
		Edge e;
		e=*i;
		if( ((e.start)==a && (e.end)==b) || ((e.start)==b &&
(e.end)==a) )
			return e.color;
	}
	return -1;
}
set<int> Graph::S(int a,int b)
{
	set<int>colors;
	for(list<int>::iterator
i=vertexArray[b].adjacencyList.begin();i!=vertexArray[b].adjacencyList.end();
i++)
	{
		if((*i)!=a)
		{
			colors.insert(findColor(b,*i));
		}
	}
	return colors;
}
set<int> Graph::findCandidateColors(Edge const &e)
{
	Vertex v,u;
	u=vertexArray[e.start];
	v=vertexArray[e.end];

	set<int>fu,fv;
	for(list<int>::iterator
i=u.adjacencyList.begin();i!=u.adjacencyList.end();i++)
		fu.insert(findColor(u.vertexNumber,*i)); // insert the color of
Fu into the set.
	for(list<int>::iterator
i=v.adjacencyList.begin();i!=v.adjacencyList.end();i++)
```

```
            fv.insert(findColor(v.vertexNumber,*i)); // insert the color of
Fv into the set.

      set<int>fuUNIONfv;
      set_union(fu.begin(),fu.end(),fv.begin(),fv.end(),inserter(fuUNIONfv,fu
UNIONfv.begin()));

      set<int>candidateColors;
      set_difference(C.begin(),C.end(),fuUNIONfv.begin(),fuUNIONfv.end(),inse
rter(candidateColors,candidateColors.begin()));
      return candidateColors;
}
void printSet(set<int>const &a)
{
      printf("Printing the set : ");
      for(set<int>::iterator i=a.begin();i!=a.end();i++)
            printf("%d, ",*i);
      printf("\n");

}
void Graph::recolor(Edge &e,int c)
{
      e.color=c;
}
void Graph::colorExchange(Edge &e1,Edge &e2)
{
      int temp=e1.color;
      e1.color=e2.color;
      e2.color=temp;
}
Edge Graph::findEdge(int a,int b)
{
      for(list<Edge>::iterator i=edgeList.begin();i!=edgeList.end();i++)
      {
            Edge e=*i;
            if( (e.start==a && e.end==b) || (e.start==b && e.end==a) )
                  return e;
      }
}

// this method checks where there exists a (e,a,b) critical path whith colors
a and b
// filling color b for edge e will lead to bichromatic cycle
bool Graph::isCriticalPath(Edge const&e,int a,int b) // this is a important
method.
{
      int currentVertexNumber=e.start;
      int currentColor=a;
      while(currentVertexNumber!=e.end)
      {
            list<int>::iterator i;

      for(i=vertexArray[currentVertexNumber].adjacencyList.begin();i!=vertexA
rray[currentVertexNumber].adjacencyList.end();i++)
            {
                  if(findColor(currentVertexNumber,*i)==currentColor)
                        break;
            }
            if(i==vertexArray[currentVertexNumber].adjacencyList.end())
                  return false;
            currentVertexNumber=*i;
            /* here you need to flip the current color */
            if(currentColor==a)
                  currentColor=b;
```

```
            else
                  currentColor=a;
      }
      return true;
}

bool Graph::isInConfigurationA(Vertex const&u,Vertex const&a,Vertex
const&b,set<int>const&NDash,set<int>const&NDoubleDash)
{
      set<int>Sua=S(u.vertexNumber,a.vertexNumber);
      set<int>Sub=S(u.vertexNumber,b.vertexNumber);
      Edge ua=findEdge(u.vertexNumber,a.vertexNumber);
      Edge ub=findEdge(u.vertexNumber,b.vertexNumber);
      if( (Sua.find(ub.color)!=Sua.end()) ||
(Sub.find(ua.color)!=Sub.end()) )
            return false;
      set<int>intersection1,intersection2;
      set_intersection(NDash.begin(),NDash.end(),Sua.begin(),Sua.end(),insert
er(intersection1,intersection1.begin()));
      set_intersection(NDash.begin(),NDash.end(),Sub.begin(),Sub.end(),insert
er(intersection2,intersection2.begin()));
      if(intersection1.size()>0 || intersection2.size()>0)
            return false;
      return true;
}
void Graph::colorEdge(Edge&e)
{
      Vertex v,u;
      u=vertexArray[e.start];
      v=vertexArray[e.end];

