**Report for Image Analysis and Machine Learning**

Members:

Pan Jiao (6612170086-8)

Leng Ke (6612170081-7)

Zhao Ziyu (6612170092-2)

Xu Jingdi (6612170090-6)

1. **The conclusion of GA problem.**

Genetic Algorithm is a widely used and efficient method of random search based on evolution. A random set of arrays start the search process, and everyone in the arrays is a solution to the problem. These solutions use "fitness" to measure members’ quality in each generation, forming the next generation by selection, crossover and mutation. Each generation keeps the population size constant, and the principle of choice is the higher the fitness, the greater the probability of being selected. After several generations, the algorithm converges to the best solution.

1. **Discuss the effect of varying the four input parameters population size, crossover rate, mutation rate, and number of GA iterations.**
2. Varying population size when crossover rate, mutation rate and number of GA iterations are equal to 60, 5 and 100 respectively.

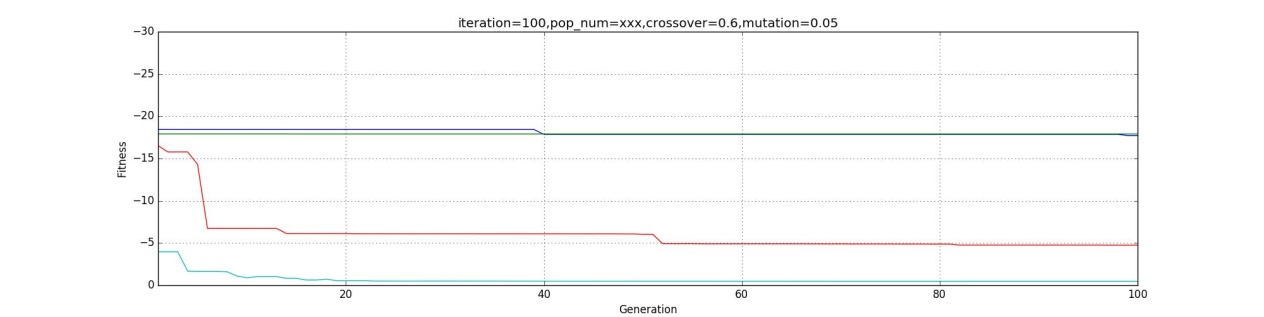


Figure 1 The best fitness in each generation when varying population size

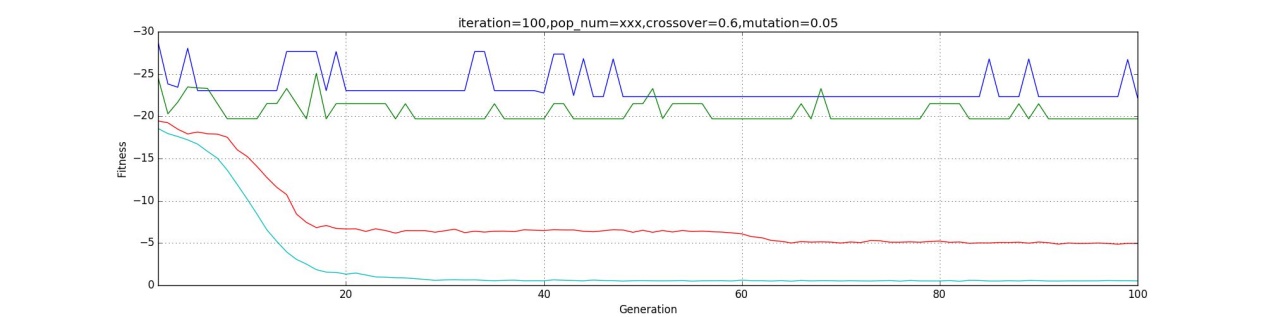


Figure 2 The average fitness in each generation when varying population size

The blue line, green line, red line and cyan line are the fitness when population size equal 4, 10, 100 and 1000.

From the graphs, we can see the change when the population size is getting bigger:

1. There are more elements to search for the result, so the searching speed is higher, and GA conversion time is becoming shorter.
2. The possibility of finding the best fitness is higher, so the optimized result is smaller, which means the result is closer to what we expect.
3. Varying crossover rate when population size, mutation rate and number of GA iterations are equal to 100, 5 and 100 respectively.

