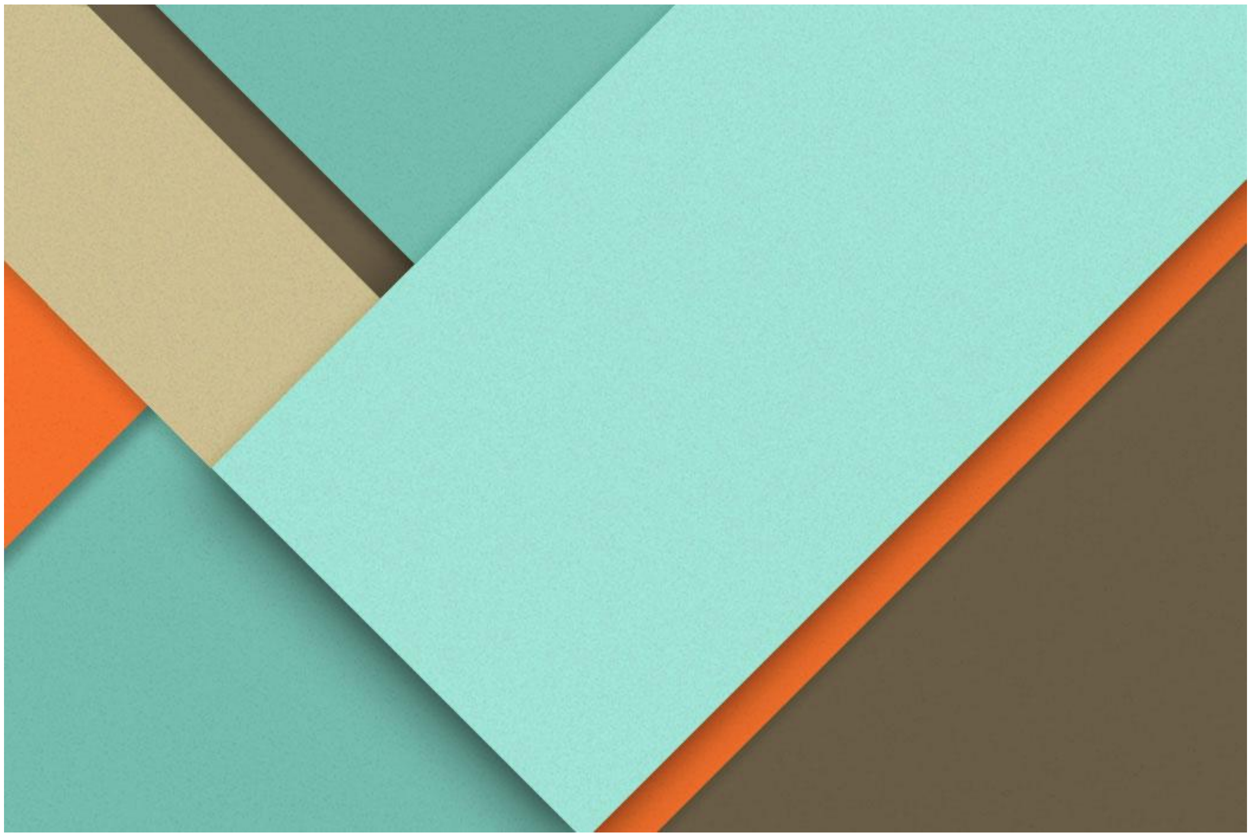


Genetic Algorithms in Engineering Process Modelling

Term Project : Genetic Algorithms for Vertex Cover Problem

Rishabh Agarwal
19MA20042

Rishit Singhania
19MA20043

1. Abstract

In this paper we present a technique that uses Genetic Algorithms to solve the **Vertex Cover Problem**. We compare the results with the results obtained from solving the problem with popular heuristics and also discuss the applications of the proposed algorithm.

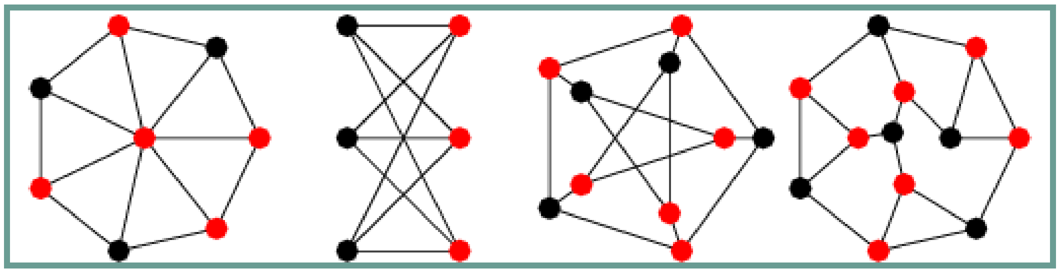
2. Introduction

The vertex cover problem is a well-known classical **NP-complete** optimization graph problem. A vertex cover of a graph is a set of vertices such that it includes at least one endpoint of every edge of the graph. The vertex cover problem is to find a cover of minimum size.

The vertex cover problem is one of the most studied problems and is a very active field of research, primarily because of its application in:

- Network and Security
- Biology
- Meteorology
- Finance

Although a solution to an NP-Complete problem can be verified “quickly”, there is no known way to find a solution quickly. Hence NP-Complete problems are often addressed by using approximation algorithms or heuristic methods. It is tempting therefore, to use search heuristics like Genetic Algorithms. In this paper we thus compare the performance of Genetic Algorithms to approximation algorithms.



