**NAMAL UNIVERSITY MIANWALI**

**AI PROJECT REPORT**

Prepared by

**M Asad Ayyub**

**BSCS-2019-25**

**The Project**

This project is about image recognition. If we are given two images, A bigger image and a smaller one, this project helps us to locate the smaller picture in a bigger picture. The problem was solved using AI approach as a computer scientist. We used the process of science to produce an algorithm which solves this problem. The algorithm used in this process “Evolutionary Algorithm” is generated inspired by the “Darwin’s Theory of Evolution”.

**Evolutionary Algorithm**

The algorithm used in this project to solve the problem is designed through some scientific process. It is inspired by the Darwin’s Theory of Evolution.

There are four steps to design this algorithm:

1. **Natural Reality**
2. **Theory**
3. **Model**
4. **Application**

This algorithm produces diversity in population just like Darwin described the diversity through process of Selection, crossing and then selects the fittest individuals from the population.

**Background:**

**“Theory Of Evolution”** is the famous theory presented by “**Charles Robert Darwin”** in his famous book “**On the origin of species**”. This theory is the heart of the Evolutionary Algorithm.

* **The Process**

1. **Natural Reality**

The nature is full of diverse species. Why is there so much diversity in the nature? There are thousands of kinds of a flower or living organism. The theory is made on the basis of this reality.

1. **Theory**

All species of organisms arise and develop through the natural selection of small, inherited variations that increase the individual's ability to compete, survive, and reproduce.

The main highlights of Darwin’s Theory used in the Evolutionary Algorithm are:

* Selection
* Crossing
* Mutation
* Survival of the fittest

1. **The Computational Model**

A model is generated from this theory. We are given an image; we select some random population(points) from the bigger image. Then we check for the fittest individuals in this population. We set the population according to their fitness. Then in the next step, the fittest individuals are crossed to produce the next fit generation. Individuals are crossed according to their fitness value to produce the next generation. The fittest individuals combine to produce the next fit individuals. After the crossover process, we mutate the generation produced after crossover to make some variety in the population. We repeat the process until we get some fittest individual and then stop.

**Start**

* **Initialize population**

**Repeat Until Condition is satisfied**

* **Find fitness**
* **Rank the population**
* **Crossover**
* **Mutate**

**Objectives**

The project’s objectives were:

1. Use Scientific process to solve problems.
2. Practice AI as a science.
3. To invoke scientific thoughts by practicing.
4. Recognition of a smaller part of image in a bigger one.

**The Evolutionary Algorithm Comprises of Following Fundamental Functions:**

1. **Population Initialization**

First of all, we initialize some random points in the bigger image, and store them in an array. This function returns an array of population which consists of tuples of row and column co-ordinates (row, column).

**What to pass this function:**

(No of rows in bigger image, no of columns of big image, random co-ordinates we want)

**What this function returns:**

An array of population i.e., the list of random co-ordinates of bigger image.

1. **Fitness\_Evaluation**

This function takes the population array as input and calculates the correlation value for every tuple in the population array. The ‘Correlation’ Function is used for every tuple to calculate its correlation value and then stores co-ordinates with their correlation value and stores them in separate array.

**What to pass this function:**

(Population array)

**What this function returns:**

An array of population in which tuples of coordinates are stored along with their correlation value.

1. **Ranked Population:**

This function ranks/sorts the population on the basis of their correlation value.

It means this function selects the best individuals and place the best in start of array and place them according to their fitness in the array.

**What to pass this function:**

(Population array with fitness values)

**What this function returns:**

Sorted population array.

1. **Crossover**

The crossover operator is similar to crossing concept in Evolutionary theory. We are given a population with its fitness value, we cross the best individuals with best, middle with middle and less fit with less. This will produce a next diverse generation. We separated the co-ordinate into row and column, made their binary, tore them into two random parts and concatenate with the other parts of individual, convert them back into decimal. Hence, new childs are produced after crossover.

**What to pass this function:**

(Ranked Population array with fitness values)

**What this function returns:**

(New population after crossover)

1. **Mutation**

The mutation process is used to produce the variety in population by just changing a bit from 0 to 1 or 1 to 0.

