Genetic Algorithm

(MT21104)

Term Project

Scheduling Exam Timetable Using Graph Coloring

Rohit Pasumarty 19IM10038

Mridul Gupta 19IM30025

Introduction

Genetic Algorithm (GA) is a search-based optimization technique based on the principles of Genetics and Natural Selection. It is frequently used to find optimal or near-optimal solutions to difficult problems and is used to solve optimization problems, in research, and in machine learning.

Nature has always been a great source of inspiration to all mankind. Genetic Algorithms (GAs) are search based algorithms based on the concepts of natural selection and genetics. GAs are a subset of a much larger branch of computation known as Evolutionary Computation.

GAs were developed by John Holland and his students and colleagues at the University of Michigan, most notably David E. Goldberg and has since been tried on various optimization problems with a high degree of success.

In GAs, we have a population of possible solutions to the given problem. These solutions then undergo crossover and mutation (like in natural genetics), producing new children, and the process is repeated over various generations. Each individual (or candidate solution) is assigned a fitness value (based on its objective function value) and the fitter individuals are given a higher chance to mate

and yield more “fitter” individuals. In this way, we keep “evolving” better individuals or solutions over generations, till we reach a stopping criterion.

There are many advantages of using Genetic Algorithms. It does not require any calculation of derivative. It is faster and optimizes both continuous and discrete functions and also multi-objective problems. It is more useful when the search space is very large and there are a large number of parameters involved.

On the other hand, there are some limitations also such as fitness value is calculated repeatedly which might be computationally expensive for some problems and if not implemented properly, the GA may not converge to the optimal solution.

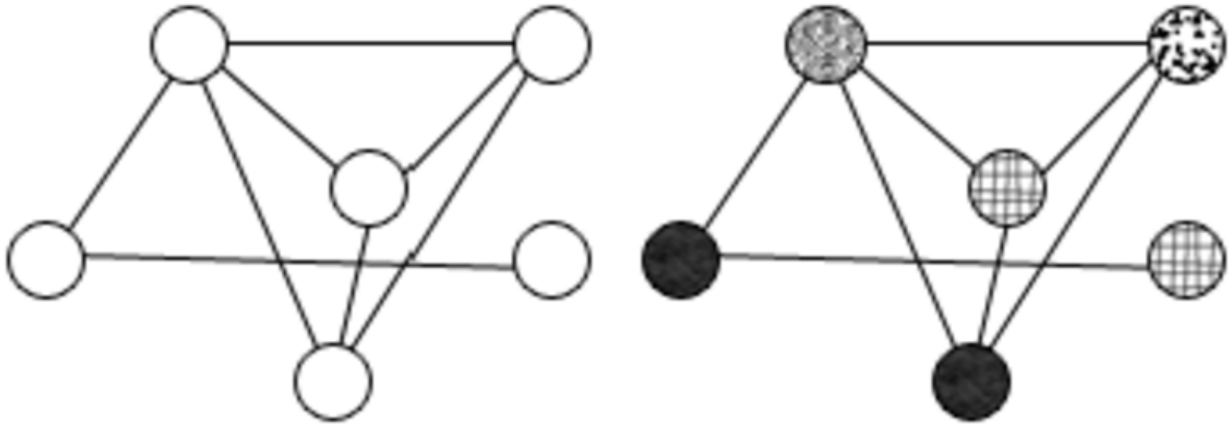
Problem Statement

In a college, there are many students who take courses from various departments to pursue their interests in other fields. As a result, there are many students from different departments who end up taking common courses. So the problem is to schedule their exams so that the exams of the courses taken by the students do not conflict with the other courses and also to keep the slots for the exams as minimum as possible.

Graph Colouring

Graph colouring is a way of colouring the vertices of a graph such that no two adjacent vertices share the same colour. It is also known as vertex colouring.

An example of graph colouring is shown below-



The smallest number of colours needed to colour a graph *G* is called its chromatic number and is often denoted χ(*G*). A graph that can be assigned a (proper) *k-colouring* is *k-colourable*, and it is *k*-chromatic if its chromatic number is exactly *k*. A subset of vertices assigned to the same colour is called a *colour class*, every such class forms an independent set. Thus, a *k-colouring* is the same as a partition of the vertex set into *k* independent sets, and the terms *k-partite* and *k-colourable* have the same meaning.

The chromatic polynomial counts the number of ways a graph can be coloured using no more than a given number of colours. If k={} and P(G,k) is the number of possible solutions for colouring the graph with G with k colours, then

X(G)=min{k:P(G,k)>0}

Graph colouring enjoys many practical applications such as pattern matching, designing seating plans, scheduling exam timetable, solving sudoku puzzles as well as theoretical challenges. In our project, we have considered the application of graph colouring in the scheduling of the exam timetable.