      set<int>fu,fv;
      for(list<int>::iterator
i=u.adjacencyList.begin();i!=u.adjacencyList.end();i++)
      {
            int col=findColor(u.vertexNumber,*i);
            if(col!=-1)
                  fu.insert(col); // insert the color of Fu into the set.
      }
      for(list<int>::iterator
i=v.adjacencyList.begin();i!=v.adjacencyList.end();i++)
      {
            int col=findColor(v.vertexNumber,*i);
            if(col!=-1)
                  fv.insert(col); // insert the color of Fv into the set.
      }

      /* here we will be finding the union of fu and fv and also the
intersection of fu and fv */
      set<int>fuINTERSECTIONfv,fuUNIONfv;
      set_intersection(fu.begin(),fu.end(),fv.begin(),fv.end(),inserter(fuINT
ERSECTIONfv,fuINTERSECTIONfv.begin()));
      set_union(fu.begin(),fu.end(),fv.begin(),fv.end(),inserter(fuUNIONfv,fu
UNIONfv.begin()));

      if(fuINTERSECTIONfv.size()==0) // no need to check for bichromatic
cycles here. // case 1 of the research paper is covered here.
      {
            e.color=*(findCandidateColors(e).begin()); // for this case we
will definitely get a color that is also valid for the edge 'e'.
      }
      else if(fuINTERSECTIONfv.size()==1) // we need to check for the
Bichromatic cycles // case 2 of the research paper
      {
```

```
            set<int>candidateColors=findCandidateColors(e);
            if(fuUNIONfv.size()==1) // DONE
            {
                  set<int>Sua=S(u.vertexNumber,*(u.adjacencyList.begin()));
                  set<int>newSet; // this newSet will contain all the
candidate colors which will also be valid for the the current Edge

      Sua.insert(findColor(u.vertexNumber,*(u.adjacencyList.begin())));

      set_difference(candidateColors.begin(),candidateColors.end(),Sua.begin(
),Sua.end(),inserter(newSet,newSet.begin()));
                  e.color=*(newSet.begin()); // for this case we will
definitely get a color that is also valid for the edge 'e'.
            }
            else if(fuUNIONfv.size()==2)
            {
                  /* currently working on this. */
                  int count=0;
                  set<int>::iterator i;
                  for(i=candidateColors.begin();i!=candidateColors.end();i++)
                  {
                        if(isCriticalPath(e,*(fuINTERSECTIONfv.begin()),*i))
                              count++;
                        else break;
                  }
                  if(count==candidateColors.size()) // means all the
candidate colors are invalid. some recoloring and exchange is essential.
                  {
                        Vertex x,y; // x is the vertex with degree 1 and y is
the vertex with degree 2
                        if(vertexArray[e.start].adjacencyList.size()==1)
                        {
                              x=vertexArray[e.start];
                              y=vertexArray[e.end];
                        }
                        else
                        {
                              y=vertexArray[e.start];
                              x=vertexArray[e.end];
                        }
                        set<int>newSet;

      set_difference(fuUNIONfv.begin(),fuUNIONfv.end(),fuINTERSECTIONfv.begin
(),fuINTERSECTIONfv.end(),inserter(newSet,newSet.begin()));
                        Edge
e1=findEdge(x.vertexNumber,*(x.adjacencyList.begin()));
                        recolor(e1,*(newSet.begin()));
                        /* now if still the candidate colors are invalid then
there still exists critical paths. */
                        /* to remove these critical paths I need to exchange
the coloring of ya and yb where a,b are neighbors of y. */

                        /* we need to calculate the new color which will be
in the intersection of Fx and Fy*/
                        /* the set of candidate colors will not change here.
*/

      set_intersection(fu.begin(),fu.end(),fv.begin(),fv.end(),inserter(fuINT
ERSECTIONfv,fuINTERSECTIONfv.begin()));
                        int commonColor=*(fuINTERSECTIONfv.begin());
                        count=0;

      for(i=candidateColors.begin();i!=candidateColors.end();i++)
                        {
```

