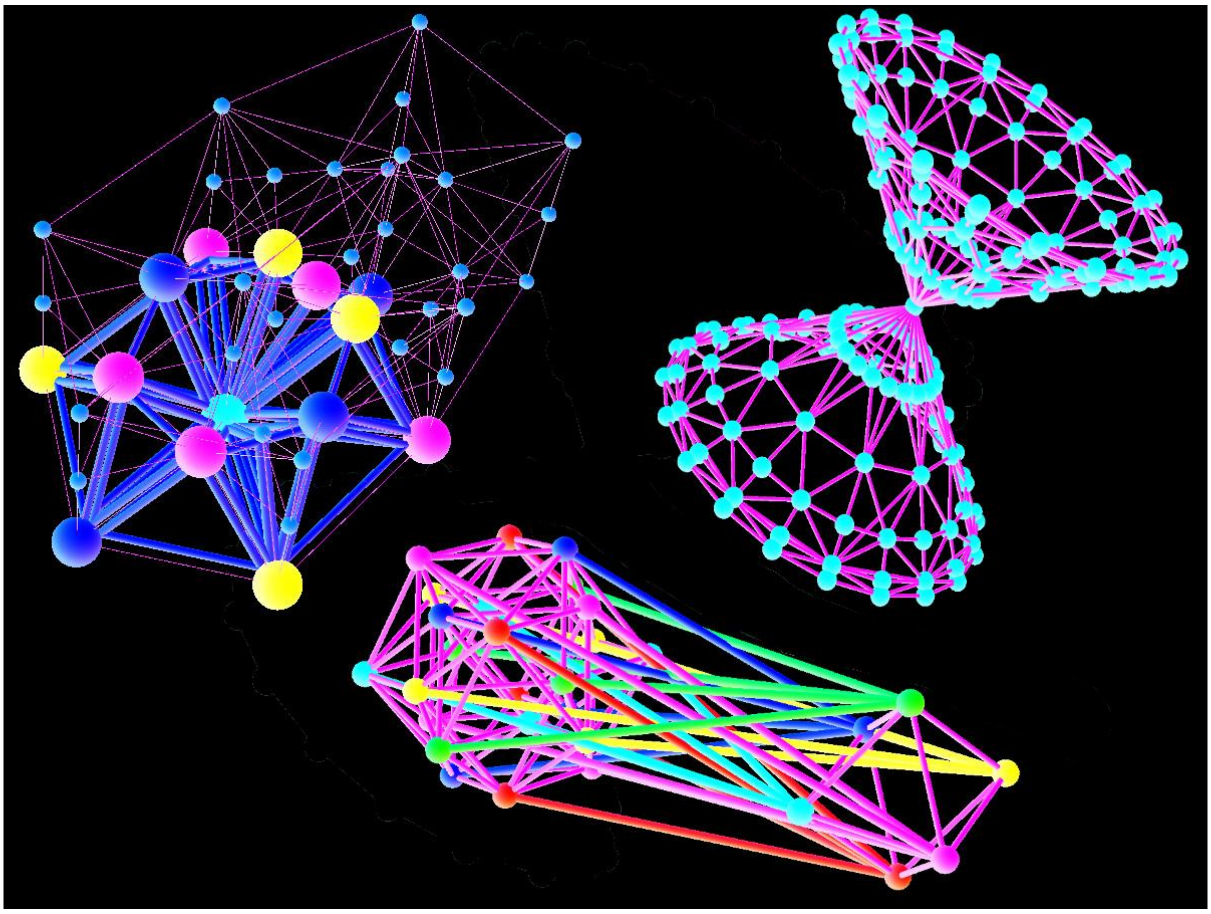
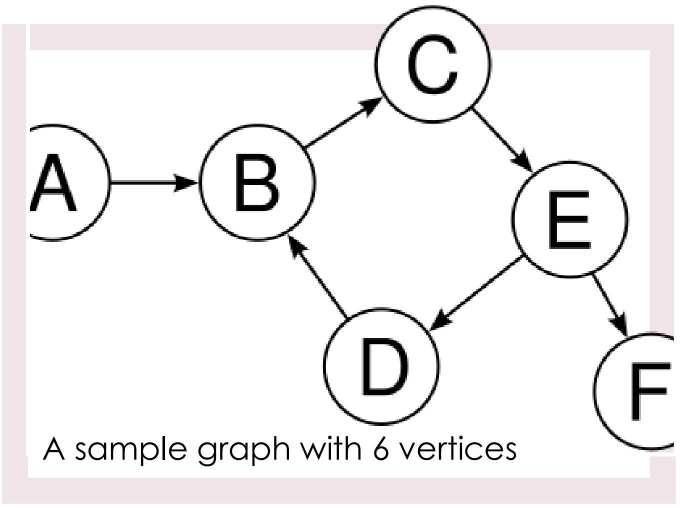