Idea

The problem can be solved using graph coloring. Here we consider the nodes as the courses whose exams are to be scheduled. If there is a minimum of 1 student enrolled in two courses(nodes) then an edge is drawn between these two nodes. Then we implement graph colouring. The chromatic number represents the minimum number of slots required to conduct exams for all courses. The nodes having the same colour i.e., those courses exams can be conducted in the same slot without any conflicts with other courses.

Simple Genetic Algorithm

Simple Genetic Algorithm was developed by the Late Professor John Holland (1929-2015) of the University of Michigan as a classifier and after some years his graduate student Kenneth de Jong [De Jong 1975] firmly established that SGA is also capable of optimizing function. SGA begins with creating a random population and then create a loop in which first selection is done within the population followed by crossover and mutation. Here we have several options to terminate out of the loop such as we may stop when the maximum fitness in the population stabilizes or stops when the average fitness of the population is nearly the same as the maximum fitness or we may stop after a prescribed number of generations. It depends on the user.

- **Genes** – The characteristics of an individual are called genes.
- **Individuals** – It is also referred to as chromosomes. It is the collection of all gene values in a particular set.
- **Randomly Generated Initial Population** - The collection of all individuals involved in the process is known as population. The initial population is initiated randomly. The idea here is to introduce a wide variety of schemas in the population so that the better ones will prevail. Then we need to calculate fitness for each candidate of the population.

- **Fitness Function -** At each iteration of the algorithm, the individuals are evaluated using a fitness function. This is the objective function we seek to optimize. Individuals who achieve a better fitness score represent better solutions and are more likely to be chosen by the selection operator for the next generation. Over time, the quality of the solutions improves, the fitness values increase, and the process can stop once a solution is found with a satisfactory fitness value.
- **Selection** - After calculating fitness, the selection operator acts on them. The aim of the operator is to find out the candidates for the next generation based on their fitness value. More the fitness value higher will be the probability to get selected for the next generation. It should be mentioned while selecting, we also have to maintain population diversity. Population diversity is essential as it avoids any premature convergence of the population. There are many selection operators such as Roulette wheel selection operator, Stochastic Remainder Selection, Tournament Selection operator.
- **Crossover** - Next we perform a crossover operation between two random parents where the parts of their chromosomes are interchanged to create two new chromosomes representing the offspring. It will take place if the crossover probability is favourable. We will generate a random number ran in the interval [0,1]. Usually in SGA $p$cross is taken as a number closer to the upper bound of the random number, in the range of 0.8

to 0.9 for example and crossover is conducted only when the value of ran does not exceed $p$cross. There are many crossover operators such as Single Point Crossover, Two Point Crossover and Uniform Crossover.

- **Mutation** - The purpose of the mutation operator is to periodically and randomly refresh the population, introduce new patterns into the chromosomes, and encourage search in uncharted areas of the solution space. In SGA mutation is done with a low value of mutation probability $p\mu$, often well below 0.1. Mutation is performed if ran<$p\mu$. There are many mutation operators such as Bitwise Mutation, Flip Bit Mutation, etc.

The Algorithm

Libraries used

```
import numpy as np
import random
import matplotlib.pyplot as plt
import networkx as nx
```

Generating Population

Using the generateindividual() function we generate a chromosome containing a random permutation of colours

```
def generateindividual(N_nodes, N_Colors):
    C = np.concatenate((np.random.permutation(N_Colors),[random.choice(list(range(N_Colors))) for _ in range(N_nodes-N_Colors)]))
    return C
```

Fitness Function

The Fitness Function used here is defined below. The objective is to minimize the number of violations.

```
def cost_fcn(chromosome, G, N_nodes):
  violations=0
  for i in range(0,N_nodes):
    for j in range(i,N_nodes):
      if i != j:
        if G[i,j] == 1:
          if chromosome[i]==chromosome[j]:
            violations=violations+1
  C = violations
  return C
```

Crossover

Here we apply multipoint crossover, where we consider the 2nd half of a chromosome as a crossover site and we swap it.

We considered a high crossover probability = 0.9 for better results.

```
def crossover(parent1, parent2):
    child = parent1
    if random.uniform(0,1) < 0.9:
      cross_point = round(len(parent1) / 2)
      child[cross_point:len(parent1)] = parent2[cross_point:len(parent1)]
    return child
```

Mutation

For mutation, we consider a random point and swap it with a random colour.