```
	if(isCriticalPath(e,*(fuINTERSECTIONfv.begin()),*i))
					count++;
				else break;
			}
			if(count==candidateColors.size()) // here i need to
apply some color exchange.
			{
				/* here I will exchange the colors of ya and yb
where y is the vertex of degree 2 and a & b are its neighbors. */
				Vertex a,b;
				list<int>::iterator it=y.adjacencyList.begin();
				a=vertexArray[*it];
				it++;
				b=vertexArray[*it];
				Edge
e1=findEdge(y.vertexNumber,a.vertexNumber);
				Edge
e2=findEdge(y.vertexNumber,b.vertexNumber);
				colorExchange(e1,e2);

				/* now both the candidate colors will
definitely be valid for the edge e. */
				e.color=*(candidateColors.begin());
			}
			else
			{
				e.color=*i;
			}
		}
		else // one of the candidate colors is not forming any
critical cycle.
		{
			e.color=*i;
		}
	}
	}
}
Vertex& Graph::findVertexWithDegreeAtMost2(bool *isVertexCovered)
{
	for(int i=0;i<numberOfVertices;i++)
	{
		if(!isVertexCovered[i])
		{
			int deg=0;
			for(list<int>::iterator
it=vertexArray[i].adjacencyList.begin();it!=vertexArray[i].adjacencyList.end(
);it++)
			{
				if(!isVertexCovered[*it])
					deg++;
			}
			if(deg<=2)
				return vertexArray[i];
		}
	}
}
void Graph::arrangeEdges()
{
	bool *isVertexCovered=new bool[numberOfVertices];
	for(int i=0;i<numberOfVertices;i++)
		isVertexCovered[i]=false; // initially no Vertex is covered.

	/* now I need to find a vertex which has degree less than equal to 2 */
```

```
	list<Edge>newEdgeList;

	for(int i=0;i<numberOfVertices;i++)
	{
		Vertex &x=findVertexWithDegreeAtMost2(isVertexCovered);
		isVertexCovered[x.vertexNumber]=true; // now the vertex x is
covered.
		/*
			now traverse the adjacencyList of x (only for the vertices
that are uncovered.)
			find the edge that is formed between x and its adjacency
			let this edge be called e
			enumerate e.
		*/
		for(list<int>::iterator
it=x.adjacencyList.begin();it!=x.adjacencyList.end();it++)
		{
			if(!isVertexCovered[*it])
			{
				Edge e=findEdge(x.vertexNumber,*it);
				newEdgeList.push_back(*(new Edge(e)));
			}
		}
	}
	edgeList.erase(edgeList.begin(),edgeList.end());
	for(list<Edge>::iterator
it=newEdgeList.begin();it!=newEdgeList.end();it++)
	{
		edgeList.push_back(*it);
	}
	delete(isVertexCovered);
}