GRAPH COLOURING ALGORITHMS

A Project by SHREYANSH AGRAWAL

What is a **GRAPH**?

A graph data structure consists of a finite set of *vertices* or *nodes* or *points*, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as *edges*, *arcs*, or *lines* for an undirected graph and as *arrows*, *directed edges*, *directed arcs*, or *directed lines* for a directed graph. The vertices may be part of the graph structure, or may be external entities represented by integer indices or references.



A sample graph with 6 vertices

Vertex

A vertex is the data type of which the graph is made up. The vertices are interconnected to create the graph. A vertex contains all the necessary data needed for the graph. A vertex also contains the list of all the vertices that have an edge with it.

Edges

An edge is a pair of vertices. If there is an edge between two vertices, it denotes that both the vertex are adjacent. An edge represents the physical connections in the Graph. It represents that there is a factor which the two vertices have in common.

A Graph consists of set of vertices and edges connecting those vertices.

COLOURING of a Graph.

One of the most typical problems related with the Graph is the colouring of the graph. Graph colouring is nothing but a simple way of labelling graph components such as vertices, edges and regions under some constraints. While graph colouring the constraints that are set on the graph are colour, order of colouring, way of assigning colours, etc. The different ways of colouring a graph are:

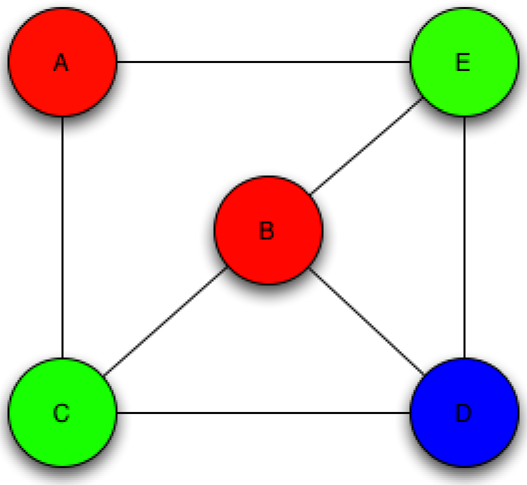
- Vertex colouring
- Edge colouring
- Face colouring

Vertex colouring

Vertex colouring is the most common graph colouring problem. It is an assignment of the colour to the vertices of a graph such that no two adjacent vertices have the same colour. In other words, there should not be an edge between two vertices of the same colour or label.

Chromatic Number

The minimum number of colours required for vertex colouring of a graph is called the chromatic number of the graph. The chromatic number of a graph 'G' is denoted by x(G).
x(G) = 1 if and only if 'G' is a null graph ('G' has vertices but no edges). If 'G' is not a null graph them x(G) >= 2.
A graph 'G' is said to be n-coverable if there is a colouring that uses at most 'n' colours.



Sample vertex colouring of a graph.

Applications

The graph colouring problem has huge number of applications. Some of them are listed below,

- Making schedule or time table
- Mobile radio frequency assignment
- Solving Sudoku
- Register allocation
- Bipartite Graphs
- Map colouring

There can be many more applications.

Algorithms

The three graph colouring algorithms that I have tried in this project are:

- Greedy colouring algorithm
- Backtracking algorithm
- Brute-force algorithm

A brief discussion on each algorithm is done separately.