The mutation probability used is 0.5 for better convergence.

```
def mutation(child, N_Colors):
    new_child = child
    if random.uniform(0,1) < 0.5:
        new_child[random.randint(0,len(child)-1)] = random.choice(list(range(N_Colors)))
    return new_child
```

Violations

This function is used to calculate the number of violations a given colour arrangement makes. Violation occurs when two connected nodes are of the same color.

```
def Violations(G,solution):
  if len(solution)==0:
    violations=100
  else:
    violations=0
    for i in range(0,N_nodes):
      for j in range(i,N_nodes):
        if i != j:
          if G[i,j] == 1:
            if solution[i]==solution[j]:
              violations=violations+1

  return violations
```

Plotting the Graph

```
colorList = list(set(bestsol))
colors = plt.cm.rainbow(np.linspace(0, 1, len(colorList)))

colorMap = []
for i in range(N_nodes):
    color = colors[colorList.index(bestsol[i])]
    colorMap.append(color)



def plotGraph(adjacency_matrix,noofnodes):
    gr = nx.from_numpy_matrix(adjacency_matrix)
    nx.draw(gr, node_size=500, node_color=colorMap, with_labels=True)
    plt.show()

plotGraph(G,N_nodes)
```

Main function

We consider a fixed population size of 100. The initial approach was to use a population of 10*number of nodes. But as the size of the dataset increased, the large population only increased the time complexity. So, a fixed population size of 100 was considered and returned good results.

The maximum number of generations is set to 300. (Based on various test cases)

To find the chromatic number, we loop through each number starting from 0 to the maximum possible chromatic number i.e, the number of nodes. (The number of generations taken is low due to the number of loops).

In each generation, the cost of each chromosome is calculated and the costs are sorted. If the minimum cost is 0, the process stops and returns the number of colours and colour arrangement. Otherwise, the top 50% of the sorted population is passed on to the next generation. Every chromosome part of the top 50% undergoes crossover and the generated child is subjected to mutation. These children are added to complete the rest of 50% of the population. These children may or may not introduce some diversity to the population. Diversity is important to avoid premature convergence as the population often gets trapped at the local optima. But in our case, there are no local or global optima as we are just aiming to make the objective function equal to 0. Every graph can have

multiple optimal solutions for a fixed number of colours so diversity is not very important.

Now, the algorithm runs until it gets a solution with 0 violations.

```
MaxColors = N_nodes
N_Colors = 0
Npop = 100
MaxGeneration = 300
chromosome = np.zeros((Npop,N_nodes))
bestsol=[]
while Violations(G,bestsol)!=0 and N_Colors<=MaxColors:
  fitness = []
  N_Colors = N_Colors + 1
  #Population generation
  for i in range(0,Npop):
      chromosome[i] = generateindividual(N_nodes, N_Colors)
# Genetic Optimization:
  costs = np.zeros(Npop)
  best = np.ones(MaxGeneration)
  for iter in range(0,MaxGeneration):
      for i in range(0,Npop):
          costs[i] = cost_fcn(chromosome[i], G, N_nodes)
      best[iter] = min(costs)
      fitness.append(best[iter])
      bestIDX = np.argmin(costs)
      bestsol = chromosome[bestIDX]
      if best[iter] == 0:
          break
      sortIDX = np.argsort(costs)
      bests = sortIDX[1:int(Npop / 2)]
      for i in np.arange(0,len(bests)-1,2):
          parent1 = chromosome[bests[i]]
          parent2 = chromosome[bests[i + 1]]
          child1 = crossover(parent1, parent2)
          child2 = crossover(parent2, parent1)
          child1 = mutation(child1, N_Colors)
          child2 = mutation(child2, N_Colors)
          chromosome[sortIDX[len(sortIDX) - i - 1]]= child1
          chromosome[sortIDX[len(sortIDX) - i - 2]]= child2
      if iter == MaxGeneration:
          print('Maximum Number of Generation Reached!')
          if best(iter) > 0:
              print('Input Graph and input number of colors has cost for GCP')
print(bestsol)
print(N Colors)
```

Input Format

The input used here to test our code was a text file with the number of nodes and all the edges of the graph. The general form of input is shown below-

n

e a b

where n represents the total number of nodes(courses)

e represents the edge number connecting two nodes a and b

If there is at least one student taking two courses then those two courses(nodes) are connected by an edge.

Here is a sample input(input.txt)

```
11
1 1 2
2 1 4
3 1 7
4 1 9
5 2 3
6 2 6
7 2 8
8 3 5
9 3 7
10 3 10
11 4 5
12 4 6
13 4 10
14 5 8
15 5 9
16 6 11
17 7 11
18 8 11
19 9 11
20 10 11
```

Then we create an adjacency matrix G of size n*n which contains 0 and 1 only.