void Graph::print()
{
	printf("\nThe graph is as follows : \n");
	printf("\nThe number of Vertices in the Graph
= %d\n",numberOfVertices);
	printf("The number of Edges in the Graph = %d\n",numberOfEdges);
	printf("\nThe adjacency List of the Graph is as follows : \n");
	for(int i=0;i<numberOfVertices;i++)
	{
		printf("%d : ",i);
		for(list<int>::iterator
j=vertexArray[i].adjacencyList.begin();j!=vertexArray[i].adjacencyList.end();
j++)
			printf("%d, ",(*j));
		printf("\n");
	}
	printf("\nThe edge List of the Graph is as follows : \n");
	for(list<Edge>::iterator i=edgeList.begin();i!=edgeList.end();i++)
		printf("%d <--> %d\tColor=%d\n",(*i).start,(*i).end,(*i).color);
	printf("\n");
	/*
		Generating the output file
	*/
	FILE* fp;
	fp=fopen("result.txt","w");
	if(fp)
	{
		fprintf(fp,"%d %d\n",numberOfVertices,numberOfEdges);
		for(list<Edge>::iterator
it=edgeList.begin();it!=edgeList.end();it++)
			fprintf(fp,"%d %d %d\n",(*it).start,(*it).end,(*it).color);
```

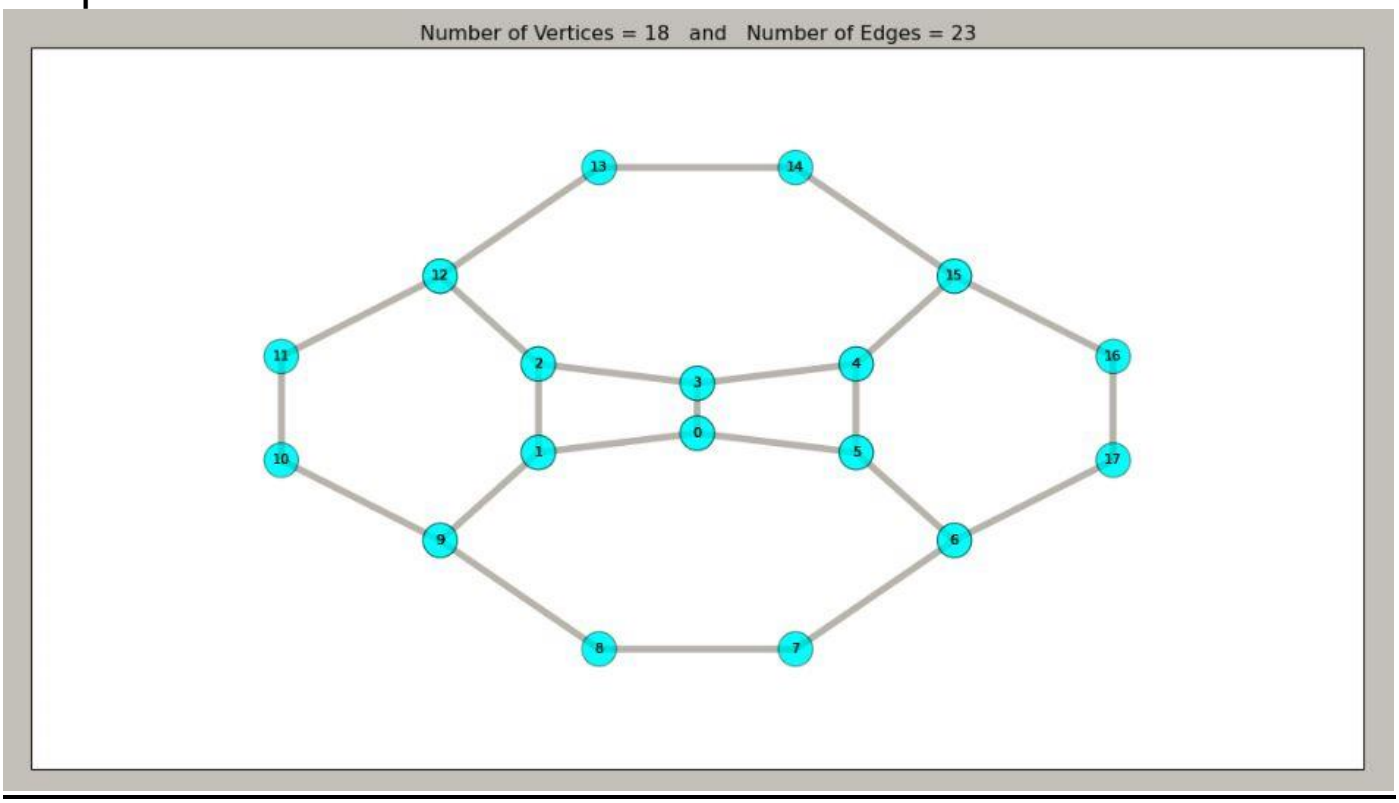
```
            fclose(fp);
      }
      else printf("There was some error in opening the file\n");
}
void Graph::deleteGraph()
{}