The **PROBLEM…**

The main problem underlying this project was to create a program which could help in generating a feasible academic time-table for IISER. Each student can choose various number of courses in a semester. The challenge was to create such a program which could, based on the selection of courses of students, allocate separate slots to separate courses, such that, no two subjects selected by the same student fall on the same spot. This way the set time table would be optimum for all students.

What the program does, is that it allocates different slots to different courses based on the choice of students. This would determine the maximum number of slots needed in a week to fit all courses. Then the week could be divided into separate slots and each slot could be given a number. The subjects would then be paced on different slots based on the numbers. This program, however, can be used just to check which subjects cannot be placed in the same slot. This can also check whether it is possible or not to fit all the subjects in some predefined number of slots.

The **SOLUTION**…

It is trivial that this problem could very well be handled using graphs and graph colouring algorithms. Here is how, we know we have various subjects, and many subjects will have common students. The main aim is to allocate different slots to subjects with a common student. This problem can be represented as a graph where each subject is a vertex of the graph and a common student is the edge between two vertices (or subjects).

The first phase was to create a graph.

The various ways in which a graph could be stored are:

- **Adjacency list**: Vertices are stored as records or objects, and every vertex stores a list of adjacent vertices. This data structure allows the storage of additional data on the vertices. Additional data can be stored if edges are also stored as objects, in which case each vertex stores its incident edges and each edge stores its incident vertices.
- **Adjacency matrix:** A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices. Data on edges and vertices must be stored externally. Only the cost for one edge can be stored between each pair of vertices.
- **Incidence matrix:** A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges. The entries indicate whether the vertex at a row is incident to the edge at a column.

Adjacency lists were used in this program as they are more efficient over Adjacency matrices and Incidence matrices.

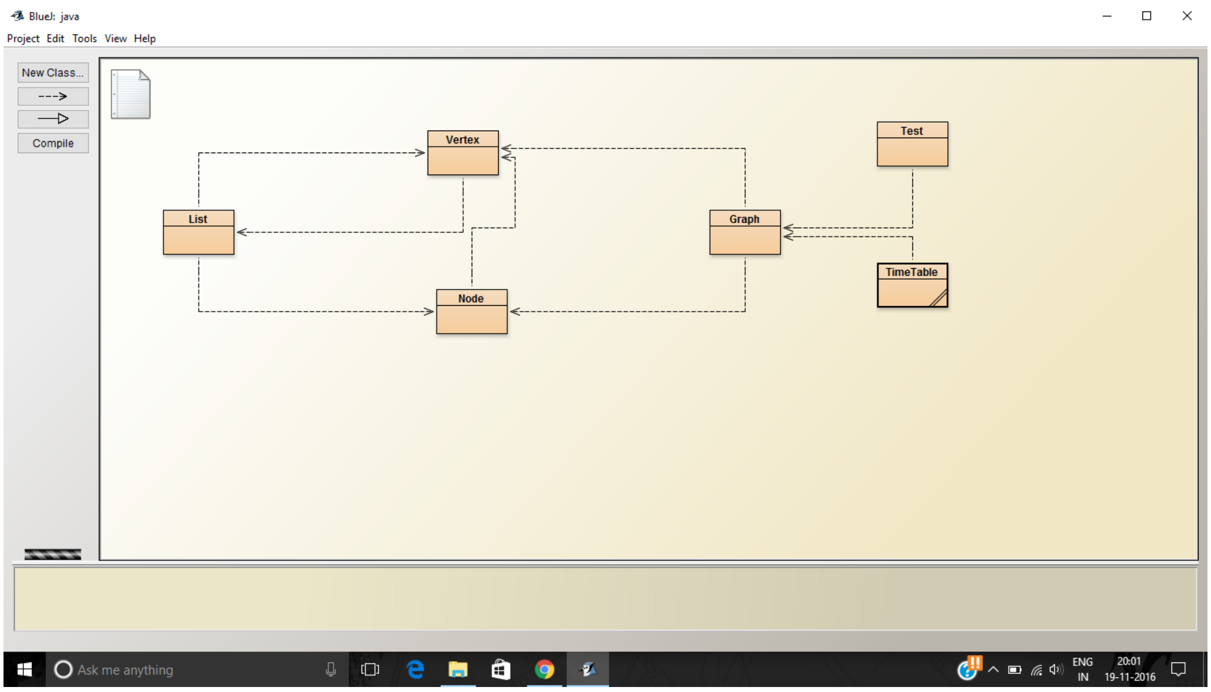
The classes that were used to implement this graph are:

- Node
- List
- Vertex
- Graph

The main class of the project is "TimeTable" which reads the input from a file, converts it to a graph and calls the colouring functions as per the choice of the user. All the classes are written in the file "TimeTable.java".