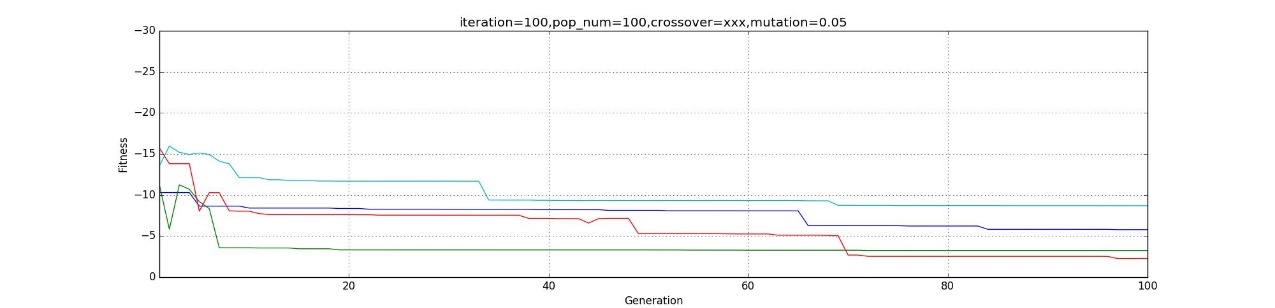


Figure 3 The best fitness in each generation when varying crossover rate

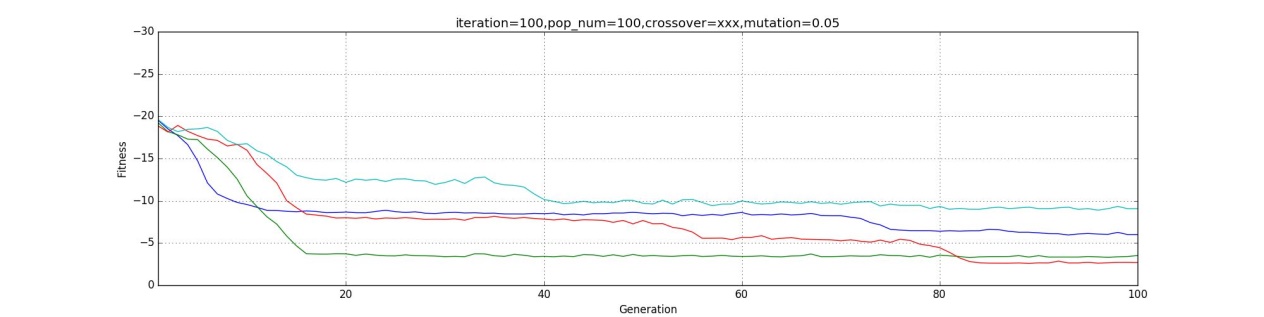


Figure 4 The average fitness in each generation when varying crossover rate

The blue line, green line, red line and cyan line are the fitness when crossover rate equal 0.0, 0.2, 0.6 and 1.0.

From the graphs, we can see the change when the crossover rate is getting bigger:

1. The possibility of new DNA in next generation is getting higher, so the fitness can be found faster, and GA conversion time should be smaller too. In this case, the new DNA is produced too much, so the good DNA is possibly covered by them and the GA conversion time will be long.
2. Because of the high update speed, the DNA with the best fitness may be eliminated prematurely, so the optimized result may not be the best answer.
3. Varying mutation rate when population size, crossover rate and number of GA iterations are equal to 100, 60 and 100 respectively.

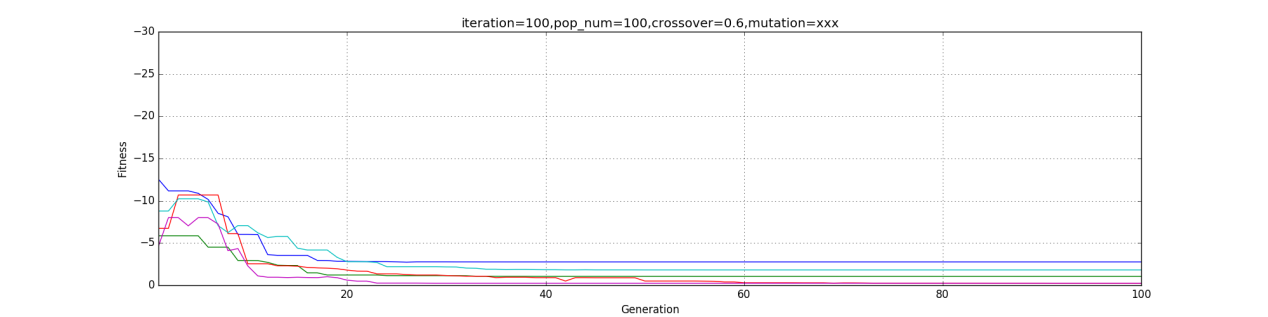


Figure 5 The best fitness in each generation when varying mutation rate as pop equals 100

