Part 2

1)

The Baldwin Effect:

we created a new class to observe the Baldwin effect

```
class Baldwin:
    def __init__(self,args):
        self.args = args
        self.tsize = len(self.args.GA_TARGET)
        self.esize = int(self.args.GA_POPSIZE * self.args.GA_ELITRATE)
```

First we chose a string from the alphabet {0,1} :

```
please enter problem number:
1 for String matching
2 for Nqueens
3 for Minimal Conflicts
4 for baldwin
4
please enter text: 1011101001
please enter crossing type: Uniform
```

And then we initialized the population:

```
def init_population(self,population:list,buffer:list):
    for i in range(self.args.GA_POPSIZE):
        str1=""

        for j in range(self.tsize):
            x = randint(0, 3)
            if(x==0):
                str1+='0'
            if(x==1):
                str1+='1'
            if(x==2or x==3):
                str1+='?'
        population[i]=Struct(str1,0)
```

We gave a 25% probability for a bit to be ‘0’, 25% probability for the bit to be ‘1’ and 50% probability for ‘?’

And then we used the following learning algorithm to calculate the fitness of each pattern using the equation

$$f(n) = 1 + (19 * n)/1000$$

```
def calc_fitness(self,population):
    incorrect=0
    fixed=0
    tempstr=""
    for pop in population:
        pop.fitness = 1
        fixed += self.diff_letters(pop.str, self.args.GA_TARGET)
        for i in range(1000):
            tempstr=pop.str
            for j in range(self.tsize):
                if(pop.str[j]=='?'):
                    x = randint(0, 1)
                    if (x == 0):
                        tempstr=tempstr[:j]+'0'+tempstr[j+1:]
                    if (x == 1):
                        tempstr=tempstr[:j]+'1'+tempstr[j+1:]
            incorrect += self.diff_letters(tempstr, self.args.GA_TARGET)
            if(tempstr==self.args.GA_TARGET):
                pop.fitness=1+(19*(1000-i))/1000
                break
            # incorrect+=self.diff_letters(tempstr,self.args.GA_TARGET)
    incorrectper=incorrect/(1000*self.tsize*self.args.GA_POPSIZE)
    correctper=1-incorrectper
    fixedper=1-(fixed/(self.tsize*self.args.GA_POPSIZE))
    print("incorrect position percentage:",incorrectper)
    print("correct position percentage:", correctper)
    print("fixed position percentage:", fixedper)
    return incorrectper,correctper,fixedper
```

So we will go over each string in the population 1000 times and if a bit in a string is equal to ‘?’ then we will randomly change it to ‘0’ or ‘1’ and check if the new string is equal to the target string

And n will be the 1000-(number of iteration required)

And after we are done we will then create a new generation

Using one of the crossover methods we learned and the RWS selection algorithm

And we also kept track of the percentage of the correct and incorrect positions and the percentage of the learned bits