Name	Description
Class:	
Node	Used to create a node for the linked list.
Data members:	
Vertex v	Stores the reference of the vertex which is in the list.
Node next	Stores the link of the next node in the list.
Node prev	Stores the link of the previous node in the list.
Constructor:	
Node(Vertex v)	Initialises the node with ‘v’.
Class:	
List	A linked list to store all the vertices adjacent to a particular vertex.
Data members:	
Node ptr	Pointer to store temporarily the reference of a node.
Node head	Stores the reference of the first node.
Node tail	Stores the reference of the last node.
Functions:	
void add(Vertex v)	Appends ‘v’ to the pre-existing linked list or creates a new one if ‘v’ is the first vertex to be added to the list.
Boolean search(String s)	Searches for a node which has a Vertex with name ‘s’ in the linked list. If such a node exists, returns true, returns false otherwise.
Class:	
Vertex	A data type to store the vertices of the graph.
Data members:	
String name	Stores the name of the vertex.
int clr	Stores the colour assigned to the vertex.
List edges	Stores the adjacency list.
Constructors:	
Vertex(String n)	Initialises a Vertex with name ‘n’ and colour -1.
Vertex(String n, int c)	Initialises a Vertex with name ‘n’ and colour ‘c’.
Functions:	
Void addEdge(Vertex v)	Add ‘v’ to the adjacency list of the current object.
Class:	
Graph	Creates a graph data structure to store the various vertices and colours the vertices in such a way that no two vertices get the same colour.
Data members:	
Vertex[] subs	Stores the list of all the vertices of the graph.
int ind	Stores the index of the last vertex in the array ‘subs’.
Constructors:	
Graph(int n)	Initialises the graph setting the length of subs to be ‘n’.
Functions:	
Void add(String name)	Adds a vertex with the name ‘name’ to the graph.

Void addEdge(String s1, String s2)	Adds an edge between the vertices with the name ‘s1’ and ‘s2’.
boolean checkColoring()	Returns true if the current colour assigned to the vertices are valid, otherwise returns false.
int getColors()	Returns the total number of colours used to colour the vertices.
void color()	Colour the vertices of the graph using the minimum number of colours.
void print()	Prints the vertices of the graph and the colour allotted to them.
void printGraph()	Prints the vertices of the graph along with the list of the vertices they have an edge with.
boolean colorGreedy()	Colours the graph using the greedy colouring algorithm. Returns true if the colouring if possible, otherwise returns false.
void reset()	Resets the colours of the vertices to -1.
boolean check(String n)	Searches the array ‘subs’ for a vertex with name ‘n’. Returns true if it is found, otherwise returns false.
boolean colorBack(int k, int c)	Tries to colour the graph in ‘c’ colours. Returns true if such colouring is possible, and false if not. k is the variable to control recursion, it runs from k = 0 to ‘ind’.



A diagram showing which class uses reference which other class, using arrows.

The **ALGORITHMS…**

Four different colouring algorithms were tried with the program. The task was to select the best algorithm, i.e. the algorithm that would take the minimal time and give the most optimal colouring scheme for the graph. All the algorithms were compared using different graphs, and then the best one was implemented.