**What to pass this function:**

(Population after crossover)

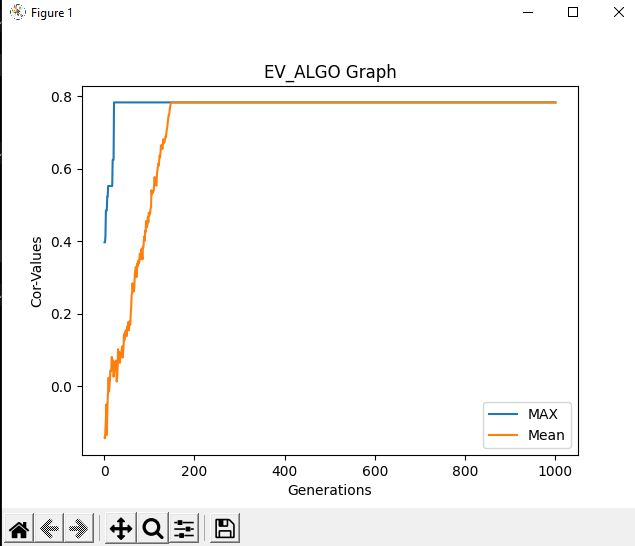
**What this function returns:**

(New population with variety)

**EXPERIMENTS**

1. **Keeping Population size 100 with crossover operation only on 1000 generations.**

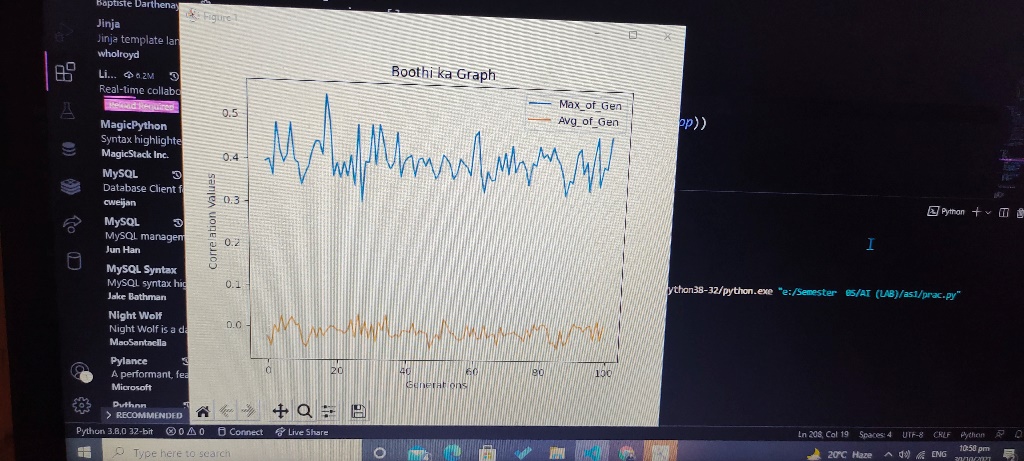
When I initialized the population of 500 and applied only crossover operation, the correlation value increased up to 200 generations and then constant. It means after 200 generations, due to no variety in population the trend became constant.





1. **Keeping Population size 100 with constant crossover operation on 100 generations.**

When I kept the crossover operation constant, i.e., same operation for every individual it returned the graph which was not growing. It was symmetric along x axis.

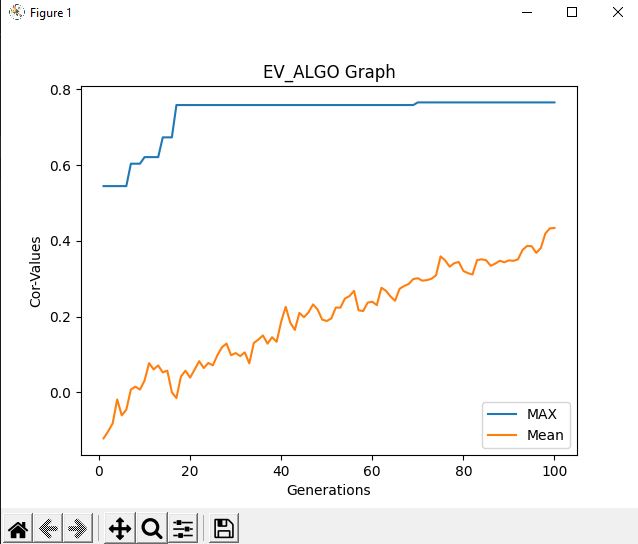


­­

­­­­

1. **Keeping Population size 100 with 100 generations.**

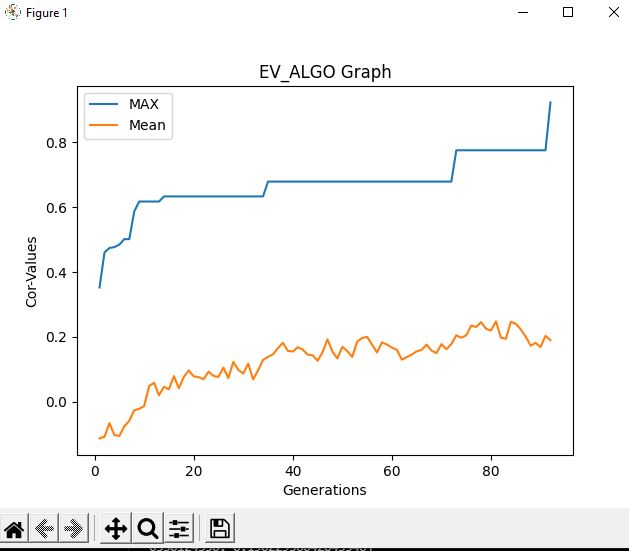
When population is 100 with 100 generations, sometimes it found the threshold value and sometimes it could not because of low number on generations.