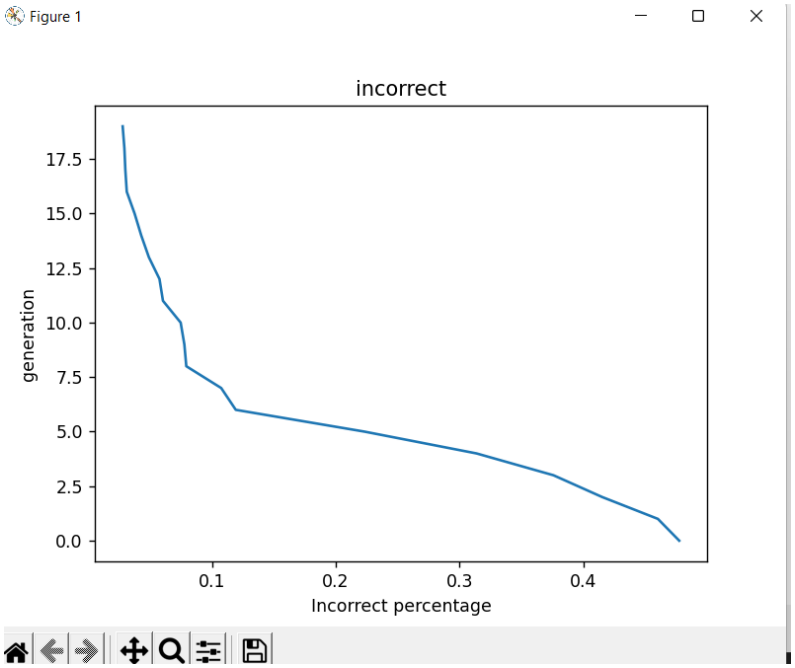


Figure 1
incorrect
generation
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.1
0.2
0.3
0.4
Incorrect percentage

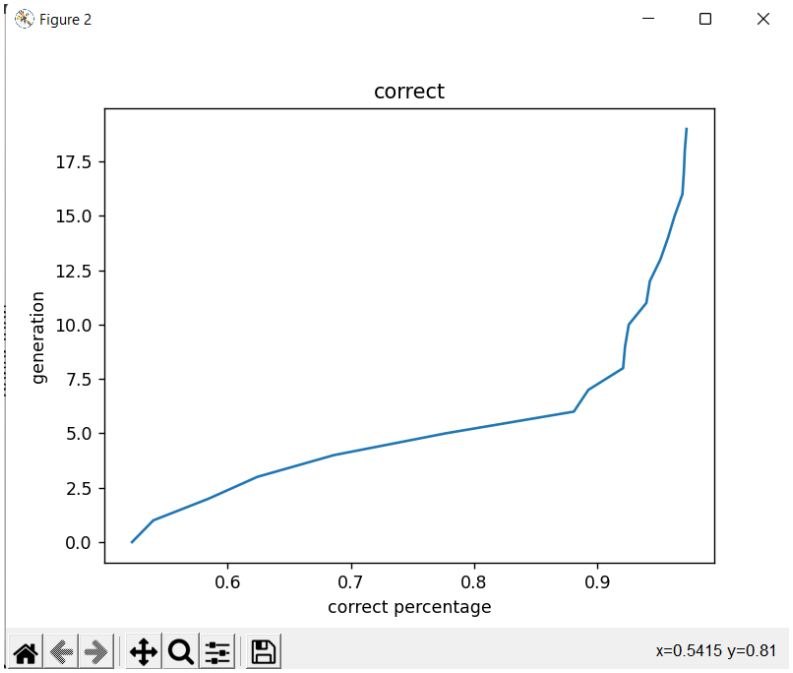
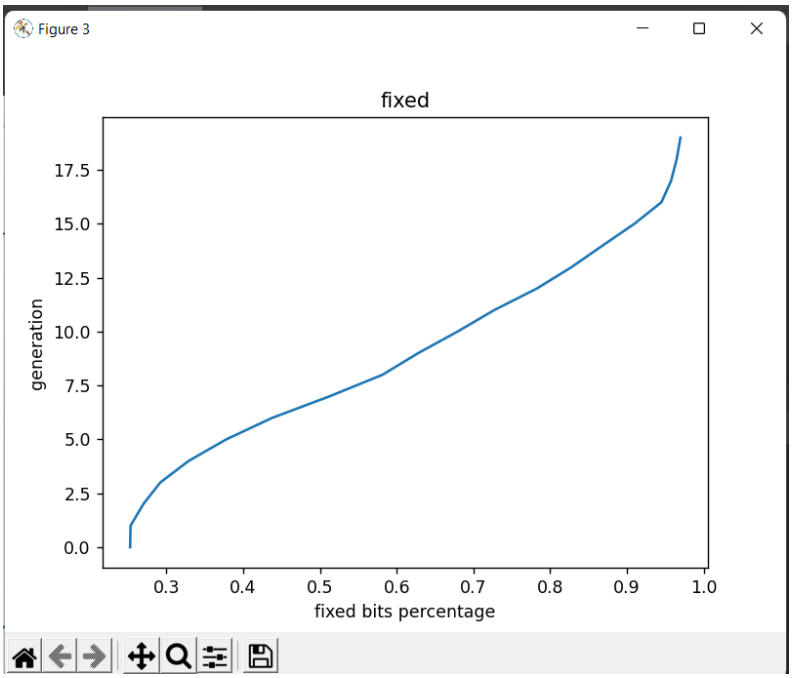