(Fig. 1) Vertex Cover for different graphs

3. Proposed Genetic Algorithm

Genetic algorithm(GA) is a heuristic method based on Darwin's theory of evolution and genetic laws. In the first iteration of the algorithm, the population usually consists of randomly generated individuals. Each individual represents an encoded solution of a problem and has a value named fitness associated with it, which represents the quality of the individual in the current population. After applying the genetic operators of selection, crossover and mutation to the current population, the next generation is formed. This process is iteratively performed until some finishing criterion is satisfied.

I. Mathematical Formulation

In proposed implementation of GA, the binary encoding of the individuals is used. Each solution is represented by a binary string of length $|V|$. Digit 1 in the genetic code denotes that particular vertex is in corresponding vertex cover S, while 0 shows it is not.To define a fitness function, we assign a penalty of 1 for every vertex that is included in the vertex cover, i.e, every bit that is assigned the value 1. For every edge we check if either of the end-points are in the vertex cover. If this isn't the case we add a large penalty to the fitness value of the individual.

penalty(i, j) = 10, if (i, j) is an edge but neither i nor j is included in the vertex cover,

penalty(i, j) = 0, otherwise.

Fitness = Σ penalty(i, j) for all edges + number of vertices included in vertex cover.

The minimum vertex cover has the lowest fitness value so the individual having lower fitness value will be the fitter individual.

II. Graph Generation

The input graph is generated in the form of an adjacency matrix.

graph[i][j] = 1, (i,j) is an edge

= 0, otherwise.

The density of the graph is controlled by a threshold value **p**. For each pair (i,j) a value is picked randomly from the range (0,1) and if it is less than p an edge is added between i and j.

So, for dense graphs we use greater values of p (>0.3).

For plotting the graphs we use **Networkx** which is a **Python** language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

III. Creating the population

We need a big enough population to maintain diversity, however the population cannot be too big as computation time will increase. The population size used is 50. For preserving diversity, the individuals in the initial population are randomly generated. The population consists of n length vectors where each index shows whether this vertex is included in the vertex cover represented by that individual.

IV. Selection

Selection is the stage of a **genetic algorithm** in which individual genomes are chosen from a population for later breeding (using the crossover operator). The fitness function is evaluated for each individual, providing fitness values, which are then normalized.

We use the **tournament selection** operator to find the minimum vertex cover of the given graph.

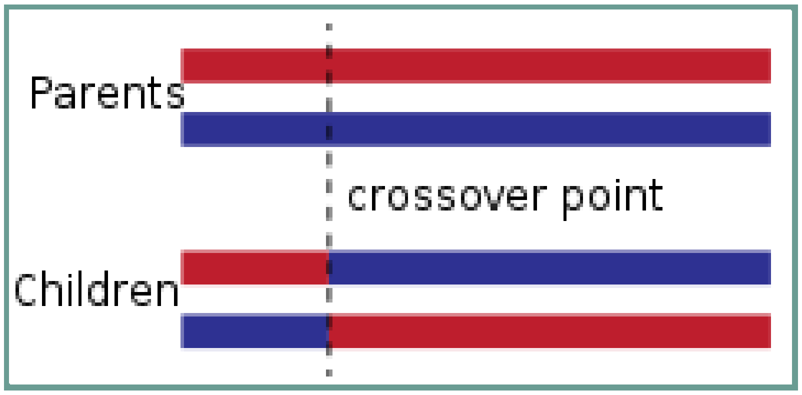
The population is first shuffled. We group the individuals in pairs and the fitter one goes to the next generation . If our population size is N, N/2 individuals are selected. The same process is repeated, so we get N/2 more individuals maintaining the population size. This technique ensures that there are two copies of the fittest individual and no copies of the least fit individual.

V. Crossover

In genetic algorithms and evolutionary computation, **crossover**, also called recombination, is a genetic operator used to combine the genetic information of two parents to generate new offspring. It is one way to stochastically generate new solutions from an existing population. Newly generated solutions are typically mutated before being added to the population.

The type of crossover used is **Single-point Crossover**.

A point on both parents' chromosomes is picked randomly, and designated a '**crossover point**'. Bits to the right of that point are swapped between the two parent chromosomes. This results in two offspring, each carrying some genetic information from both parents.



(Fig. 2) Schematic Diagram of Single-point Crossover

VI. Mutation

Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. It is analogous to biological mutation. Mutation alters one or more gene values in a chromosome from its initial state. In mutation, the solution may change entirely from the previous solution. Hence GA can come to a better solution by using mutation. Mutation occurs during evolution according to a user-definable mutation probability. This probability should be set low. If it is set too high, the search will turn into a primitive random search.

The mutation operator in our code randomly selects vertices and includes them in the vertex cover if they were not present or can remove them from the vertex cover if they were already present. This speeds up the convergence of the GA and even prevents the code from being stuck in a local optimum.

During the initial 400 generations, we want our algorithm to explore different solutions and thus the mutation probability is set to be 0.4.

However during the later generations, we have reached close to the optimal solution and thus we just want our code to converge to the most optimal solution rather than exploring. So, the mutation probability is lowered to 0.2.

4. Heuristic Used

To evaluate the performance of our code we use an approximate algorithm which finds a vertex cover of the graph and compare it with the best fitness obtained from the genetic algorithm.

The heuristic used is known as **LR method**.

LR: if $u \notin C$, $\{v \mid uv \in E \land v \notin C\}$ is put in C;

For each vertex not included in the vertex cover, we iterate over its neighbours and if the neighbour is also not in the vertex cover, it is added to the vertex cover.

Since this is an approximate algorithm, the vertex cover obtained might not be the most optimal one but it is close to the best solution. So, we can compare the results obtained from the heuristic method and the genetic algorithm to check the performance of our evolutionary algorithm.

5. Experimental Results

We want to analyse the working of our algorithm for dense as well as sparse graphs. The density of the graph is controlled by the threshold value p explained earlier.