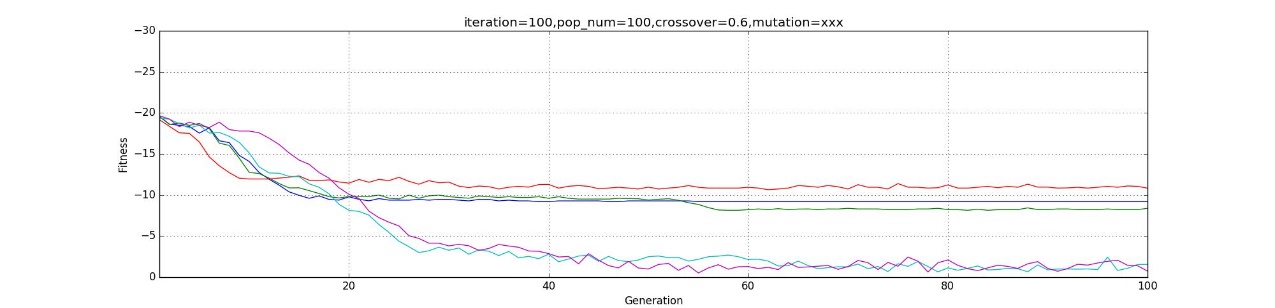


Figure 6 The average fitness in each generation varying mutation rate as pop equals 100

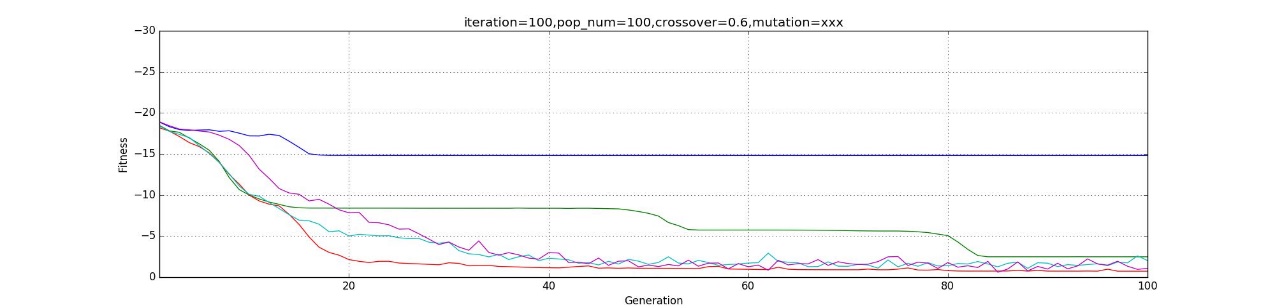


Figure 8 The average fitness in each generation varying mutation rate as pop equals 600

The blue line, green line, red line, cyan line and the purple line are the fitness when mutation rate equal 0.0, 0.01, 0.05, 0.5 and 1.0.

Because of the unstable results, varying mutation when population equals 100 is not good for analyzing. After lots of tests, we transform population to 600 which the best fitness can show as mutation equals 0.05 and can see the change when the mutation rate is getting bigger:

1. New DNA is created more in next generation, which caused the possibility of finding the best fitness instability, so the GA conversion time may be longer.
2. Although new DNA is produced, the best results may be overwritten in the new generation, so the best fitness may be covered in the update, and the optimized result may not be the best answer.
3. If the mutation is too high, it is hard to keep the fitness stable.
4. Varying number of GA iterations when population size, crossover rate and mutation rate are equal to 100, 60 and 5 respectively.