int main()
{
      system("clear");
      printf("\nYou have the following options : \n");
      printf("1. Give the input in Adjacency Matrix form\n");
      printf("2. Give the input as each edge of the Graph\n");
      printf("3. Give the input from a file\n");
      printf("\nEnter your Choice : ");
      int choice;
      scanf("%d",&choice);
      if(choice==1)
      {
            int n,e;
            printf("\nEnter the number of Vertices : ");
            scanf("%d",&n);
            printf("Enter the number of Edges : ");
            scanf("%d",&e);
            Graph graph(n,e);
            printf("Enter the Adjacency Matrix of the graph : \n");
            for(int i=0;i<n;i++)
            {
                  for(int j=0;j<n;j++)
                  {
                        int temp;
                        scanf("%d",&temp);
                        if(i<j && temp==1)
                              graph.addEdge(i,j);
                  }
                  //graph.addEdge(s,d);
            }

            graph.arrangeEdges();
            // this is a test to print the edgelist
            for(list<Edge>::iterator
it=graph.edgeList.begin();it!=graph.edgeList.end();it++)
                  graph.colorEdge(*it);
            graph.print();
            graph.deleteGraph();
      }
      else if(choice==2)
      {
            int n,e;

            printf("\nEnter the number of Vertices : ");
            scanf("%d",&n);
            printf("Enter the number of Edges : ");
            scanf("%d",&e);
            Graph graph(n,e);
            //printf("Graph initialization complete\n");
            printf("Enter the source and destination of %d number of edges :
\n",e);
            for(int i=0;i<e;i++)
            {
                  int s,d;
                  scanf("%d%d",&s,&d);
                  graph.addEdge(s,d);
            }
```

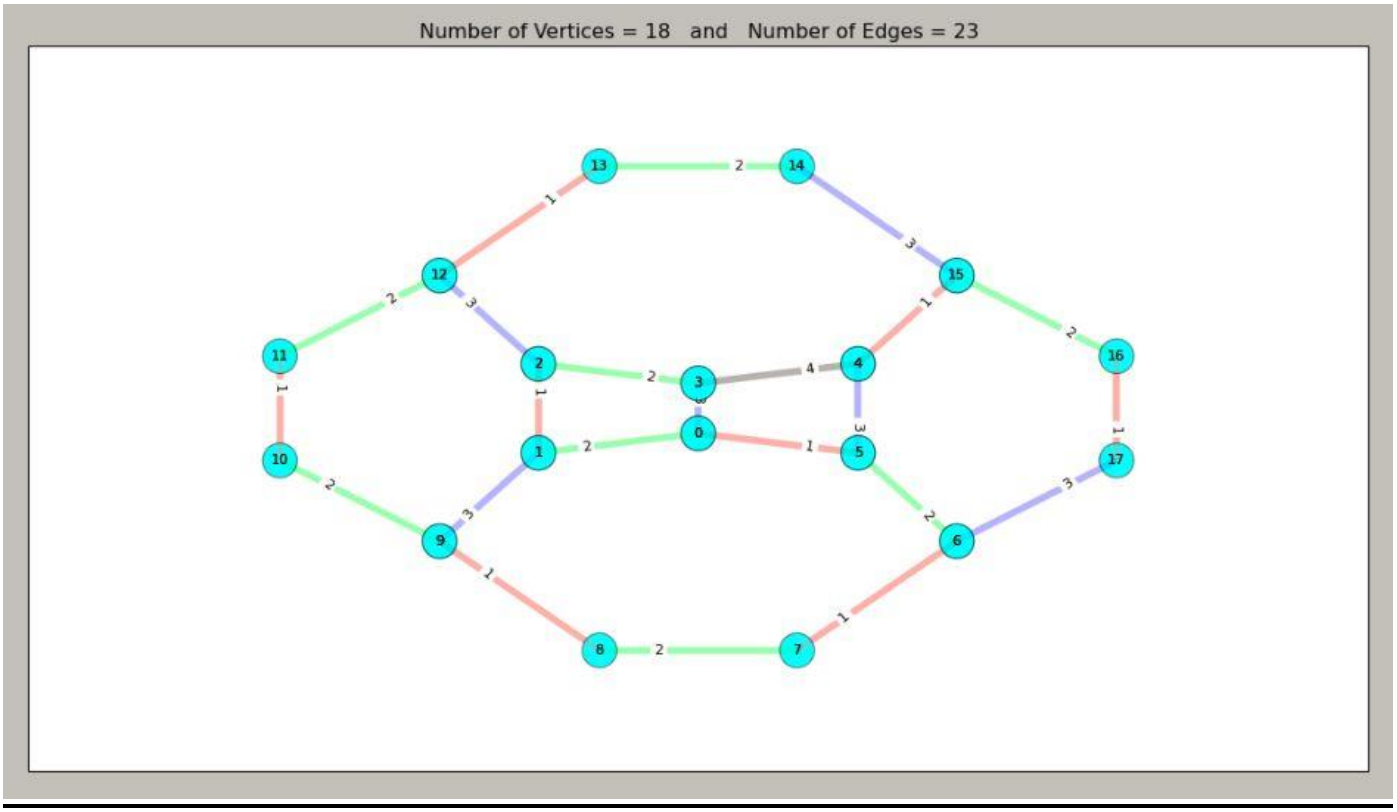
```
            graph.arrangeEdges();
            // this is a test to print the edgelist
            for(list<Edge>::iterator
it=graph.edgeList.begin();it!=graph.edgeList.end();it++)
                  graph.colorEdge(*it);
            graph.print();
            graph.deleteGraph();
      }
      else if(choice==3)
      {
            FILE*fp;
            printf("\nEnter the file name for input : ");
            char fileName[100];
            scanf("%s",fileName);
            fp=fopen(fileName,"r");
            if(fp)
            {
                  int n,e;
                  fscanf(fp,"%d%d",&n,&e);
                  Graph graph(n,e);
                  //printf("Graph initialization complete\n");
                  for(int i=0;i<e;i++)
                  {
                        int s,d;
                        fscanf(fp,"%d%d",&s,&d);
                        graph.addEdge(s,d);
                  }