I. Sparse Graph

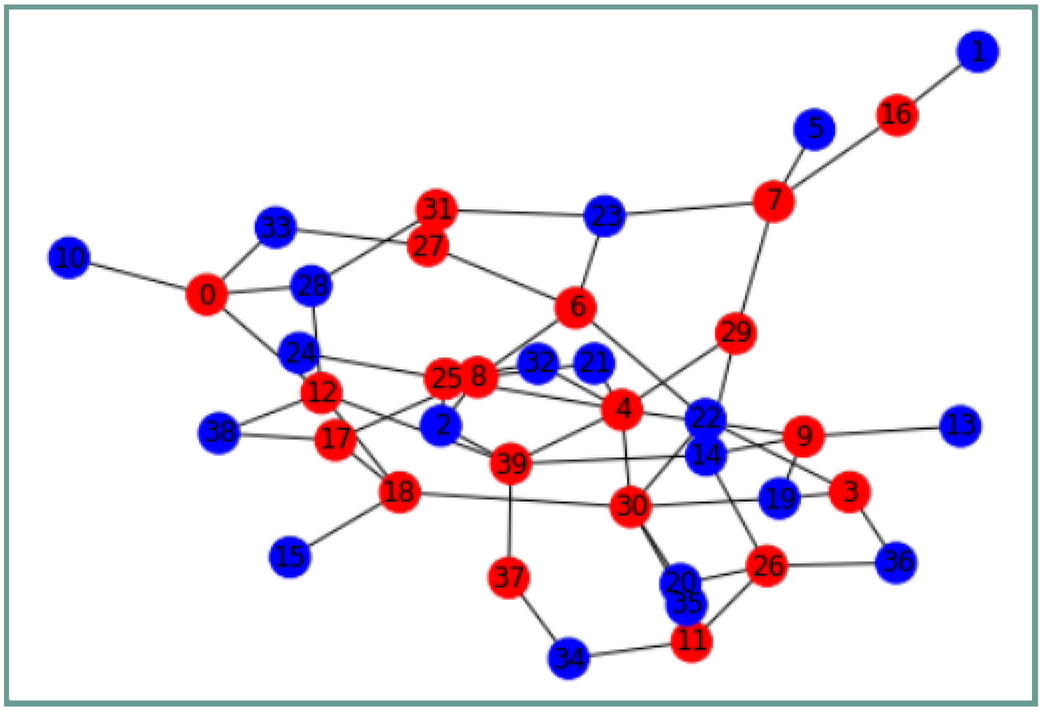
We generate a sparse graph consisting of 40 vertices by setting the value of p to be 8%.

The vertices which are red are included in the vertex cover.

The vertices which are blue are not part of the vertex cover.

There cannot be an edge connecting two blue vertices in the graph.

The graph looks like:



(Fig. 3.1) Vertex Cover for Sparse Graph

Details of the fittest individual per century:

Generation: 0 Best Fitness: 54 Individual: [0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1]

Generation: 100 Best Fitness: 20 Individual: [1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1]

Generation: 200 Best Fitness: 20 Individual: [1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1]

Generation: 300 Best Fitness: 20 Individual: [1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1]

Generation: 400 Best Fitness: 20 Individual: [1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1]

Generation: 500 Best Fitness: 20 Individual: [1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1]

Generation: 600 Best Fitness: 20 Individual: [1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1]

Generation: 700 Best Fitness: 20 Individual: [1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1]

Generation: 800 Best Fitness: 20 Individual: [1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1]

Generation: 900 Best Fitness: 20 Individual: [1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1]

The vertex cover of the given graph consists of 20 vertices.

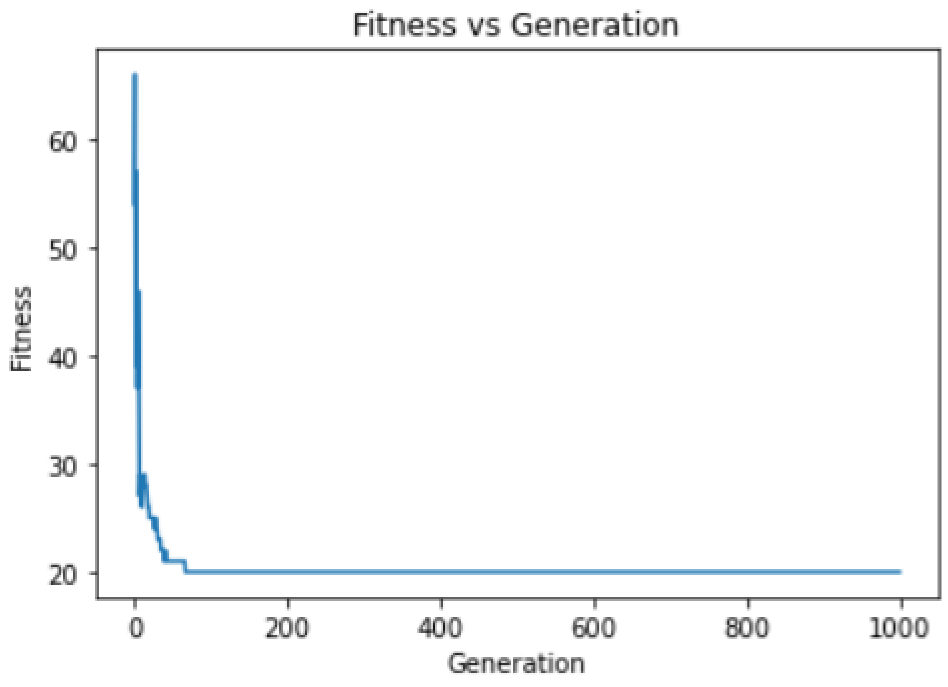
Vertex cover obtained by the heuristic:

Fitness: 25

Individual: [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1]

So, we can see that the genetic algorithm produces more optimal solution than the approximate algorithm.