Figure 2
correct
generation
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.6
0.7
0.8
0.9
correct percentage
x=0.5415 y=0.81



And as we can see with every generation the percentage of the correct positions increase since we choose the strings with the highest fitness and therefor the percentage of the incorrect positions decline for the same reason

Since the calc_fitness function returns for each generation those 3 variables after we finished going over all generation we added all the values to their arrays and used the matplotlib library to draw the charts

```
        incorrect,correct,fixed=ga.calc_fitness(population)
        incorrectarr.append(incorrect)
        correctarr.append(correct)
        fixedarr.append(fixed)
    ga.sort_by_fitness(population)
    ga.print_best(population)
```

2)

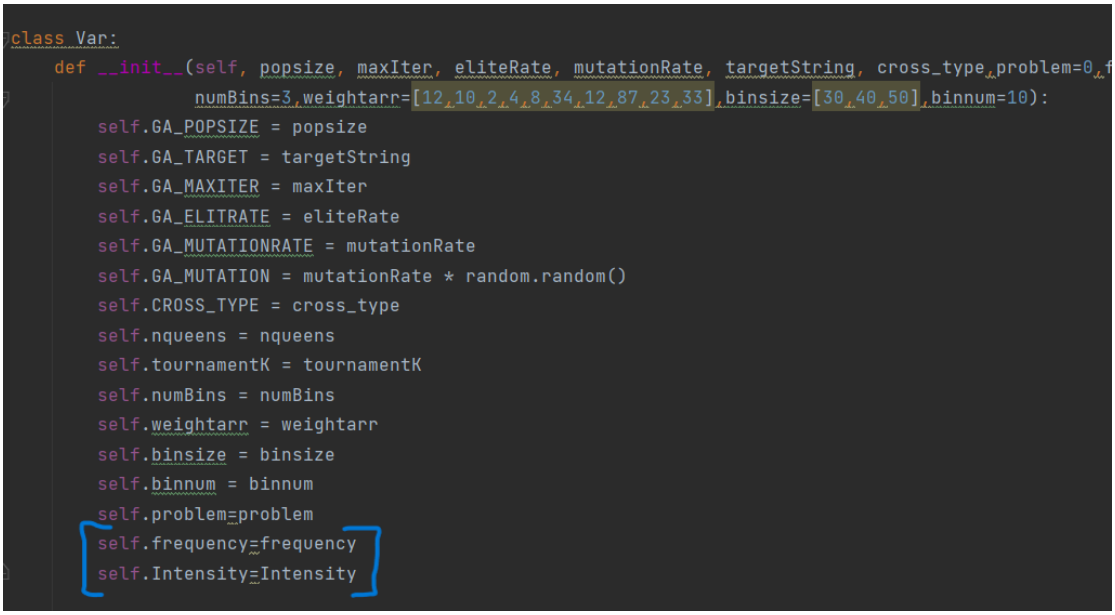
Hybrid GA-MA with local search

First we added 2 more arguments to our arguments list

Frequency: which means how often will we run the learning algorithm

Intensity: how many times will we run the learning algorithm for each time we do

And for every learning algorithm depending on the algorithm itself and the type of problem we will send to the function another variable that is, what part of the population do we want to go through the learning algorithm



```
class Var:
    def __init__(self, popsize, maxIter, eliteRate, mutationRate, targetString, cross_type,problem=0,f
                 numBins=3,weightarr=[12,10,2,4,8,34,12,87,23,33],binsize=[30,40,50],binnum=10):
        self.GA_POPSIZE = popsize
        self.GA_TARGET = targetString
        self.GA_MAXITER = maxIter
        self.GA_ELITRATE = eliteRate
        self.GA_MUTATIONRATE = mutationRate
        self.GA_MUTATION = mutationRate * random.random()
        self.CROSS_TYPE = cross_type
        self.nqueens = nqueens
        self.tournamentK = tournamentK
        self.numBins = numBins
        self.weightarr = weightarr
        self.binsize = binsize
        self.binnum = binnum
        self.problem=problem
        self.frequency=frequency
        self.Intensity=Intensity
```