                  graph.arrangeEdges();
                  // this is a test to print the edgelist
                  for(list<Edge>::iterator
it=graph.edgeList.begin();it!=graph.edgeList.end();it++)
                        graph.colorEdge(*it);
                  graph.print();
                  graph.deleteGraph();
                  fclose(fp);
            }
            else printf("There was some error in opening the file.\n");
      }
      return 0;
}
/*************************************************************************
/
python file to draw graph
/*************************************************************************
/
import networkx as nx
import matplotlib.pyplot as plt

def draw_graph(edges,vertices,graph, labels=None, graph_layout='shell',
               node_size=600, node_color='cyan', node_alpha=0.3,
               node_text_size=10,
               edge_color='black', edge_alpha=0.3, edge_tickness=5,
               edge_text_pos=0.3,
               text_font='sans-serif'):

      # create networkx graph
      G=nx.Graph()

      # add edges
      for edge in graph:
            G.add_edge(edge[0], edge[1])
```

```
      # these are different layouts for the network you may try
      # shell seems to work best
      if graph_layout == 'spring':
            graph_pos=nx.spring_layout(G)
      elif graph_layout == 'spectral':
            graph_pos=nx.spectral_layout(G)
      elif graph_layout == 'random':
            graph_pos=nx.random_layout(G)
      else:
            graph_pos=nx.shell_layout(G)


    # draw graph
      G0=nx.Graph();
      G1=nx.Graph();
      G2=nx.Graph();
      G3=nx.Graph();
      graph0=[]
      graph1=[]
      graph2=[]
      graph3=[]
      for i in range(0,len(labels)):
            if labels[i]==1:
                  G0.add_edge(graph[i][0],graph[i][1])
                  graph0.append(graph[i])
            if labels[i]==2:
                  G1.add_edge(graph[i][0],graph[i][1])
                  graph1.append(graph[i])
            if labels[i]==3:
                  G2.add_edge(graph[i][0],graph[i][1])
                  graph2.append(graph[i])
            if labels[i]==4:
                  G3.add_edge(graph[i][0],graph[i][1])
                  graph3.append(graph[i])

      nx.draw_networkx_nodes(G0,graph_pos,node_size=node_size,alpha=node_alph
a, node_color=node_color)
      nx.draw_networkx_nodes(G1,graph_pos,node_size=node_size,alpha=node_alph
a, node_color=node_color)
      nx.draw_networkx_nodes(G2,graph_pos,node_size=node_size,alpha=node_alph
a, node_color=node_color)
      nx.draw_networkx_nodes(G3,graph_pos,node_size=node_size,alpha=node_alph
a, node_color=node_color)