To show the convergence of the Genetic algorithm, we plot the fitness of the fittest individual against the generation number.



(Fig. 3.2) Plot of Fitness vs Generation for Sparse Graph

II. Dense Graphs

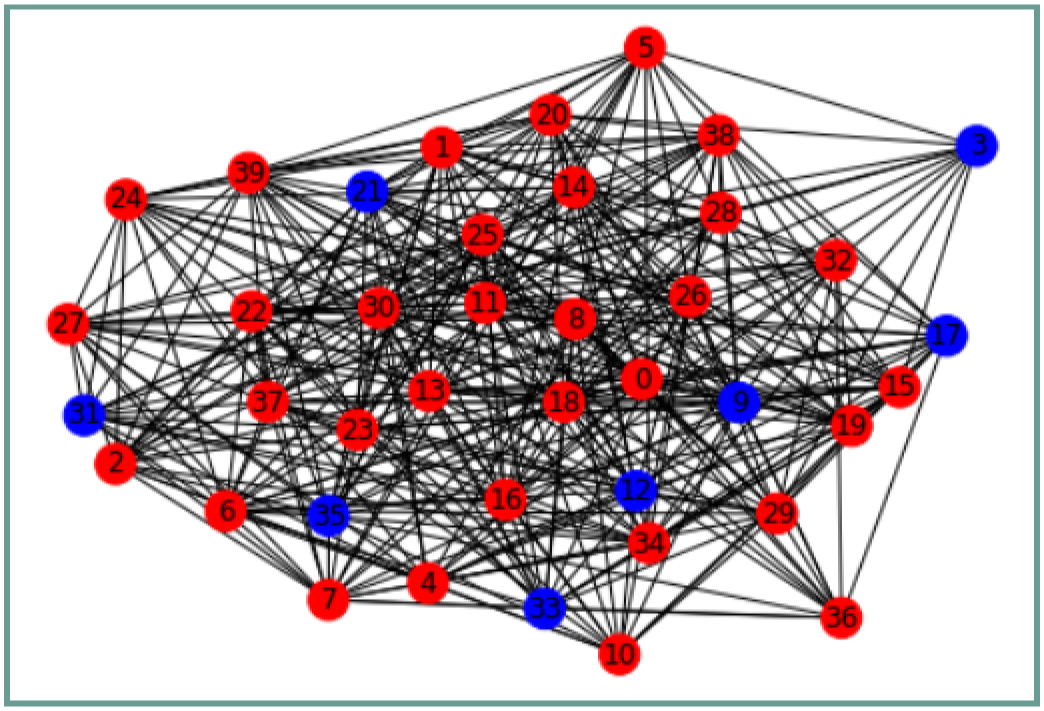
In dense graphs, the number of edges is closer to the maximum possible number of edges. The threshold value is set to be 0.5.

The vertices which are red are included in the vertex cover.

The vertices which are blue are not part of the vertex cover.

There cannot be an edge connecting two blue vertices in the graph.

The graph looks like:



(Fig. 4.1) Vertex Cover for Dense Graph

Details of the fittest individual per century:

Generation: 0 Best Fitness: 72 Individual: [1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1]

Generation: 100 Best Fitness: 35 Individual: [1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1]

Generation: 200 Best Fitness: 35 Individual: [1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1]

Generation: 300 Best Fitness: 34 Individual: [1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1]

Generation: 400 Best Fitness: 34 Individual: [1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]

Generation: 500 Best Fitness: 34 Individual: [1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]

Generation: 600 Best Fitness: 34 Individual: [1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]

Generation: 700 Best Fitness: 33 Individual: [1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1]

Generation: 800 Best Fitness: 33 Individual: [1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1]

Generation: 900 Best Fitness: 33 Individual: [1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1]

The vertex cover of the given graph contains 33 vertices.

Vertex cover obtained by the heuristic:

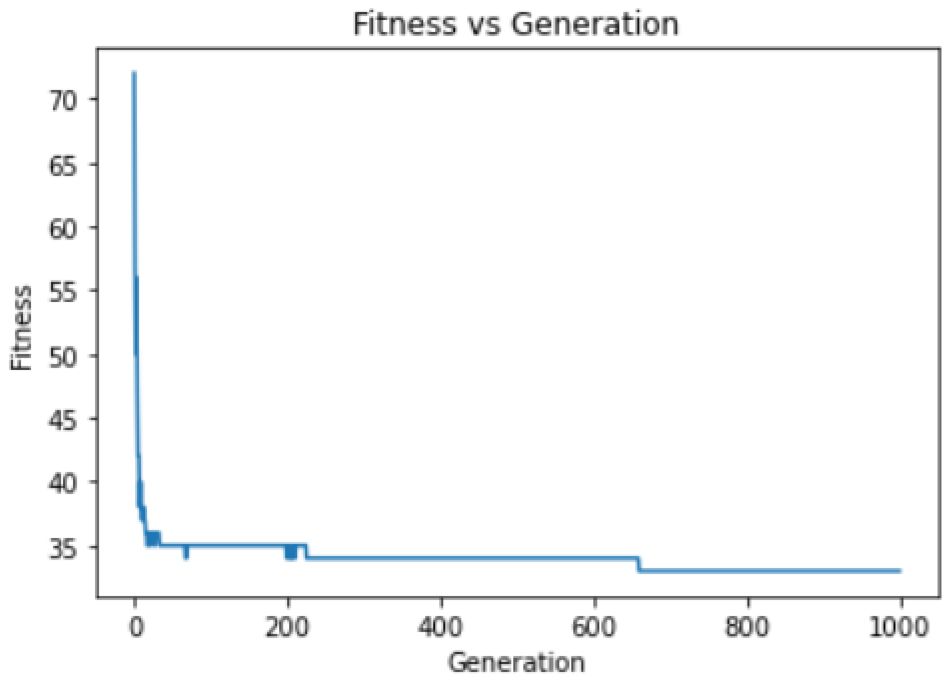
Fitness: 36

Individual: [0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Here also the genetic algorithm performs better compared to the heuristic used.

