Graph Colouring Problem

November 10, 2019

GAA Project 2019 . Course teacher: ***Dr. Vibhor Kant***

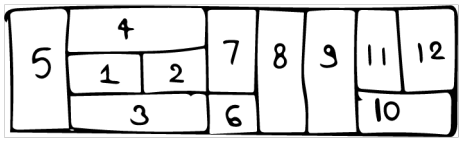
Team Member	Roll Number
Ansh Mittal	17UCS028
Anshul Jain	17UCS029
Anshul Kiyawat	17UCS030
Anshu Musaddi	17UCS185

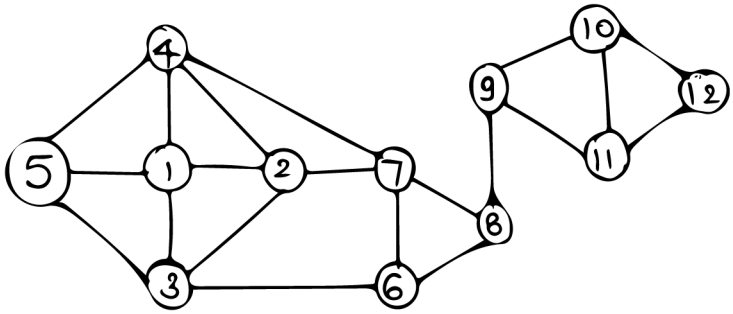
1 Importing the required libraries

[1]:
```
import pandas as pd
import numpy as np
import random
```

2 Defining the Problem

The problem consistis of 12 nodes which are connected as shown below. There are thre colours namely C1, C2, C3





[2]:
```
graph = [[0,1,1,1,1,0,0,0,0,0,0,0],
         [1,0,1,1,0,0,1,0,0,0,0,0],
         [1,1,0,0,1,1,0,0,0,0,0,0],
         [1,1,0,0,1,0,1,0,0,0,0,0],
         [1,0,1,1,0,0,0,0,0,0,0,0],
         [0,0,1,0,0,0,1,1,0,0,0,0],
         [0,1,0,1,0,1,0,1,0,0,0,0],
         [0,0,0,0,0,1,1,0,1,0,0,0],
         [0,0,0,0,0,0,0,1,0,1,1,0],
         [0,0,0,0,0,0,0,0,1,0,1,1],
         [0,0,0,0,0,0,0,0,1,1,0,1],
         [0,0,0,0,0,0,0,0,0,1,1,0]]
colour = ["C1","C2","C3"]
```

3 Defining the functions required for Genetic Algorithm

3.1 Fitness Function

We have defined the fitness function to calculate the number of collision. A collision is occuring if two adjacent nodes are of the same colour

[3]:
```
def fitness(chrm,graph):
    coll = 0
    for i in range(len(chrm)):
        for j in range(len(chrm)):
            if graph[i][j] == 1:
                if chrm[i]==chrm[j]:
                    coll = coll+1
    return coll
```

3.2 Population Initialization

The following function initializes a unique random chromosomes. The population size will be of size n passes as a parameter. The following function will return a Pandas dataframe consisting of Chromosomes and their respective fitness values

[4]:
```
def randinit(n,graph,colour):
    df = pd.DataFrame(np.zeros((n,2)),columns=["Chromosome","Fitness"],␣
 ↪dtype=object)
    i = 0
    chrmlen = len(graph)
    cl = len(colour)
    while i < n:
        chrm = []
        for j in range(chrmlen):
            chrm.append(colour[random.randint(0,cl-1)])
        if chrm not in list(df["Chromosome"]):
```

```
            df.at[i,"Chromosome"] = chrm.copy()
            df.at[i,"Fitness"] = fitness(chrm,graph)
            i = i+1
    df["Fitness"] = df["Fitness"].astype('int64')
    return df
```

3.3 Crossover

We are using **1 point crossover** where the crossover point is chosen randomly.

[5]:
```
def crossover(chrm1,chrm2):
    n = len(chrm1)
    point = random.randint(0,n)
    chrm3 = chrm1[:point]
    chrm3.extend(chrm2[point:])
    chrm4 = chrm2[:point]
    chrm4.extend(chrm1[point:])
    return chrm3,chrm4
```

3.4 Mutation

We are performing **random swapping** between any two allele in the chromosome. It is ensured that both the points are unique.

[6]:
```
def mutation(chrm):
    n = len(chrm)
    point1 = random.randint(0,n-1)
    point2 = random.randint(0,n-1)
    while(point1==point2):
        point1 = random.randint(0,n-1)
    chrm[point2],chrm[point1] = chrm[point1],chrm[point2]
    return chrm.copy()
```

3.5 Parent Selection

We are selecting a random parent from the top 100 chromosomes of the population and 1 from the rest 400 chromosomes.

[7]:
```
def parentsel(chrmlist):
    p1 = chrmlist[random.randint(0,99)]
    p2 = chrmlist[random.randint(100,499)]
    return p1,p2
```

3.6 Survival Selection

The bottom 2 worst chromosomes are replaced with the new better chromosomes.