1. **Keeping Population size 100 with 1000 generations.**

When I kept population size 100, and run it on 1000 generations, with mutation rate controlled, it gave me the required value in almost 90% cases.





SOURCE CODE

from os import readlink

import random as ran

import numpy as np

from typing import Counter

import matplotlib

import matplotlib.pyplot as plt

from matplotlib import image

import math

import matplotlib.patches as patches

imgbaba = *image.imread*('boothiGray.jpg')

imggrp = *image.imread*('groupGray.jpg')

*def* Sort\_Tuple(*tuple*):

*tuple.sort*(*key* = *lambda x*: *x*[2], *reverse* = *True* )

    return tuple

*def* binary\_conversion(*number*,*bits*):

    a = *bin*(*number*).*replace*("0b"*,* "")

    b = a[::-1]

    while *len*(*b*) < bits:

        b += '0'

    b = b[::-1]

    return(b)

*def* max\_of\_gen(*tup\_list*):

    m\_list = []

    for i in *range*(*len*(*tup\_list*)):

        mem = tup\_list[i]

        ele = mem[2]

*m\_list.append*(*ele*)

    maxo = *max*(*m\_list*)

    return maxo

*def* avg\_tuple(*tuple*):

    counter = 0

    for i in *range*(*len*(*tuple*)):

        mem = tuple[i]

        ele = mem[2]

        counter = counter + ele

    avg = counter/*len*(*tuple*)

    return avg

*def* pop\_initialization(*row*,*column*,*popsize*):

    p\_list = []

    for a in *range* (*popsize*):

        r = *ran.randint*(0*,row*)

        c = *ran.randint*(0*,column*)

        x = (r,c)

*p\_list.append*(*x*)

    return p\_list

*def* correlation(*template*,*img\_part*):

    baba\_mean = *np.mean*(*template*)

    part\_mean = *np.mean*(*img\_part*)

    num = 0

    v1 = 0

    v2 = 0

    for i in *range*(*len*(*img\_part*)):

        temp =template[i]

        sliced = img\_part[i]

        for j in *range*(*len*(*temp*)):

            if j < *len*(*sliced*):

                slice\_value = sliced[j]

                # *print(temp[i])*

                num +=(temp[j] - baba\_mean)\*(slice\_value - part\_mean)

                v1 += (temp[j] - baba\_mean)\*\* 2

                v2 += (slice\_value - part\_mean)\*\*2

                a = v1 \* v2

    dvd = *math.sqrt*(*a*)

    corr = num / dvd

    return corr

*def* fitness\_Eval(*imggrp*,*imgb*,*p\_list*):

    val = []

    for i in *range*(*len*(*p\_list*)):

        point = p\_list[i]

        if point[0]+35 < (imggrp.shape)[0] and point[1]+29 < (imggrp.shape)[1]:

            p\_row,p\_col = point[0],point[1]

            slicedimg = imggrp[point[0]:point[0]+35,point[1]:point[1]+29]

            fitness = *correlation*(*imgb,slicedimg*)

*val.append*(*fitness*)

        else:

            corr = 0

*val.append*(*corr*)

    return val

*def* rank\_pop(*asad*,*values*):

    rankedlist = []

    for a in *range*(*len*(*asad*)):

        member = asad[a]

        x = member[0]

        y = member[1]

        corel\_val = values[a]

        val = x,y,corel\_val

*rankedlist.append*(*val*)

*Sort\_Tuple*(*rankedlist*)

    return rankedlist

*def* Crossover(*rankedlist*):

    listy = []

    no\_1 = 0

    no\_2 = 0

    for a in *range*(*len*(*rankedlist*)):

        if no\_1 == 0:

            no\_1 = rankedlist[a]

        else:

            no\_2 = rankedlist [a]

        if no\_1 != 0 and no\_2 != 0 :