G[i][j]=0 if i=j or there is no common student between i and j courses

G[i][j]=1 if there is at least one common student between i and j courses

After creating an adjacency matrix it is passed on to the main function for output.

```
h = open('input.txt', 'r')

content = h.readlines()
N_nodes = int(content[0])

G = np.zeros([N_nodes,N_nodes])

for line in range(1,len(content)):
  num = content[line].split(" ")
  i = int(num[1])-1
  j = int(num[2])-1
  G[i,j]=1
  G[j,i]=1
```

The code was tested using different data files given below and manual test cases.

input.txt

https://drive.google.com/file/d/1jNPmJzDDZJPh-bB9gUh719p1k_655max/view?usp=sharing

input2.txt

https://drive.google.com/file/d/1g_aAiDZnlxtCenHRE_inMYeIL09ygql2/view?usp=sharing

input3.txt

https://drive.google.com/file/d/1FyEGjpr6y3WoQQ60Aky7r4WWZKNVxefC/view?usp=sharing

input4.txt

https://drive.google.com/file/d/1P3v0UDIt6-PFe5knucCoZqwav3Kju5Hn/view?usp=sharing

input5.txt

https://drive.google.com/file/d/16bjCqjZ6HW90iiV0Zgp11at2jFLMTLDO/view?usp=sharing