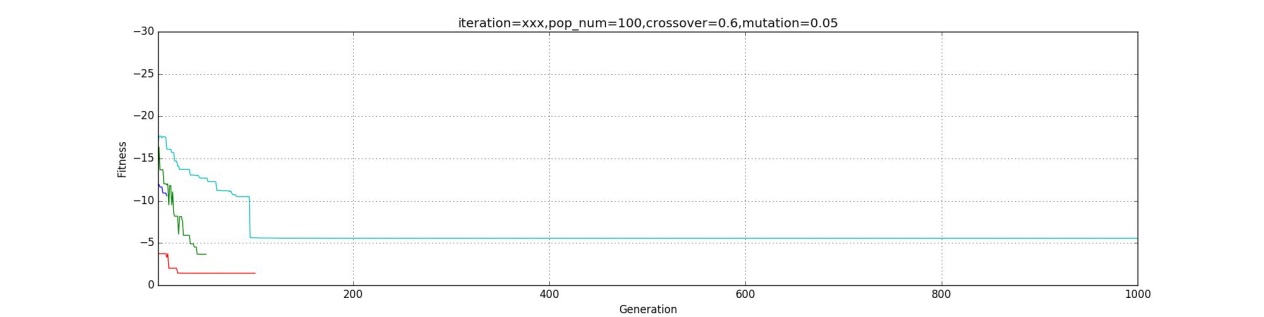


Figure 9 The best fitness in each generation when varying number of GA iterations

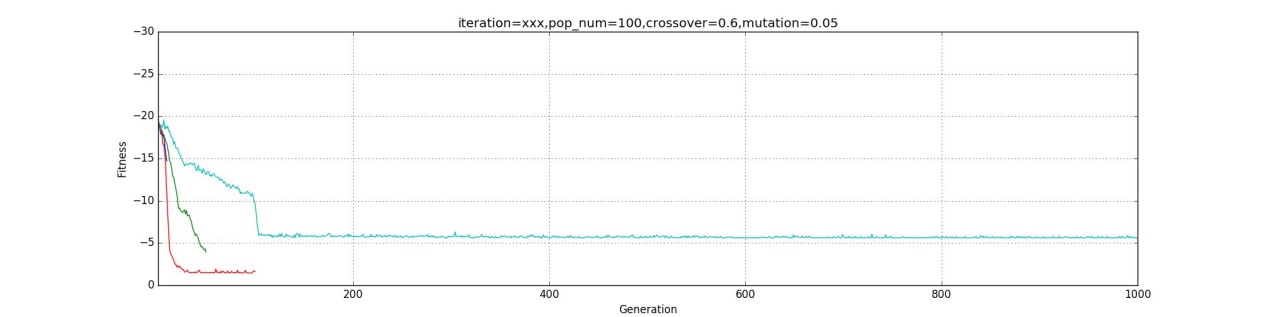


Figure 10 The average fitness in each generation when varying number of GA iterations

The blue line, green line, red line and cyan line are the fitness when the number of GA iterations equal 10, 50, 100 and 1000.

From the graphs, we can see the change when the number of GA iterations is getting bigger:

1. The GA conversion time is longer.
2. The optimized result should be better, but the new DNAs are not enough, so the best fitness is hard to find. The graphs show that every parameter is important when finding the best fitness.
3. **The pictures below are the output results when the numbers of GA iterations are 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100.**

Table 1

|  |  |  |  |
| --- | --- | --- | --- |
|  | The created noisy Lena | The created noise | The best fitness in this generation |
| Generation 10 |  |  |  |
| Generation 20 |  |  |  |
| Generation 30 |  |  |  |
| Generation 40 |  |  |  |
| Generation 50 |  |  |  |
| Generation 60 |  |  |  |
| Generation 70 |  |  |  |
| Generation 80 |  |  |  |
| Generation 90 |  |  |  |
| Generation 100 |  |  |  |

1. After the analysis, we can find out that the fitness can be close to zero if each parameter is taken properly. In this case, when population size is 1000, crossover rate is 0.6, mutation rate is 0.05, and number of GA iterations is 100, we can get the closest answer to GA problem.

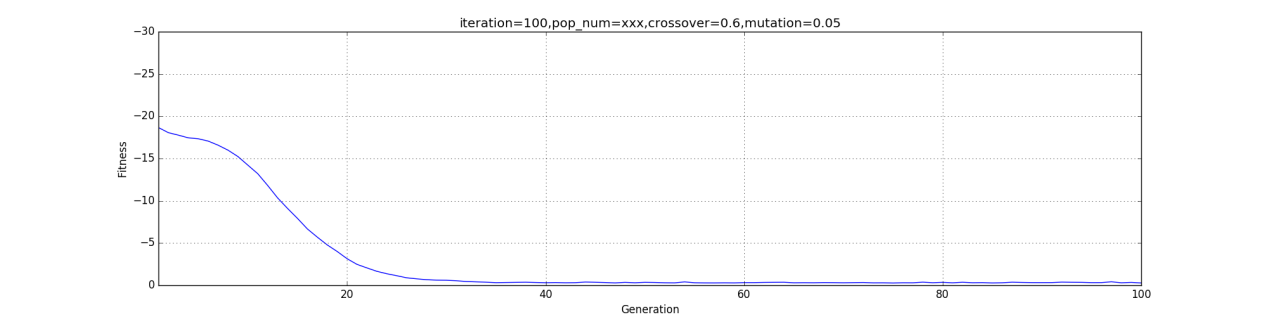


Figure 11 The closest answer in this case

Table 2 shows the best parameters values of the best fitness.