[8]:
```
def replaceWorst(pop,graph,v=False):
    pop = pop.sort_values(by=['Fitness'])
    p1, p2 = parentsel(list(pop["Chromosome"]))
    c1, c2 = crossover(p1, p2)
    c1 = mutation(c1)
    c2 = mutation(c2)
    if v:
        print('Child 1:{0}\tFitness: {1}\nChild 2:{2}\tFitness: {3}'.format(c1,␣
↪fitness(c1,graph), c2, fitness(c2,graph)))
    pop = pop.sort_values(by=['Fitness'])
    pop = pop.reset_index(drop=True)
    pop.iloc[pop.shape[0]-2] = [c1, fitness(c1,graph)]
    pop.iloc[pop.shape[0]-1] = [c2, fitness(c2,graph)]
    return pop
```

3.7 Utility Functions

The following function ensures that weather the population has the Fittest chromosome.

[9]:
```
def hasSolution(p):
    if (pop[pop['Fitness']==0]).shape[0] == 0:
        return False
    return True
```

The following function finds and returns the fittest possible chromosome from the given population.

[10]:
```
def bestFitness(pop):
    p = pop.sort_values(by=['Fitness'])
    return p.iloc[0, 1]
```

The following function gives the fitness of the fittest chromosome from the population in return.

[11]:
```
def bestChromosome(pop):
    p = pop.sort_values(by=['Fitness'])
    return p.iloc[0, 0]
```

The following function gives the fittest chromosome from the population in return.

[12]:
```
def getSolution(pop):
    return (pop[pop['Fitness']==0]).iloc[0][0]
```

4 Execution of the Genetic Algorithm

[13]:
```
%%time

show_progress = False
lastgeneration = 0
```

```
generationInfo = pd.DataFrame(np.zeros((5000, 2)), columns=['Chromosome',␣
↪'Fitness'], dtype=object)

for generation in range(5000):
    lastgeneration = generation
    if generation == 0:
        pop = randinit(500, graph, colour)

    generationInfo.loc[generation] = [bestChromosome(pop), bestFitness(pop)]

    if hasSolution(pop):
        print("Found Solution. {:,} Generations generated.".format(generation))
        break

    if(show_progress):
        print("\nGeneration {}".format(generation))

    pop = replaceWorst(pop, graph, show_progress)

generationInfo = generationInfo[generationInfo['Chromosome']!=0]

if(not hasSolution(pop)):
    print("Solution not found program ended.")
```

```
Found Solution. 411 Generations generated.
CPU times: user 2.05 s, sys: 3.09 ms, total: 2.05 s
Wall time: 2.06 s
```

4.1 Generated Solution

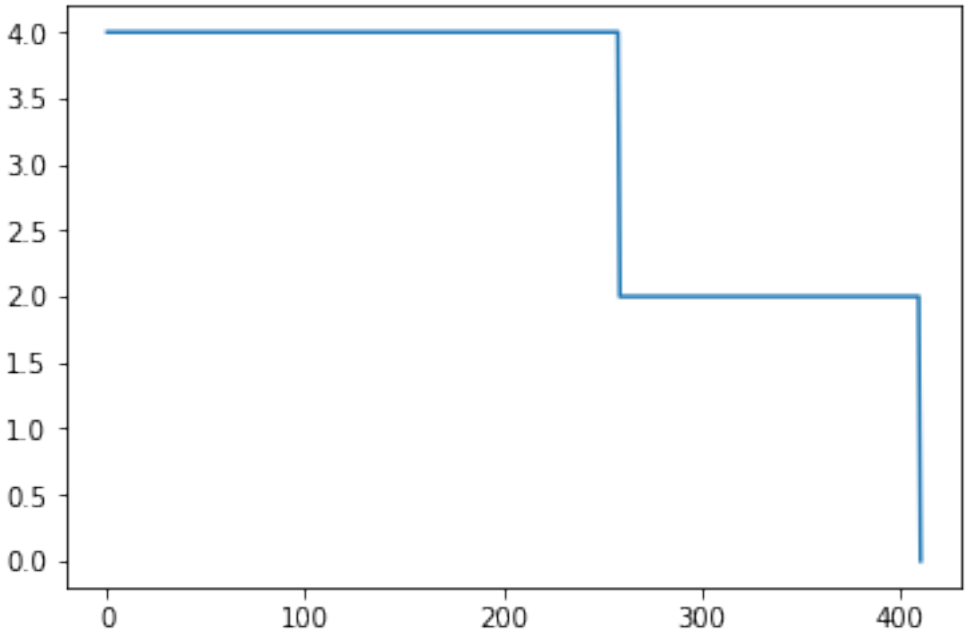
[14]:
```
print(getSolution(pop))
```

```
['C3', 'C2', 'C1', 'C1', 'C2', 'C2', 'C3', 'C1', 'C3', 'C2', 'C1', 'C3']
```

4.2 Performance of the Genetic Algorithm For Each Generation

Plot of the best fitness in a generation versus generation number.

[16]:
```
import matplotlib.pyplot as plt
plt.plot(range(generationInfo.shape[0]), list(generationInfo['Fitness']))
plt.show()
```


4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
100
200
300
400