The Greedy Colouring Algorithm:

It colours the graph using a hit and trial method. Following is a step wise algorithm:

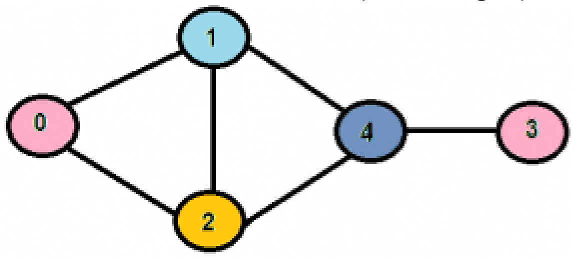
Step 1: START
Step 2: Select the first colour (say 0).
Step 3: If the first vertex is not already coloured, colour it with the selected colour.
Step 4: If the next vertex does not have a colour go to next step, otherwise go to step 6.
Step 5: Check whether this colour can be assigned to the next vertex in the graph. For this, traverse the edges of that vertex checking if any vertex in that list already have that colour, if not then the colour can be safely assigned to the vertex.
Step 6: Do the above step 4 for all the vertices in the graph, and assign colour only to those vertices which do not already have a colour.
Step 7: Increment the value of colour by 1 and go back to step 3. If the value of colour becomes more that the number of vertices, then that that means the colouring is over, so go to next step.
Step 8: END

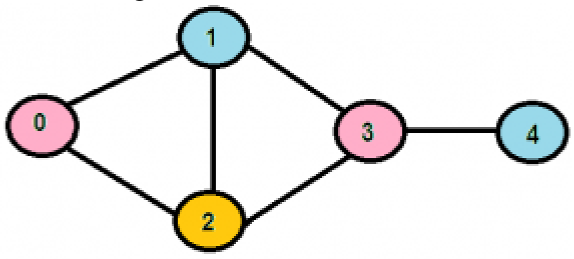
Pseudo code:

```
➢ n = number of vertices
➢ for c = 0 to n
➢   for i = 0 to n
➢     If vertex[i].colour == -1 then
➢       If edges.colour != c then
➢         vertex[i].colour = c
➢   next i+1
➢ next c+1
```

Analysis:

The greedy colouring algorithm has a time complexity of O(n^2). The algorithm is highly efficient in terms of time taken to colour the graph, but it does not always give the optimum colouring scheme. The colour depends also on the order in which the vertices are visited. Here is one example of a graph where this algorithm fails:





In the first graph the algorithm uses 4 colours, while we see in the second graph that the minimum required colour is 4. This could have been achieved just b changing the order in which the graphs were visited. This algorithm could be used with a sparse graph, but usually with a dense graph, the difference from the minimum colour can be greater. However, this algorithm can be used to get a rough estimate of the number of colours required to colour the graph.

Brute-force Algorithm:

This algorithm generates all possible colouring combinations, checks them and returns the most optimum one.

Step 1: START
Step 2: Set k to 1 and n to number f vertices.
Step 3: Loop from 0 to k^{n-1} and in each iteration convert the value of the loop variable to base k.
Step 4: Note that every digit in the base k number will range from 0 to k-1. So there are n values each ranging from 0 to k-1. This can be treated as a colouring of the n vertices of the graph using k colours.
Step 5: Check whether this colouring is valid or not.
Step 6: If colouring is valid then exit the loop, otherwise increment k by 1 and repeat from step 3. Go back as long as k is not greater than n, otherwise exit the loop.
Step 7: END

Pseudo code:

```
➢ n = number of vertices
➢ for k = 1 to n
➢   for i = 0 to k^(n-1)
➢     x = i in base k
➢     subs[j].colour = jth digit of x
➢     if Colouring is valid then
➢       exit
➢     next i+1
➢ next k+1
```

Analysis:

The code is very poor on time. It has a time complexity of $O(n^n)$. The algorithm grows exponentially with number of vertices. Moreover, time is not the only problem with this algorithm. As we are converting bases, we are limited to a total availability of 36 bases (0 to 1 and A to Z). So, the base of decimal number cannot be converted once we go over 36 vertices. The other problem is with the storage of the number 'k^{n-1}'. The largest data type in java capable of storing large values is 'long'. Long too has a maximum bound of 2^{64} or 1.6E19. So, even long cannot store huge values of k^{n-1}. Therefore, this is a very poor algorithm.

Leaving the time complexity, the other two problems can be removed using a recursive approach. Here's how:

Step 1: START
Step 2: Assign colours from 0 to n to the first vertex.
Step 3: During each assigning, go to the next vertex and repeat from step 2.

Step 4: If at the last vertex, then check if the colouring is valid. Exit if valid otherwise continue assigning colours.