Table 2 the final parameters

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | NoiseAmp | NoiseFreqRow | NoiseFreqCol |
| Value | 28.0115 | 0.0075 | 0.0051 |

The program of Lena problem is in the appendix.

Appendix 1:

Main program

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

"""

Created on Fri Oct 13 13:51:49 2017

@author: tanakalab03

"""

import os

import matplotlib.pyplot as plt

from random import random, randint

import lena

import matplotlib.pyplot as plt

import matplotlib.cm as cm

from scipy import misc

dna\_bits =192

#def random\_individual(n) : return [coin (0.5) for i in range(n)]

#random 0,1

def coin(p)

x = random()

if x<p: return 1

else: return 0

#

def random\_individual(n):

s=[]

for i in range(n):

v= coin(0.5)

s.append(v)

return s

def fitness(s):

f=0

for v in s:

if v==1 : f+=1

return f

def initial\_population(N,n):

p=[]

for i in range(N):

c=random\_individual(n)

p.append(c)

return p

'''

print("This is generation 0")

for s in p:

print (s,fitness(s))

pop\_fit =[]

for s in p:

s\_f = s,fitness(s)

pop\_fit.append(s\_f)

'''

#print(len(pop\_fit)) 50

#print("i dont know")

#print (pop\_fit) 是[([s1],f1),([s2],f2)。。。。。。([sn],fn)]

def crossover(c):

c1,c2=c

n = len(c1)

p = randint(0,n)

#print (p) # can see where to cross

c1[:3]

c1[3:]

nc1 = c1[:p] + c2[p:]

nc2 = c2[:p] + c1[p:]

return nc1, nc2

def tournament(c1\_f1,c2\_f2):

c1,f1 =c1\_f1

c2,f2 =c2\_f2

if f1>=f2 : return c1

return c2

def selection(population\_fitness):

N=len(population\_fitness)

#print(N)

new\_population = []

for i in range(N):

i1,i2 = randint(0,N-1),randint(0,N-1)

c1\_f1 = population\_fitness[i1]

#print(i1,i2)

c2\_f2 = population\_fitness[i2]

nc1 = tournament(c1\_f1,c2\_f2)

new\_population.append(nc1)

return new\_population

def select\_parents(population\_fitness):

new\_population=selection(population\_fitness)

#print(new\_population)

next\_population=[]

parent\_pair=[]

i=0

while(i<len(new\_population)):

#print(len(new\_population))

parent\_pair=new\_population[i],new\_population[i+1]

#print(new\_population[i])

#print(parent\_pair)

i=i+2

next\_population.append(parent\_pair)

#print(next\_population)

#print(len(next\_population))

return next\_population

def mutation(c1):

n=len(c1)

p=randint(0,n-1)

nc1=c1[:]

nc1[p]=1-nc1[p] # change 1 to 0 and 0 to 1 amazing

return nc1

def check\_stop(p):

for s in p:

ch,chs=s

#print("ch=/.................",ch)

#print("chs/////////////////",chs)

print(ch,fitness(ch))

if(chs==dna\_bits):

return 1;

#else:

# return 0

def run(iter,pop,p\_c,p\_m):

n = pop

prob\_c=p\_c

prob\_m=p\_m

population = initial\_population(n,dna\_bits)

generation=1

#print(population)

fitness\_average=[]

fitness\_history=[]

iter\_num=iter

iter\_num\_visual =iter\_num

plt.close()

fig=plt.figure(num='visualize',figsize=(36,36))

plt.grid(True)

plt.ion()

while iter\_num>=0:

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*This is Generation:",generation,"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

'''

This is maxone

'''

#fits\_pops = [(ch,fitness(ch)) for ch in population]

#fitness\_history.append(visualize(fits\_pops))

#if check\_stop(fits\_pops): break

'''