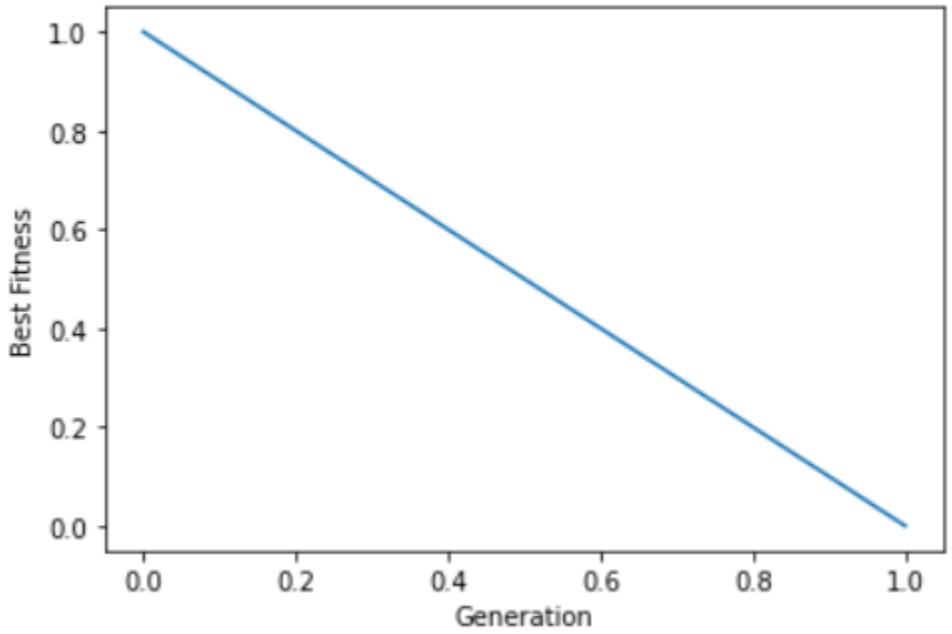
Results

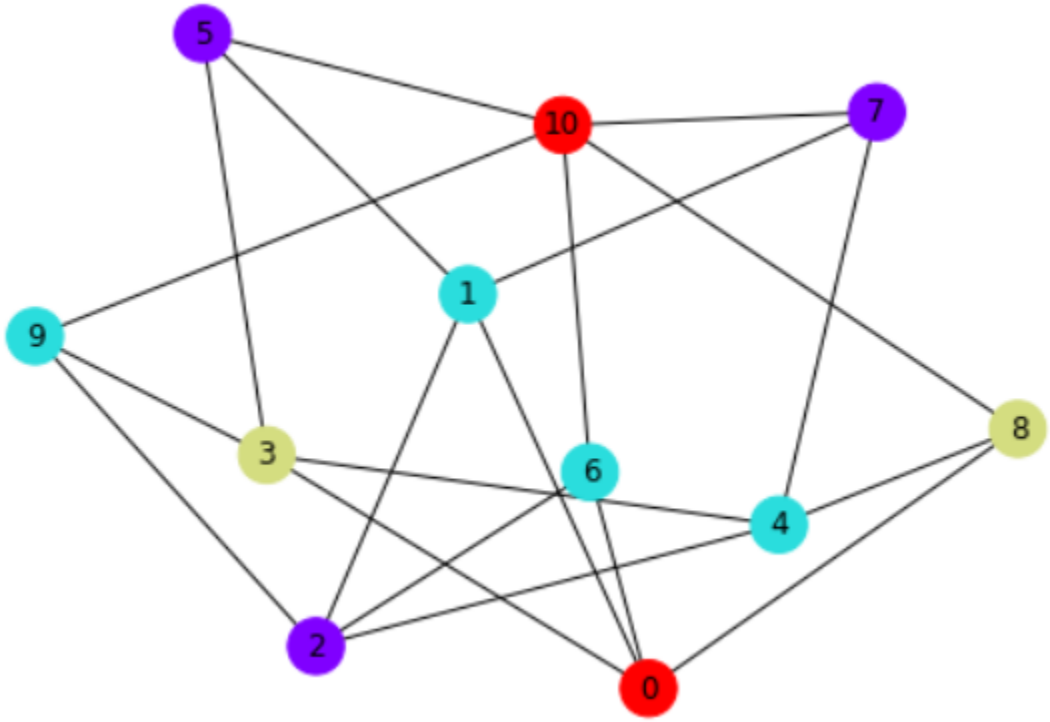
The output given by the code is

- Optimal Colour Arrangement
- Minimum Chromatic Number
- Number of Violations
- Best Fitness v/s Generations Graph
- Coloured Graph

input.txt file(No of nodes = 11 , Edges = 20) Time taken = 3s

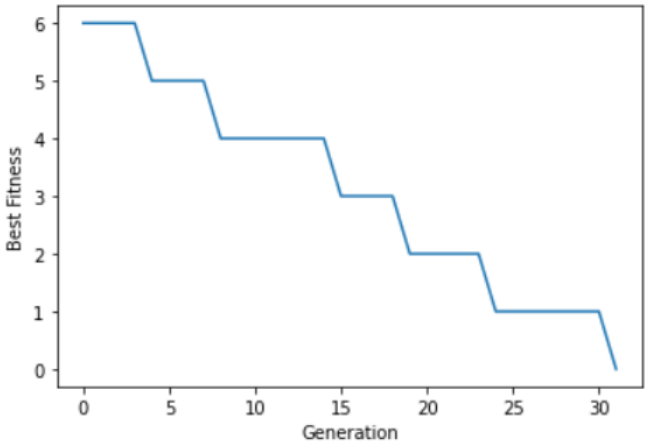
```
Colour Arrangement [3. 1. 0. 2. 1. 0. 1. 0. 2. 1. 3.]
Min Chromatic Number 4
Violations 0
```