We note that for dense graphs the size of the vertex cover increases. More vertices need to be included such that each edge is covered by at least 1 vertex.

To show the convergence of the Genetic algorithm, we plot the fitness of the fittest individual against the generation number.



(Fig. 4.2) Plot of Fitness vs Generation for Dense Graph

6. Applications

I. Network Design

Say you have an art gallery with many hallways and turns. Your gallery is displaying very valuable paintings, and you want to keep them secure. You are planning to install security cameras in each hallway so that the cameras have every painting in view. If there is a security camera in a hallway, it can see every painting in the hallway. If there is a camera in the corner where two hallways meet (the turn), it can view paintings in both hallways. We can model this system as a graph where the nodes represent the places where the hallways meet or when a hallway becomes a dead end, and the edges are the hallways. Thus the problem reduces to the **Vertex Cover** problem.

II. Text Summarization

Text summarization is the process of reducing the content of a document with an automated system that retains most important points of the original document. In other words it is the process of automatically creating a compressed version of a given text that provides useful information for the user. We are interested to select a few sentences which will cover the semantics of the given text. We see the problem of summarization as a problem of selecting important sentences from a set of sentences. As each sentence is semantically connected, we view the collection of sentences as a connected weighted graph, considering the sentence as a node. We use **minimum vertex cover(MVC)** to select the important sentence from the connected weighted graph.

7. Problems

- The algorithm might be time-consuming for large graphs (number of vertices > 100). Also the time to arrive at a solution heavily depends on the initial random population.
- The stopping criterion for the algorithm is the number of generations. This is not a very good strategy as the algorithm might not arrive at the optimal vertex cover for the given graph. This may lead to sub-standard solutions in a few cases.
- As the graph gets denser (number of vertices increases), the probability of the algorithm getting stuck in local optima increases. But it is still able to outperform the heuristics.
- The algorithm has a tendency of getting stuck in a local optima until the mutation operator perturbes it in the right direction to approach the solution. Using advanced mutation operators might help overcome this and help it converge faster.

8. Conclusion

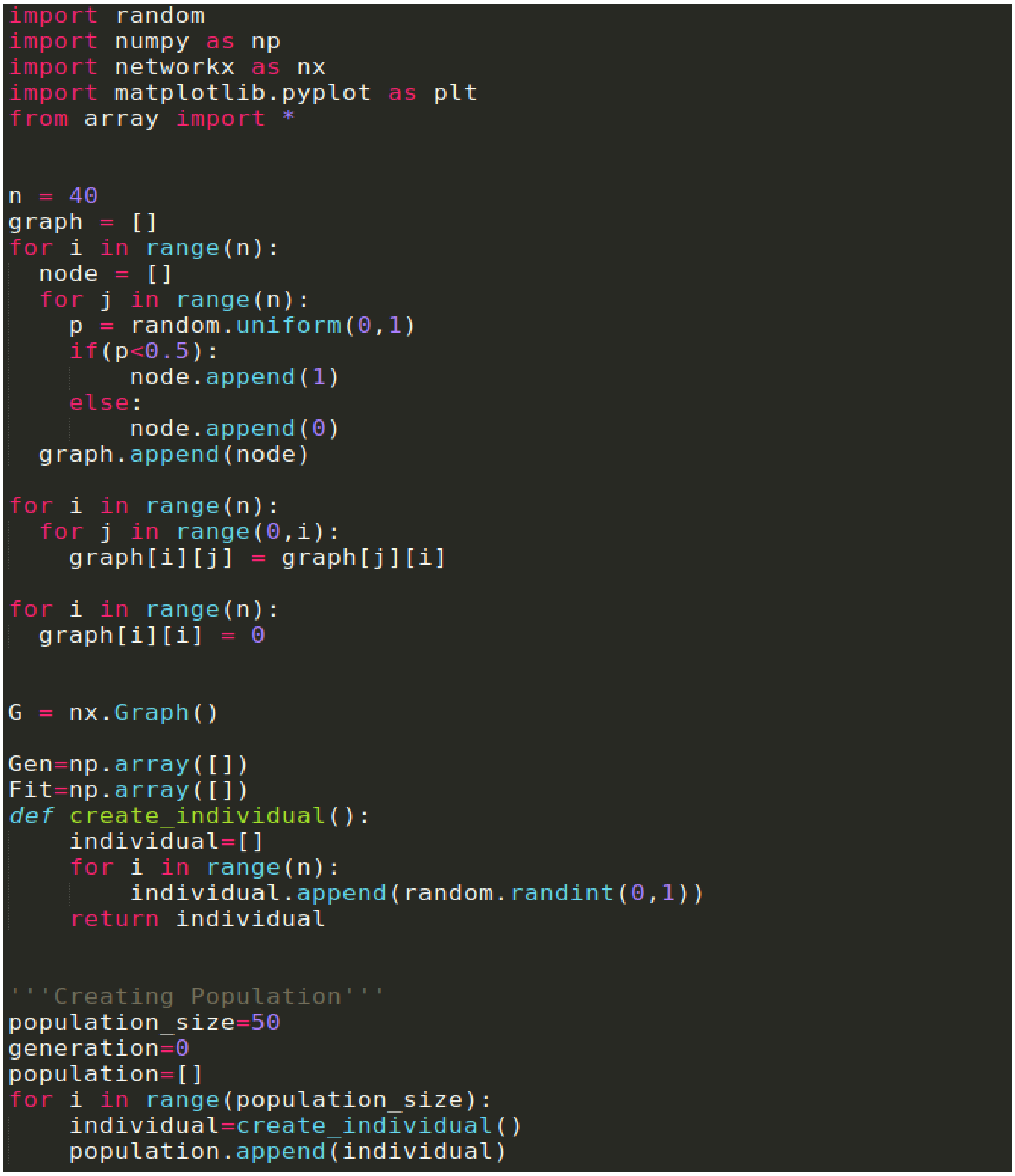
Thus we see that **genetic algorithms** are a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation. Genetic Algorithms have the ability to deliver a "good-enough" solution "fast-enough". This makes genetic algorithms attractive for use in solving optimization problems. The sublime simplicity of the algorithm and the impressive results it produces show just how powerful these algorithms are.