This is lena problem

'''

fits\_pops = [(ch,lena.lena\_fitness(ch)) for ch in population]

mini,noise,average= visualize(fits\_pops)

fitness\_history.append(mini)

fitness\_average.append(average)

lena.mat\_visual(noise,fitness\_history,iter\_num\_visual)

if((iter\_num%(iter\_num\_visual/10))==0):

misc.toimage(noise,cmax=255,cmin=0).save(os.path.abspath('.')+"\images\\noise\_"+str(iter\_num)+".png")

lena\_out=lena.GBL.lena\_noisy-noise

miss\_out=lena.GBL.lena-lena\_out

misc.toimage(lena\_out,cmax=255,cmin=0).save(os.path.abspath('.')+"\images\\lean\_out\_"+str(iter\_num)+".png")

misc.toimage(miss\_out,cmax=255,cmin=0).save(os.path.abspath('.')+"\images\\miss\_out\_"+str(iter\_num)+".png")

#if lena.lena\_check\_stop(fits\_pops): break

population = breed\_population(fits\_pops,prob\_c,prob\_m)

iter\_num-=1

generation+=1

plt.pause(0.05)

#plt.savefig("figure.png")

return population,fitness\_history,fitness\_average

def test\_iter():

fit\_draw=[]

fit\_ave=[]

iter\_num\_list=[10,50,100,1000]

#color\_list=['b','g','r','k']

for each in iter\_num\_list:

iter\_num=each

p,f,a=run(iter\_num,100,0.6,0.05)

fit\_draw.append(f)

fit\_ave.append(a)

#s=zip(iter\_num\_list,fit\_draw)

fig\_iter=plt.figure(num='iteration',figsize=(20,5))

plt.title('iteration=xxx,pop\_num=100,crossover=0.6,mutation=0.05')

plt.xlabel('Generation')

plt.ylabel('Fitness')

plt.grid(True)

#plt.xlim((1,20))

#plt.ylim((-25,0))

#i=0

for each in fit\_draw:

draw(each,1000)

#draw(each,color\_list[i])

#i=i+1

fig\_iter.savefig("loss\_iter.png")

plt.clf()

fig\_iter\_ave=plt.figure(num='iteration',figsize=(20,5))

plt.title('iteration=xxx,pop\_num=100,crossover=0.6,mutation=0.05')

plt.xlabel('Generation')

plt.ylabel('Fitness')

plt.grid(True)

#plt.xlim((1,20))

#plt.ylim((-25,0))

#i=0

for each in fit\_ave:

draw(each,1000)

#draw(each,color\_list[i])

#i=i+1

fig\_iter\_ave.savefig("loss\_iter\_ave.png")

def test\_pop():

fit\_draw=[]

fit\_ave=[]

pop\_num\_list=[4,10,100,1000]

for each in pop\_num\_list:

pop\_num=each

p,f,a=run(100,pop\_num,0.6,0.05)

fit\_draw.append(f)

fit\_ave.append(a)

#s=zip(iter\_num\_list,fit\_draw)

fig\_iter=plt.figure(num='iteration',figsize=(20,5))

plt.xlabel('Generation')

plt.ylabel('Fitness')

plt.title('iteration=100,pop\_num=xxx,crossover=0.6,mutation=0.05')

plt.grid(True)

#plt.xlim((1,20))

#plt.ylim((-25,0))

for each in fit\_draw:

draw(each,100)

fig\_iter.savefig("loss\_pop.png")

plt.clf()

fig\_iter\_ave=plt.figure(num='iteration',figsize=(20,5))

plt.title('iteration=100,pop\_num=xxx,crossover=0.6,mutation=0.05')

plt.xlabel('Generation')

plt.ylabel('Fitness')

plt.grid(True)

#plt.xlim((1,20))

#plt.ylim((-25,0))

#i=0

for each in fit\_ave:

draw(each,100)

#plt.plot(range(len(each)),each)

# fig\_iter\_ave.axis([1,10,0,-30])

#fig\_iter\_ave.show()

fig\_iter\_ave.savefig("loss\_pop\_ave.png")

def test\_cross():

fit\_draw=[]

fit\_ave=[]

cross\_num\_list=[0.0,0.2,0.6,1.0]

for each in cross\_num\_list:

cross\_num=each

p,f,a=run(100,100,cross\_num,0.05)

fit\_draw.append(f)

fit\_ave.append(a)