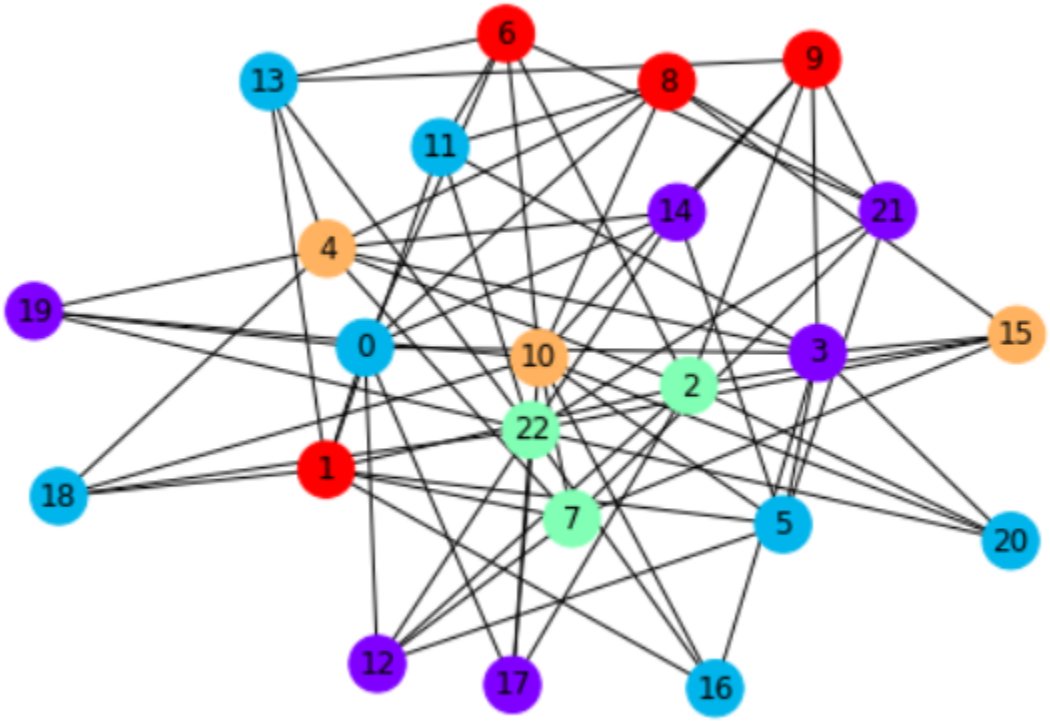




input2.txt file(No of nodes = 23, Edges = 71) Time taken = 20s

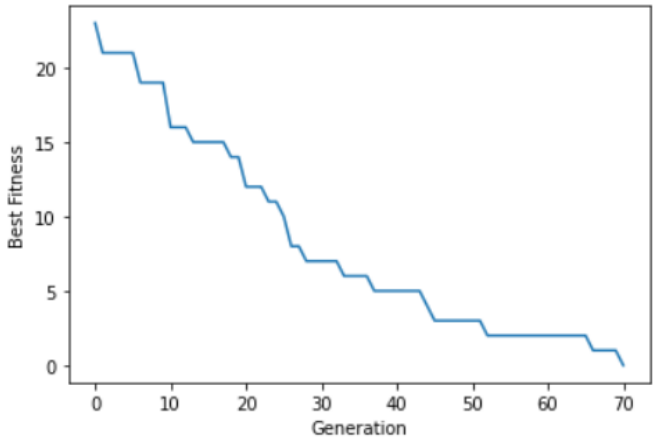
```
Colour Arrangement [1. 4. 2. 0. 3. 1. 4. 2. 4. 4. 3. 1. 0. 1. 0. 3. 1. 0. 1. 0. 1. 0. 2.]
Min Chromatic Number 5
Violations 0
```