9. References

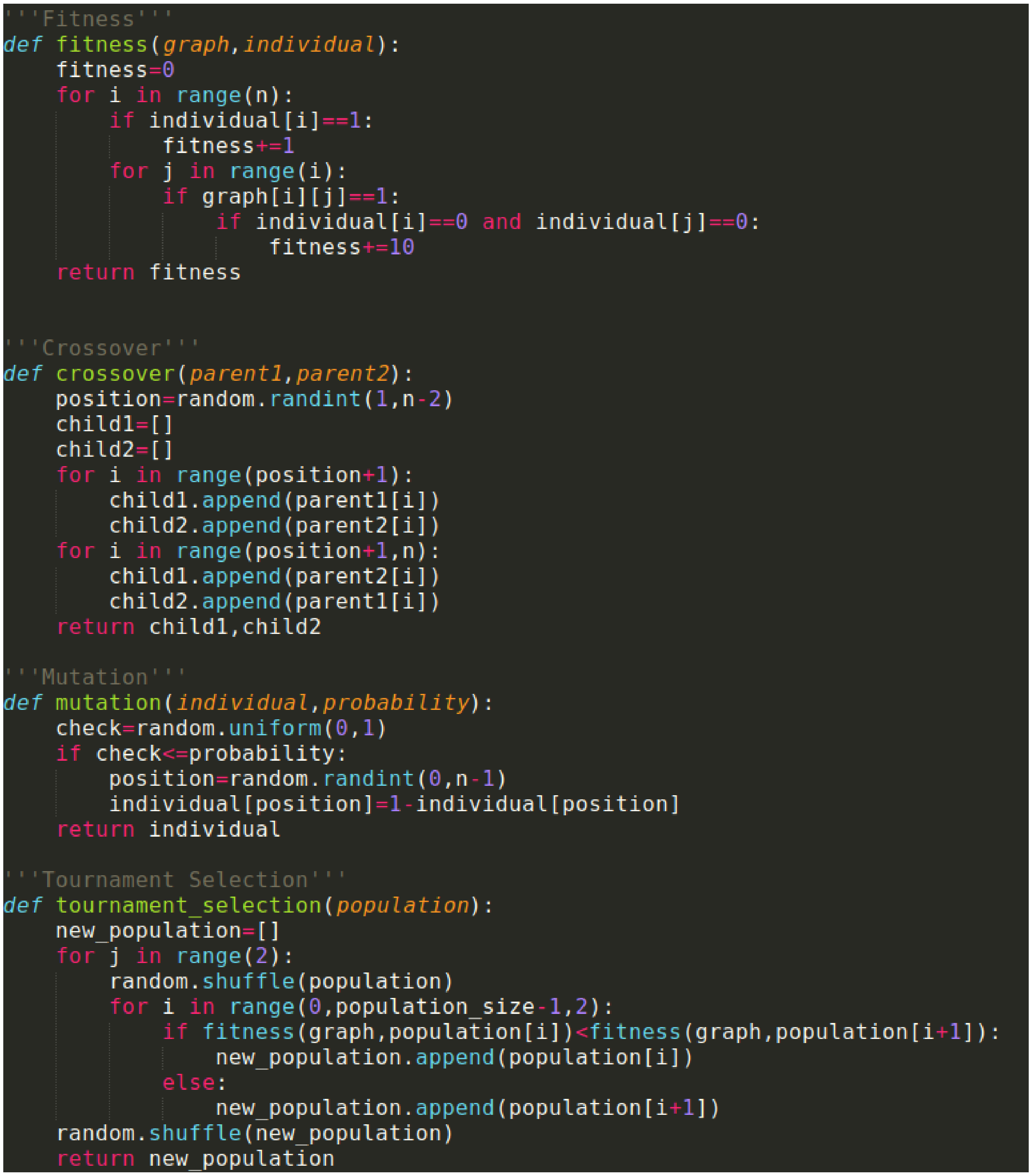
1. Atowar-Ul Islam and Bichitra Kalita, “Application of Minimum Vertex Cover for Keyword based Text Summarization Process”, *International Journal of Computational Intelligence Research (2017)*

2. Eric Angel, Romain Campigotto and Christian Laforest, “Implementation and Comparison of Heuristics for the Vertex Cover Problem on Huge Graphs”, *11th International Symposium on Experimental Algorithm (2012)*