      nx.draw_networkx_edges(G0,graph_pos,width=edge_tickness,alpha=edge_alph
a,edge_color='#ff0000') # red
      nx.draw_networkx_edges(G1,graph_pos,width=edge_tickness,alpha=edge_alph
a,edge_color='#00ff00') # green
      nx.draw_networkx_edges(G2,graph_pos,width=edge_tickness,alpha=edge_alph
a,edge_color='#0000ff') # blue
      nx.draw_networkx_edges(G3,graph_pos,width=edge_tickness,alpha=edge_alph
a,edge_color='#000000') # black
      '''
      nx.draw_networkx_edges(G0,graph_pos,width=edge_tickness,alpha=edge_alph
a,edge_color=edge_color) # red
      nx.draw_networkx_edges(G1,graph_pos,width=edge_tickness,alpha=edge_alph
a,edge_color=edge_color) # green
      nx.draw_networkx_edges(G2,graph_pos,width=edge_tickness,alpha=edge_alph
a,edge_color=edge_color) # blue
      nx.draw_networkx_edges(G3,graph_pos,width=edge_tickness,alpha=edge_alph
a,edge_color=edge_color) # black
      '''
      nx.draw_networkx_labels(G0,
graph_pos,font_size=node_text_size,font_family=text_font)
```

```
      nx.draw_networkx_labels(G1,
graph_pos,font_size=node_text_size,font_family=text_font)
      nx.draw_networkx_labels(G2,
graph_pos,font_size=node_text_size,font_family=text_font)
      nx.draw_networkx_labels(G3,
graph_pos,font_size=node_text_size,font_family=text_font)

      if labels is None:
            labels = range(len(graph))

      labels0=[]
      labels1=[]
      labels2=[]
      labels3=[]
      for i in range(0,len(graph0)):
            labels0.append(1)
      for i in range(0,len(graph1)):
            labels1.append(2)
      for i in range(0,len(graph2)):
            labels2.append(3)
      for i in range(0,len(graph3)):
            labels3.append(4)

      edge_labels0 = dict(zip(graph0, labels0))
      edge_labels1 = dict(zip(graph1, labels1))
      edge_labels2 = dict(zip(graph2, labels2))
      edge_labels3 = dict(zip(graph3, labels3))


      nx.draw_networkx_edge_labels(G0, graph_pos,
edge_labels=edge_labels0,label_pos=edge_text_pos)
      nx.draw_networkx_edge_labels(G1, graph_pos,
edge_labels=edge_labels1,label_pos=edge_text_pos)
      nx.draw_networkx_edge_labels(G2, graph_pos,
edge_labels=edge_labels2,label_pos=edge_text_pos)
      nx.draw_networkx_edge_labels(G3, graph_pos,
edge_labels=edge_labels3,label_pos=edge_text_pos)


      frame1=plt.gca()
      frame1.axes.get_xaxis().set_visible(False)
      frame1.axes.get_yaxis().set_visible(False)
      plt.title("Number of Vertices = "+str(vertices)+"   and   Number of
Edges = "+str(edges))
      # show graph
      plt.show()


inFile=open("result.txt",'r')
vertices,edges=map(int,inFile.readline().split())
graph=[]
edgeLabels=[]
for i in range(0,edges):
      a,b,c=map(int,inFile.readline().split())
      graph.append((a,b),)
      edgeLabels.append(c)
draw_graph(edges,vertices,graph, edgeLabels)
```

Output


Number of Vertices = 18 and Number of Edges = 23
13
14
12
15
11
2
3
4
16
0
10
1
5
17
9
6
8
7


Number of Vertices = 18 and Number of Edges = 23
13
14
12
15
11
2
3
4
16
0
10
1
5
17
9
6
8
7

Summary

Despite the fact that acyclic edge coloring is computationally hard problem, then also it is a very active topic in research area of Graph Theory. The techniques like Recoloring an edge and exchanging the color of two edges prove to be very helpful and handy while proving any extended and generalised version discussed in this report. The conjecture is still open and it has been proved for some special cases of the graph. More clever and sophisticated techniques are needed to prove the conjecture or to disprove it.