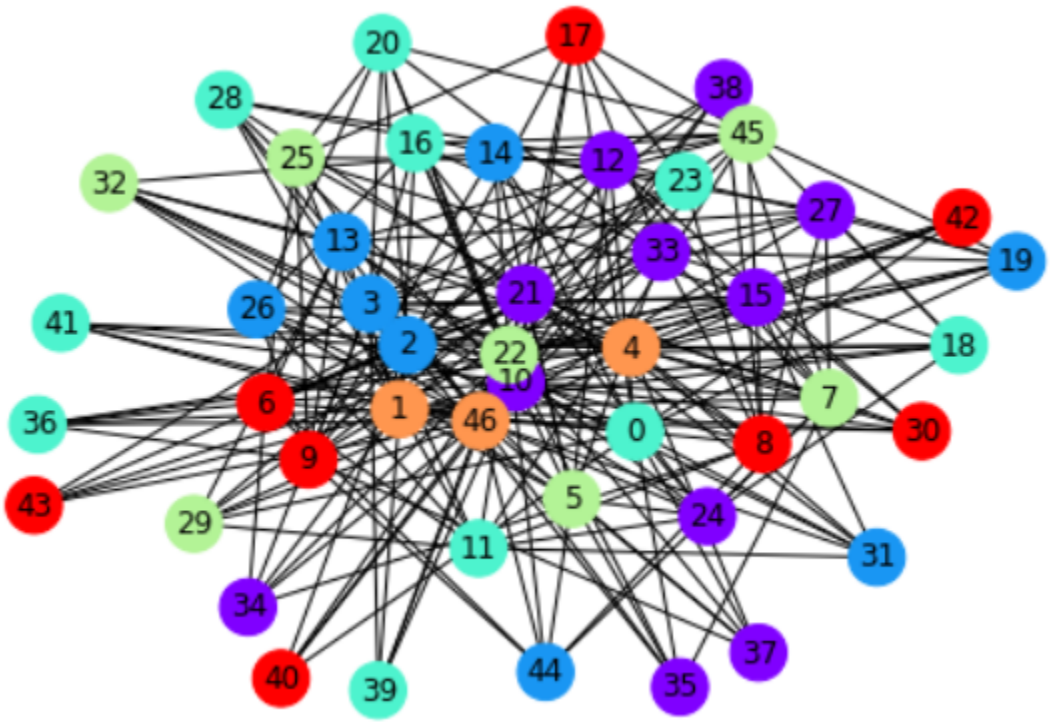




input3.txt file(No of nodes = 47, Edges = 236) Time taken = 102s

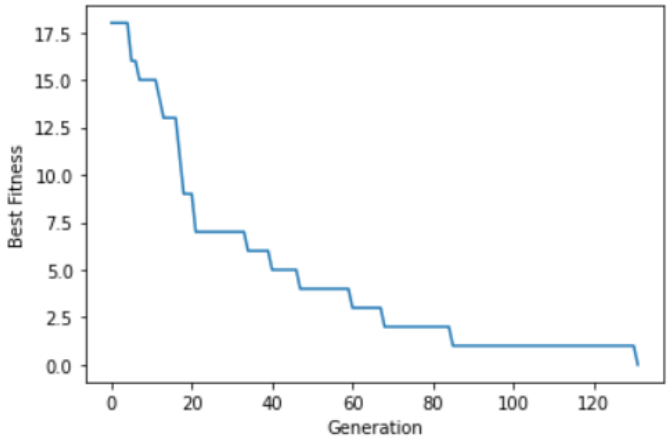
```
Colour Arrangement [2. 4. 1. 1. 4. 3. 5. 3. 5. 5. 0. 2. 0. 1. 1. 0. 2. 5. 2. 1. 2. 0. 3. 2.
 0. 3. 1. 0. 2. 3. 5. 1. 3. 0. 0. 0. 2. 0. 0. 2. 5. 2. 5. 5. 1. 3. 4.]
Min Chromatic Number 6
Violations 0
```

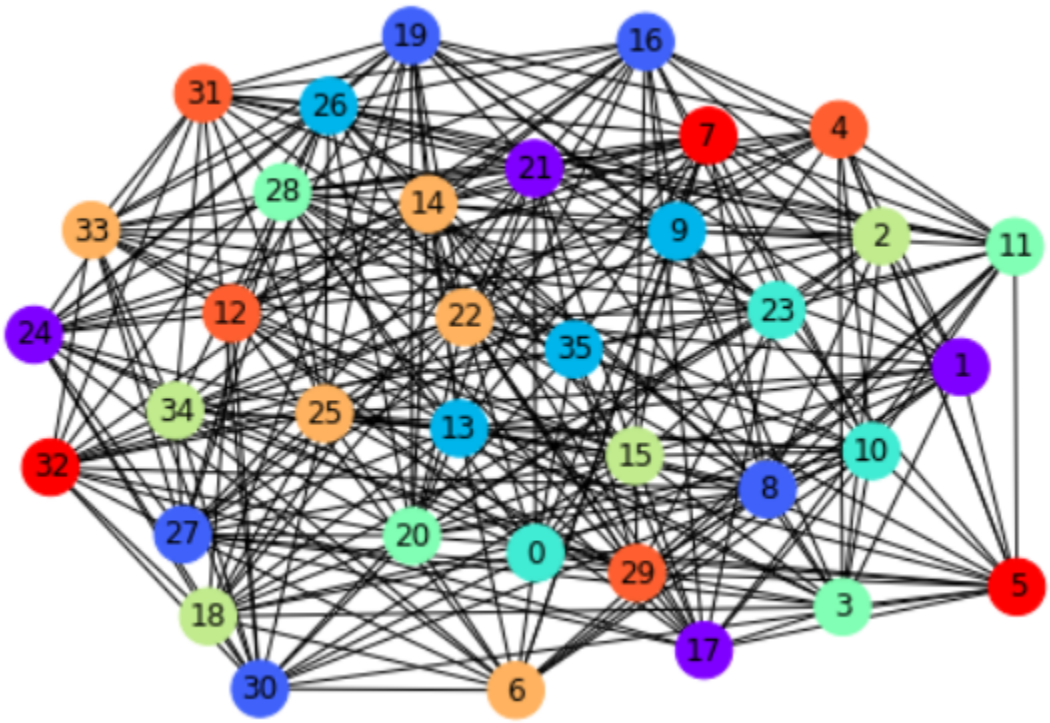




input4.txt file(No of nodes = 36, Edges = 290) Time taken = 112s

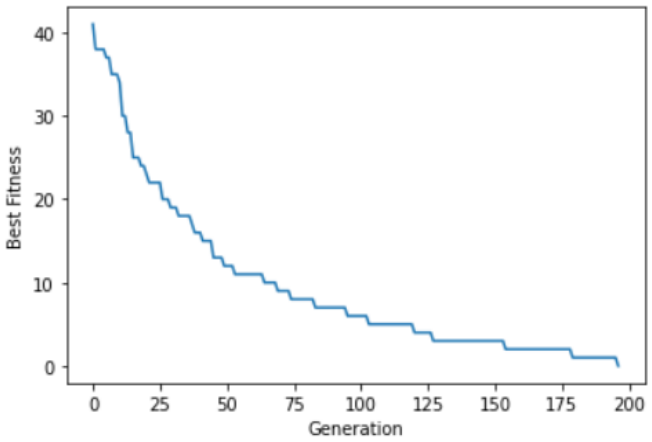
```
Colour Arrangement [3. 0. 5. 4. 7. 8. 6. 8. 1. 2. 3. 4. 7. 2. 6. 5. 1. 0. 5. 1. 4. 0. 6. 3.
 0. 6. 2. 1. 4. 7. 1. 7. 8. 6. 5. 2.]
Min Chromatic Number 9
Violations 0
```