And then we updated our gene class to be Agent and we added two variables

1)learningfit: a fitness for the learning progress

2)learningalgo: what algorithm and heuristic do we want to use as a learning algorithm

```
class Agent:
    def __init__(self, str, fitness,learningfit,learningalgo):
        self.str = str
        self.fitness = fitness
        self.age = 0
        self.permut = []
        self.learningfit=learningfit
        self.learningalgo=learningalgo

    def getString(self):
        return self.str

    def getFitness(self):
        return self.fitness

    def setString(self, string):
        self.str = string

    def setFitness(self, fitness):
        self.fitness = fitness
```

And then we implemented 3 learning algorithms

1) hill climbing -the standard hill climbing algorithm follows the first string that is better than the current one

2) steepest ascent: this is a hill climbing variant that always follows the best string out of all neighbor strings

3) random walk: changes each bit in the string by +1/-1

Note: we used the same function for hill climbing and steepest ascent because of the similarities, we added a parameter that says which algorithm are we using and if it's the standard hill climbing then we added a break command for when the algorithm finds the first better string

```
def randomwalk(self,intensity: int, population: list,subpop:int,h:int):
    counter=-1
    for pop in population:
        counter+=1

        for i in range(intensity):
            tempstr = pop.str
            for j in range(self.tsize):
                if(j%h==0):
                    x = randint(0, 1)
                    tempstr = tempstr[:j] + chr((ord(tempstr[j])+x)) + tempstr[j + 1:]

            if (tempstr == self.args.GA_TARGET and pop.fitness>1):
                pop.fitness = pop.fitness-1
                break
        if(counter>=subpop):
            break
```

```
def hillclimning(self,population,intensity,subpop,steepest:bool,h):
    counter = -1
    for pop in population:
        counter += 1
        for i in range(intensity):
            tempstr = pop.str
            bestfit = pop.fitness
            for j in range(self.tsize):
                if (j % h == 0):
                    x = randint(0, 1)
                    tempstr = tempstr[:j] + chr((ord(tempstr[j]) + x)) + tempstr[j + 1:]
                    for k in range(self.tsize):
                        fitness = fitness + abs(ord(tempstr[k]) - ord(self.args.GA_TARGET[k]))

                    if(fitness<bestfit):
                        bestfit=fitness
                        pop.str=tempstr
                        pop.fitness=bestfit
                        if steepest:
                            break

            if (tempstr == self.args.GA_TARGET and pop.fitness > 1):
                pop.fitness = pop.fitness - 1
                break
        if (counter >= subpop):
            break
```

And we also added the k-gene exchange algorithm :

```
def kgene(self,population: list[Struct],buffer,k):

    for i in range(self.esize, self.args.GA_POPSIZE):
        start = 0
        str=""
        for j in range(k):
            if j==k:
                i1 = randint(0, (self.args.GA_POPSIZE / 2) - 1)
                str = population[i].str[start:self.tsize]
                start += k
            i1 = randint(0, (self.args.GA_POPSIZE / 2) - 1)
            str=population[i].str[start:(start+self.tsize/k)]
            start+=k

        buffer[i]=Struct(str,0)
        if random() < self.args.GA_MUTATION:
            self.mutate(buffer[i])
```

We send to the k-gene function number k, which is the number of strings that we would like to use for the crossover

Comparison

And finally we compare the results with the first Lab assignment and we can clearly see that there is improvement:

Completeness: there isn't much change in this part because we managed to reach our goal in both assignments

Optimality: we can see in all 3 problems that we don't have the most optimal solution yet but there is improvement from the first assignment

Convergence: in the first assignment we can see that all 3 problems converted slowly and badly, but as the results show now there is a lot of improvements

Time Complexity: since we added life time learning for each generation obviously the each generation takes more time but we use much less generations now
As we saw for example we only used 20 generations for the Baldwin experience and we managed to reach a correct position percentage that is over 95%