            #*for number 1*

            row\_1 = no\_1[0]

            column\_1 = no\_1[1]

            val\_1 = no\_1[2]

            row1\_b = *binary\_conversion*(*row\_1,*9)

            col1\_b = *binary\_conversion*(*column\_1,*10)

            row1\_str = *str*(*row1\_b*) #*concatenate the numbers , 1st cnvrt in str then conctenate, then int*

            col1\_str = *str*(*col1\_b*)

            new\_num1 = row1\_str+col1\_str #*concatenate the row and column for new gen*

            rand = *ran.randint*(0*,*18) #*random number to slice the  binary no each time*

            new\_num1\_i = new\_num1[:rand] #*slice the concatenated string*

            new\_num1\_ii = new\_num1[rand:] #*slice the concatenated string*

            #*for number 2*

            row\_2 = no\_2[0]

            column\_2 = no\_2[1]

            val\_2 = no\_2[2]

            row2\_b = *binary\_conversion*(*row\_2,*9)

            col2\_b = *binary\_conversion*(*column\_2,*10)

            row2\_str = *str*(*row2\_b*) #*concatenate the numbers , 1st cnvrt in str then conctenate, then int*

            col2\_str = *str*(*col2\_b*)

            new\_num2 = row2\_str+col2\_str #*concatenate the row and column for new gen*

            new\_num2\_i = new\_num2[:rand]

            new\_num2\_ii = new\_num2[rand:]

            new\_num1 = new\_num1\_i + new\_num2\_ii

            new\_num2 = new\_num2\_i + new\_num1\_ii

            new\_child\_1\_i = new\_num1[:9]

            new\_child\_1\_ii = new\_num1[9:]

            new\_child\_2\_i = new\_num2[:9]

            new\_child\_2\_ii = new\_num2[9:]

            child1\_row = *int*(*new\_child\_1\_i,*2) #*convert in binary*

            child1\_col = *int*(*new\_child\_1\_ii,*2)

            child\_1 = (child1\_row,child1\_col,val\_1)

            child2\_row = *int*(*new\_child\_2\_i,*2)

            child2\_col = *int*(*new\_child\_2\_ii,*2)

            child\_2 = (child2\_row, child2\_col,val\_2)

*listy.append*(*child\_1*)

*listy.append*(*child\_2*)

            no\_1 = no\_2 = 0

    return(listy)

*def* mutation(*list*):

    mutated\_list = []

    for a in *range*(*len*(*list*)):

        member = list[a]

        row = member[0]

        column = member[1]

        val = member[2]

        row = *binary\_conversion*(*row,*9)

        col = *binary\_conversion*(*column,*10)

        row\_str = *str*(*row*) #*concatenate the numbers , 1st cnvrt in str then conctenate, then int*

        col\_str = *str*(*col*)

        rc\_str = row\_str+col\_str

        if member[2] > 0.5:

            rando = *ran.randint*(10*,*18)

        else:

            rando = *ran.randint*(1*,*9)

        if rc\_str[rando] == '0':

            rc\_str = rc\_str[:rando] + '1' + rc\_str[rando+1:]

        else:

            rc\_str = rc\_str[:rando] + '0' + rc\_str[rando+1:]

        row = rc\_str[:9]

        col = rc\_str[9:]

        row = *int*(*row,*2)

        col = *int*(*col,*2)

        rc\_val = (row,col,val)

*mutated\_list.append*(*rc\_val*)

    return mutated\_list

*def* E\_algo():

    gen = 0

    max\_gen = 100

    maxim\_cordp = []

    geny  = []

    average = []

    max\_pts = []

    pop = *pop\_initialization*(511*,*1023*,*100)

    while gen <max\_gen:

        cor\_values = (*fitness\_Eval*(*imggrp,imgbaba,pop*))

        ranked = (*rank\_pop*(*pop,cor\_values*))

        crossed = *Crossover*(*ranked*)

        pop = *mutation*(*crossed*)

        avg = *avg\_tuple*(*pop*)

*average.append*(*avg*)

        maxi = pop[0]

*maxim\_cordp.append*(*maxi*)

        maxo = maxi[2]

*max\_pts.append*(*maxo*)

        if maxo >= 0.82:

            break

        gen = gen +1

*geny.append*(*gen*)

    # *average = avg\_tuple(mut\_val\_ranked)*

    maxim\_cordp= *Sort\_Tuple*(*maxim\_cordp*)

    max\_pts = *sorted*(*max\_pts*)

    return maxim\_cordp,average,geny,max\_pts

a = *E\_algo*()

maxim = a[0]

average = a[1]

geny = a[2]

max\_pts = a[3]

max\_coordinate = maxim[0]

max\_x = max\_coordinate[0]

max\_y = max\_coordinate[1]

max\_c = (max\_x,max\_y)

fig,ax = *plt.subplots*(1)

rect = *patches.Rectangle*(*(max\_y,max\_x),*29*,*35*, linewidth* =2, *edgecolor* ='b',*fill* = *False*)

*ax.imshow*(*imggrp, cmap* ="gray")

*ax.add\_patch*(*rect*)

*plt.show*()

*plt.plot*(*max\_pts, label* ="Max\_of\_Gen")

*plt.plot*(*geny,average, label* ="Avg\_of\_Gen")

*plt.xlabel*('Generations')

*plt.ylabel*('Cor-Values')

*plt.title*("EV\_Algo Graph")

*plt.legend*()

*plt.show*()