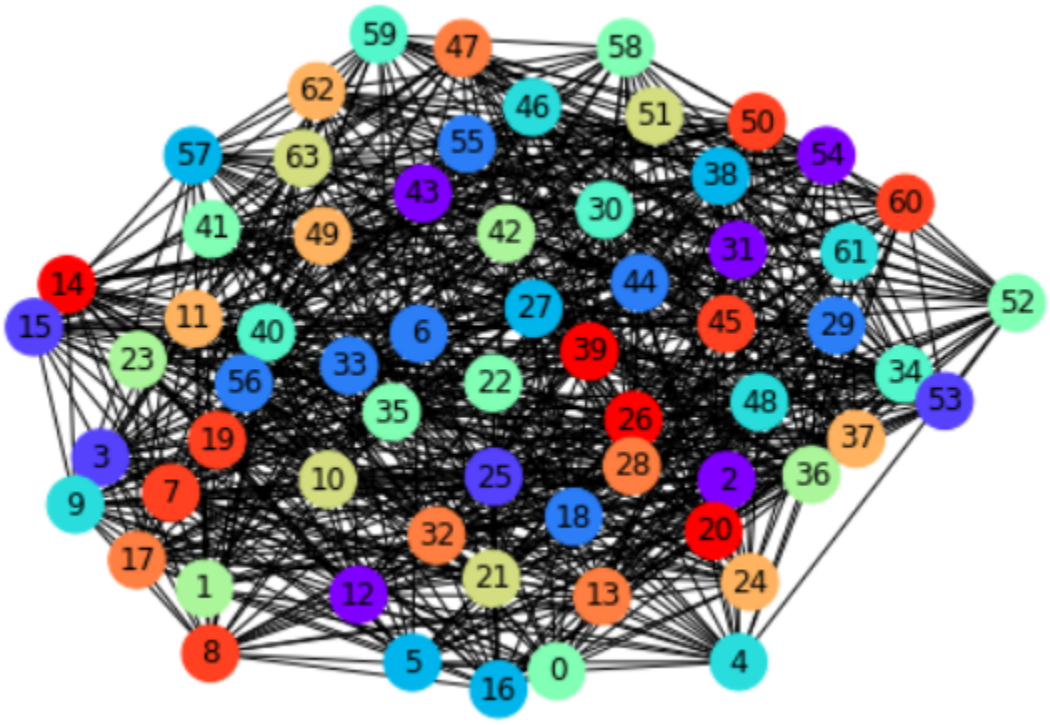




input5.txt file(No of nodes = 64, Edges = 728) Time taken = 8min 12s

```
Colour Arrangement [ 6.  7.  0.  1.  4.  3.  2. 11. 11.  4.  8.  9.  0. 10. 12.  1.  3. 10.
  2. 11. 12.  8.  6.  7.  9.  1. 12.  3. 10.  2.  5.  0. 10.  2.  5.  6.
  7.  9.  3. 12.  5.  6.  7.  0.  2. 11.  4. 10.  4.  9. 11.  8.  6.  1.
  0.  2.  2.  3.  6.  5. 11.  4.  9.  8.]
Min Chromatic Number 13
Violations 0
```





Conclusion

The nodes which are of the same colour represent those courses that can be conducted in the same slot without any conflicts. The chromatic number represents the minimum number of slots that will be needed to conduct the exams. This is the way in which we can schedule the exam timetable of colleges where there are many courses and students can choose any course of their own choice.

Colab NoteBook Link containing the code - https://colab.research.google.com/drive/1sCpRA-hzkw4uTGyb-YmvoNlXWMLcxbw2#scrollTo=A2xUTZhTvlnq

Further Enhancements

- This code is a simple and long way of solving a graph colouring problem using genetic algorithms. For nodes greater than 50, this algorithm will be ineffective because of time complexity.
- We will work on finding the chromatic number directly without looping through each number.

References

- Hands-On Genetic Algorithms with Python by Eyal Wirsansky
- Notes provided in class
- Wikipedia

- GeeksForGeeks - Graph Colouring