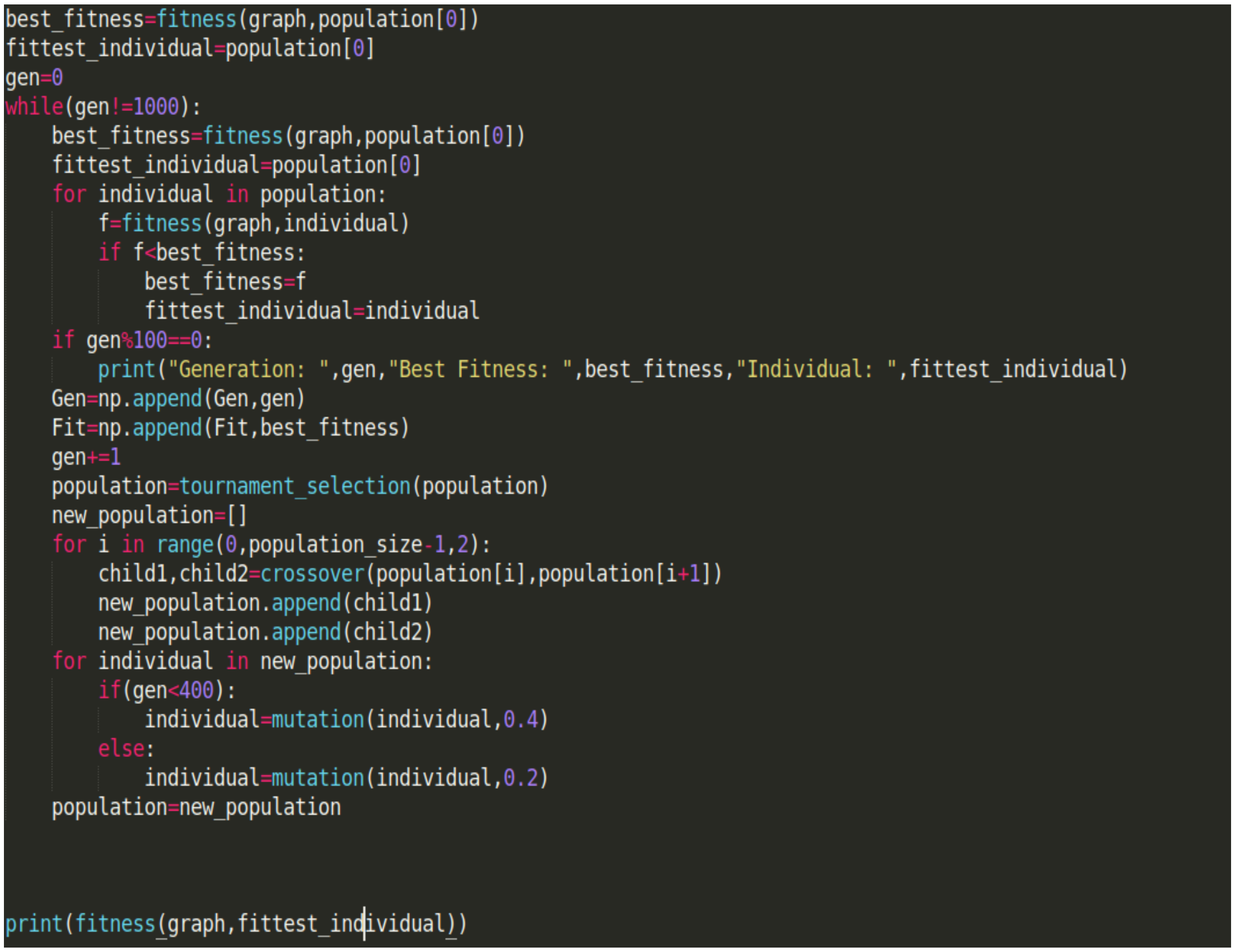
Genetic Algorithm Code:



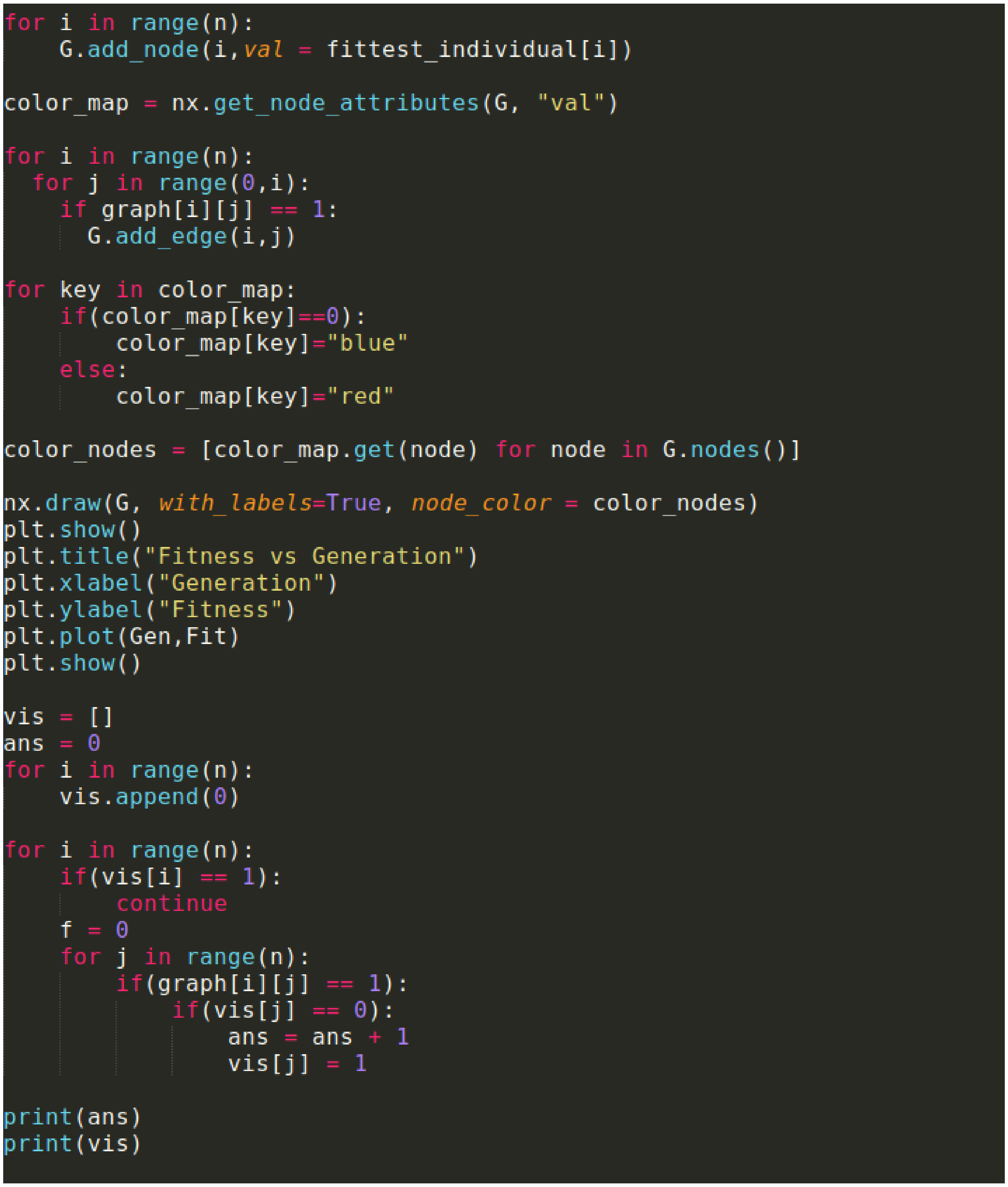
```
import random
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from array import *


n = 40
graph = []
for i in range(n):
 node = []
 for j in range(n):
  p = random.uniform(0,1)
  if(p<0.5):
    node.append(1)
  else:
    node.append(0)
 graph.append(node)

for i in range(n):
 for j in range(0,i):
  graph[i][j] = graph[j][i]

for i in range(n):
 graph[i][i] = 0


G = nx.Graph()

Gen=np.array([])
Fit=np.array([])
def create_individual():
  individual=[]
  for i in range(n):
    individual.append(random.randint(0,1))
  return individual


'''Creating Population'''
population_size=50
generation=0
population=[]
for i in range(population_size):
  individual=create_individual()
  population.append(individual)
```



```python
'''Fitness'''
def fitness(graph,individual):
    fitness=0
    for i in range(n):
        if individual[i]==1:
            fitness+=1
        for j in range(i):
            if graph[i][j]==1:
                if individual[i]==0 and individual[j]==0:
                    fitness+=10
    return fitness


'''Crossover'''
def crossover(parent1,parent2):
    position=random.randint(1,n-2)
    child1=[]
    child2=[]
    for i in range(position+1):
        child1.append(parent1[i])
        child2.append(parent2[i])
    for i in range(position+1,n):
        child1.append(parent2[i])
        child2.append(parent1[i])
    return child1,child2

'''Mutation'''
def mutation(individual,probability):
    check=random.uniform(0,1)
    if check<=probability:
        position=random.randint(0,n-1)
        individual[position]=1-individual[position]
    return individual

'''Tournament Selection'''
def tournament_selection(population):
    new_population=[]
    for j in range(2):
        random.shuffle(population)
        for i in range(0,population_size-1,2):
            if fitness(graph,population[i])<fitness(graph,population[i+1]):
                new_population.append(population[i])
            else:
                new_population.append(population[i+1])
    random.shuffle(new_population)
    return new_population
```



```
best_fitness=fitness(graph,population[0])
fittest_individual=population[0]
gen=0
while(gen!=1000):
    best_fitness=fitness(graph,population[0])
    fittest_individual=population[0]
    for individual in population:
        f=fitness(graph,individual)
        if f<best_fitness:
            best_fitness=f
            fittest_individual=individual
    if gen%100==0:
        print("Generation: ",gen,"Best Fitness: ",best_fitness,"Individual: ",fittest_individual)
    Gen=np.append(Gen,gen)
    Fit=np.append(Fit,best_fitness)
    gen+=1
    population=tournament_selection(population)
    new_population=[]
    for i in range(0,population_size-1,2):
        child1,child2=crossover(population[i],population[i+1])
        new_population.append(child1)
        new_population.append(child2)
    for individual in new_population:
        if(gen<400):
            individual=mutation(individual,0.4)
        else:
            individual=mutation(individual,0.2)
    population=new_population


print(fitness(graph,fittest_individual))
```



```
for i in range(n):
    G.add_node(i,val = fittest_individual[i])

color_map = nx.get_node_attributes(G, "val")

for i in range(n):
  for j in range(0,i):
    if graph[i][j] == 1:
      G.add_edge(i,j)

for key in color_map:
    if(color_map[key]==0):
        color_map[key]="blue"
    else:
        color_map[key]="red"

color_nodes = [color_map.get(node) for node in G.nodes()]

nx.draw(G, with_labels=True, node_color = color_nodes)
plt.show()
plt.title("Fitness vs Generation")
plt.xlabel("Generation")
plt.ylabel("Fitness")
plt.plot(Gen,Fit)
plt.show()

vis = []
ans = 0
for i in range(n):
    vis.append(0)

for i in range(n):
    if(vis[i] == 1):
        continue
    f = 0
    for j in range(n):
        if(graph[i][j] == 1):
            if(vis[j] == 0):
                ans = ans + 1
                vis[j] = 1

print(ans)
print(vis)
```