#s=zip(iter\_num\_list,fit\_draw)

fig\_iter=plt.figure(num='iteration',figsize=(20,5))

plt.title('iteration=100,pop\_num=100,crossover=xxx,mutation=0.05')

plt.xlabel('Generation')

plt.ylabel('Fitness')

plt.grid(True)

#plt.xlim((1,20))

#plt.ylim((-25,0))

for each in fit\_draw:

draw(each,100)

fig\_iter.savefig("loss\_cross.png")

plt.clf()

fig\_iter\_ave=plt.figure(num='iteration',figsize=(20,5))

plt.title('iteration=100,pop\_num=100,crossover=xxx,mutation=0.05')

plt.xlabel('Generation')

plt.ylabel('Fitness')

plt.grid(True)

#plt.xlim((1,20))

#plt.ylim((-25,0))

#i=0

for each in fit\_ave:

draw(each,100)

#plt.plot(range(len(each)),each)

# fig\_iter\_ave.axis([1,10,0,-30])

#fig\_iter\_ave.show()

fig\_iter\_ave.savefig("loss\_cross\_ave.png")

def test\_mut():

fit\_draw=[]

fit\_ave=[]

mutation\_num\_list=[0.0,0.01,0.05,0.5,1]

for each in mutation\_num\_list:

mutation\_num=each

p,f,a=run(100,100,0.6,mutation\_num)

fit\_draw.append(f)

fit\_ave.append(a)

#s=zip(iter\_num\_list,fit\_draw)

fig\_iter=plt.figure(num='iteration',figsize=(20,5))

plt.title('iteration=100,pop\_num=100,crossover=0.6,mutation=xxx')

plt.xlabel('Generation')

plt.ylabel('Fitness')

plt.grid(True)

#plt.xlim((1,20))

#plt.ylim((-25,0))

for each in fit\_draw:

draw(each,100)

fig\_iter.savefig("loss\_mut.png")

plt.clf()

fig\_iter\_ave=plt.figure(num='iteration',figsize=(20,5))

plt.title('iteration=100,pop\_num=100,crossover=0.6,mutation=xxx')

plt.xlabel('Generation')

plt.ylabel('Fitness')

plt.grid(True)

#plt.xlim((1,20))

#plt.ylim((-25,0))

#i=0

for each in fit\_ave:

draw(each,100)

#plt.plot(range(len(each)),each)

# fig\_iter\_ave.axis([1,10,0,-30])

#fig\_iter\_ave.show()

fig\_iter\_ave.savefig("loss\_mut\_ave.png")

def breed\_population(fitness\_population,prob\_c,prob\_m):

parent\_pairs = select\_parents(fitness\_population)

size = len(parent\_pairs)

next\_population = []

for k in range(size) :

parents = parent\_pairs[k]

cross = random() < prob\_c

children = crossover(parents) if cross else parents

for ch in children:

mutate = random() < prob\_m

next\_population.append(mutation(ch) if mutate else ch)

'''

This is printer for lena problem.

'''

#lena.lena\_print(next\_population)

return next\_population

def visualize(p):

mini=100

sum=0

temp=[]

for s in p:

ch,chs=s

sum=sum+chs

if(abs(chs)<abs(mini)):

sum=sum+chs

mini=chs

temp=ch

noise\_params = lena.get\_params(temp)

original\_lena, noise, lena\_noisy = lena.corrupt\_image(lena.GBL.lena,lena.size,noise\_params)

#lena.mat\_visual(noise)

#print("dddddd")

#print(mini)

#print(sum)

#print(len(p))

average=sum/len(p)

print(lena.get\_params(temp),mini)

return mini,noise,average

def draw(p,ymax):

plt.plot(range(len(p)),p)

plt.axis([1,ymax,0,-30])

plt.show()

lena.lena\_init()

test\_iter()

test\_pop()

test\_cross()

test\_mut()

input('Press Enter to exit...')