Step 5: END

Analysis:

While this algorithm erodes the problem of storage and limited base conversion, time is still a problem. This algorithm has the same time complexity as the previous one, where the algorithm grows exponentially with the number of vertices. However, here we cannot also be certain that the lower colouring scheme would be tried before any other colouring.

Following is a graph on which both the algorithms were tested:

A - B C G H I K F J

B - A C E H I J K

C - A B D F G J K

D - C F H K G J

E - B G H I J

F - D A C G J K I

G - A E F C D I K H

H - A B D E G J K I

I - A B E G F H K

J - B F H A C D E K

K - A D F G H I B C J

It is a sparse graph of 11 vertices.

Colouring given by first algorithm:

A, 1

B, 2

C, 3

D, 1

E, 0

F, 2

G, 4

H, 3

I, 5

J, 4

K, 0

Time taken: 32555 milliseconds

Colouring given by second algorithm:

A, 0

B, 1

C, 2

D, 0

E, 0

F, 1

G, 3

H, 2

I, 4

J, 3

K, 5

Time taken: 49282 milliseconds

Code used to create the graph:

```
Graph g = new Graph(35);
    long i,f;
    for(char x = 'A'; x <= 'K' ; x++)
    g.add(Character.toString(x));
    for(char x = 'A'; x <= 'K'; x++)
    {
      for(char y = 'A'; y <= 'K'; y++)
      {
        double d = Math.random();
        if(d>0.5)
        g.addEdge(Character.toString(x),Character.toString(y));
      }
    }
```

Comment:
We can clearly see that the recursive algorithm took more time to colour the graph than normal one. But when the same algorithm was used to create a graph of 12 vertices, the normal algorithm as not able to colour it at all, while the recursive algorithm took 7 minutes to accomplish the task. However, as we went to creating a 14 vertex graph, both the algorithms failed in colouring it within 30 minutes. Clearly, the brute-force algorithm cannot be used for our problem.

Backtracking algorithm:

This one is more like an extension to the greedy colouring algorithm, where a graph is tried to be coloured with a pre specified number of colours using hit and trial method.

Step 1: START
Step 2: Let 'c' be the maximum colours we can use.
Step 3: Assign the first colour to the first vertex.
Step 4: Now follow the following algorithm for other vertices
Step 5: Assign the lowest possible colour to each vertex recursively
Step 6: If no colour less than 'c' is possible, then go back to previous vertex and change its colour.
Step 7: If all the vertices are successfully coloured, then colouring with 'c' colours is possible.
Step 8: If the backtracking reaches back to the first vertex, then colouring with 'c' colours is not possible.
Step 9: END

Pseudo code:

```
➢ c = number of colours to be checked.
➢ Vertex[0].colour = 0
➢ If all colours are assigned then
➢   Return true
➢ Else
➢   Trying all possible colours, assign a colour to the vertex.
➢   If colour assignment is possible, then assign colour to the next vertex recursively
➢   Else go back to previous vertex.
➢   If this is the first vertex, return false
➢   Else assign a different colour to this vertex
```

Analysis:

To test this code, first a graph was created and coloured using greedy colouring. Then, the backtracking was tries with one less number of colours. The result was absolutely instantaneous.

The Graph:

A - C F H I K N S E G L P Q

B - C D H I J K L M N P R S E F G Q T

C - A B D E G H I J K N O P Q S T R

D - B C E F G H I J K L M P Q R S T N O

E - C D A B G J L O R S T F H I K M P

F - A D B E I J K L M N P H Q R

G - C D E A B H I J K M Q S T N O P

H - A B C D G E F J K M N O P Q R S T I L

I - A B C D F G E H J K L M Q S T N P R

J - B C D E F G H I L M O P Q R T N S

K - A B C D F G H I E M N O Q S T L

L - B D E F I J A H K N P R S T O Q

M - B D F G H I J K E N O P Q S T R

N - A B C F H K L M D G I J O R S T

O - C E H J K M N D G L P Q R T

P - B C D F H J L M O A E G I S T Q

Q - C D G H I J K M O A B F L P S R T

R - B D E H J L N O C F I M Q T

S - A B C D E G H I K L M N P Q J T

T - C D E G H I J K L M N P R S B O Q

Colours taken by greedy colouring: 12

Backtracking with 11 colours: true

Time taken: 0

The Final Algorithm:

Here is the final algorithm which passed the test for both time and minimum colours:

Step 1: START
Step 2: Colour the graph using greedy colouring algorithm.
Step 3: Get the total number of colours used to colour the graph.
Step 4: Reset the colours of vertices.
Step 5: Check with the backtracking algorithm if the graph can be coloured with one less number of colours.
Step 6: If yes, then check again with one lesser.
Step 7: If no, then the previous colouring scheme was the optimum scheme.
Step 8: END

Sample graphs:

Graph 1:

A - B C E G H I J K L N O P Q R D F S T

B - A C D F G I J K N O Q R S T E H L P

C - A B G H I J K L N O P R T D E M

D - B A C E F H I L M O Q R S G K P

E - A D B C F G M N P Q S T H I J K

F - B D E A H J M P R T K L N

G - A B C E D H J K M N O T P Q S

H - A C D F G B E K M R S T N O P Q

I - A B C D E K L N P Q S T R

J - A B C F G E K M N P Q R T S

K - A B C G H I J D E F O Q R T L N P

L - A C D I B F K M N O Q S R

M - D E F G H J L C N O Q R S P

N - A B C E G I J L M F H K S T O P Q

O - A B C D G K L M H N P Q S T

P - A C E F I J O B D G H K M N Q R S T

Q - A B D E I J K L M O P G H N R S T

R - A B C D F H J K M P Q I L S T

S - B D E H I L M N O Q R A G J P T

T - B C E F G H I J K N O Q S A P R

Final colouring:

A, 0

B, 1

C, 2

D, 3

E, 4

F, 2

G, 5

H, 6

I, 5

J, 6

K, 7

L, 6

M, 0

N, 3

O, 4

P, 9

Q, 2

R, 4

S, 7

T, 8

Colours taken: 10

Colours taken by greedy colouring: 12

Time taken: 0

Graph 2:

A - B C D E F G H I K L M P Q R W X Y Z T

B - A C D E H J K L P Q R T U V W X Y Z F M O

C - A B D H I J K L M P Q R S T U V X Y Z E G N O

D - A B C F G H I J K L M N Q S W Z O U Y

E - A B C F G H I J K L N P Q T U V Y Z O R S W X

F - A D E B G H J K L M N O P R T U V W Z I Q

G - A D E F C H J K L M R S T V W X N P Q Y

H - A B C D E F G I K N O P Q R T U V W Y Z J L M S X

I - A C D E H F J K L O P Q R X Z S T

J - B C D E F G I H K L N O P Q U W X M S T V Y Z

K - A B C D E F G H I J L N O Y Z M P Q T U X

L - A B C D E F G I J K H O P Q S T V W Z M N R U

M - A C D F G B H J K L N P Q R S U V W Y Z T X

N - D E F H J K M C G L R S T U Y Z P Q V W X

O - F H I J K L B C D E P S V Y Z R T U W X

P - A B C E F H I J L M O G K N S T U W X Y Z Q

Q - A B C D E H I J L M F G K N P R T U W Y V X

R - A B C F G H I M N Q E L O T U V W Z

S - C D G L M N O P E H I J V W X Y Z

T - B C E F G H L N P Q R A I J K M O W Y Z

U - B C E F H J M N P Q R D K L O V Y X Z

V - B C E F G H L M O R S U J N Q Y Z X

W - A B D F G H J L M P Q R S T E N O Y Z X

X - A B C G I J P S E H K M N O Q U V W Y Z

Y - A B C E H K M N O P Q S T U V W X D G J Z

Z - A B C D E F H I K L M N O P T V W X Y J R S U

Final colouring:

A, 0

B, 1

C, 2

D, 3

E, 3

F, 2

G, 1

H, 4

I, 1

J, 0

K, 5

L, 6

M, 7

N, 8

O, 7

P, 9

Q, 10

R, 9

S, 11

T, 11

U, 11

V, 5

W, 5

X, 6

Y, 12

Z, 10

Colours taken: 13

Colours taken by greedy colouring: 14

Time taken: 0