Appendix 2:

Support program

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

"""

Created on Wed Nov 8 12:55:07 2017

@author: panjiao

"""

import numpy as np

import math

from random import random, randint

from PIL import Image

from scipy import misc

import matplotlib.pyplot as plt

import matplotlib.cm as cm

size=[512,512]

class img:

lena = []

lena\_noisy = []

lena\_N = []

DTYPE = np.float

def imread(fn, dtype=DTYPE) : # img as array

return np.array(Image.open(fn).convert('L'), dtype=dtype)

# return scipy.misc.imread(fn).astype(dtype)

# return cv2.imread(fn, cv2.IMREAD\_GRAYSCALE).astype(dtype)

def corrupt\_image(im,size,noise\_params) :

# read image

signal = im if type(im) != type("") else imread(im)

# noise

noise = make\_noise(size,noise\_params)

# corrupt it

signal\_noisy = signal + noise

# voila!

return signal, noise, signal\_noisy

def get\_value(max\_value,s):

size=len(s)-1

#print(size)

value\_2 = 0.0

for each in s:

#print(each,size)

value\_2 += each\*(math.pow(2,size))

#print(value\_2)

#print(value\_2)

size-=1

#print(value\_2)

value = 0.0

value = value\_2/(math.pow(2,64))\* max\_value

return value

def get\_params(s):

part1=s[:64]

p1=get\_value(30,part1)

part2=s[64:128]

p2=get\_value(0.01,part2)

#print(p2)

#print(len(part2))

#print(part2)

part3=s[128:]

p3=get\_value(0.01,part3)

#print(p3)

p=p1,p2,p3

return p

def make\_noise(size, params) :

NoiseAmp, NoiseFreqRow, NoiseFreqCol = params

h, w = size

zero\_offset = 0

zero\_offset = 1 # Matlab starts with 1

y = np.arange(h) + zero\_offset

x = np.arange(w) + zero\_offset

col, row = np.meshgrid(x, y, sparse=True)

noise = NoiseAmp \* np.sin(2\*np.pi \* NoiseFreqRow \* row + 2\*np.pi \* NoiseFreqCol \* col)

#for each in noise:

#print(each)

return noise

def lena\_init() :

GBL.lena = imread("lena.png")

GBL.lena\_noisy = imread("lena.png\_noisy\_NA\_XXX\_NFRow\_XXX\_NFCol\_XXX.png")

GBL.lena\_N = np.prod(GBL.lena.shape[:2])

def lena\_fitness(gene) :

noise\_params = get\_params(gene)

lena, noise, lena\_noisy = corrupt\_image(GBL.lena,size,noise\_params)

noisy\_diff = GBL.lena\_noisy - lena\_noisy

noisy\_diff = np.sum(np.abs(noisy\_diff)) / GBL.lena\_N # normalized

fitness = - noisy\_diff # negative / minimize

#mat\_visual(noise)

return fitness

def lena\_print(next\_population):

mini=100

temp=[]

for each in next\_population:

if(abs(lena\_fitness(each))<mini):

mini=lena\_fitness(each)

temp=each

print(get\_params(temp),mini)

return next\_population

def lena\_check\_stop(p):

for s in p:

ch,chs=s

noise=make\_noise(size,get\_params(ch))

lena\_out=GBL.lena\_noisy-noise

miss\_out=GBL.lena-lena\_out

if(abs(chs)<=0.5):

print(get\_params(ch),chs)

misc.toimage(noise,cmax=255,cmin=0).save("noise.png")

misc.toimage(lena\_out,cmax=255,cmin=0).save("lean\_out.png")

misc.toimage(miss\_out,cmax=255,cmin=0).save("miss\_out.png")

return 1

GBL=img()

lena\_init()

def draw(p,ymax):

plt.plot(range(len(p)),p)

plt.axis([1,ymax,0,-30])

plt.show()

def mat\_visual(noise,fitness\_history,iter\_num\_visual):

loss=plt.subplot(3,1,1)

plt.title('fitness')

plt.axis([1,iter\_num\_visual,0,-25])

draw(fitness\_history,len(fitness\_history))

loss.grid(True)

#loss.figure.savefig("loss")

plt.subplot(3,3,4)

plt.title('original lena')

plt.imshow(GBL.lena,cmap=cm.gray)

plt.subplot(3,3,5)

plt.title('original noisy\_lena')

plt.imshow(GBL.lena\_noisy,cmap=cm.gray)

plt.subplot(3,3,6)

plt.title('original noise')

plt.imshow(GBL.lena\_noisy-GBL.lena,cmap=cm.gray)

plt.subplot(3,3,7)

plt.title('diff')

plt.imshow((GBL.lena\_noisy-GBL.lena)-noise,cmap=cm.gray)

plt.subplot(3,3,8)

plt.title('lena\_noisy')

plt.imshow(noise+GBL.lena,cmap=cm.gray)

plt.subplot(3,3,9)

plt.title('noise')

plt.imshow(noise,cmap=cm.gray)