AI lab 2 – part 1

Zinat Abo hgool 206721714

Elie Haddad 207931536

Q-1:

1)Genetic algorithm (Bull's eye):

In this heuristic we used string so in order to measure the distance between two strings, we used the edit distance that we learned in the lecture, here is the implementation:

```
def edit_dis(str1,str2):
    if(len(str1)> len(str2)):
        diff = len(str1) - len(str2)
        str1[:diff]
    elif(len(str2)> len(str1)):
        diff = len(str2) - len(str1)
        str2[:diff]
    else:
        diff = 0
    for i in range(len(str1)):
        if( str1[i] != str2[i]):
            diff += 1
    return diff
```

2)N-queens:

In this problem we deal with permutations, so in order to measure the distance between two permutations we used Kendall Tau , the code:

```
def kendallTau(A, B):
    pairs = itertools.combinations(range(0, len(A)), 2)

    distance = 0

    for x, y in pairs:
        a = A[x] - A[y]
        b = B[x] - B[y]

        # if discordant (different signs)
        if (a * b < 0):
            distance += 1

    return distance
```

Q-2:

1)Selection Pressure:

In the lecture we saw that selection pressure equals to the ratio between probability to select the fit member, we calculated this value previously in the implementation of RWS (prop) so we just send it as an input parameter, and the probability to select average member, so we just calculated the average of all the finesses values , and picked the genome that there fitness is very close to the average (no more than 2.5). here is the code:

```
def selection_pressure(self, population, prop):
    avg = self.calc_avg_fitness(population)
    count_avg = 0
    for pop in population:
        if pop.fitness - avg < abs(2.5):
            count_avg += 1
    avg_members = count_avg / self.args.GA_POPSIZE
    print("selection pressure is:",prop/avg_members)
```

2)genetic diversity:

In this section we used two method one (diversity) to calculate how each genome different from the rest of the population, then we used the (genetic diversity) method that assign the value to the score attribute for each genome(as we saw in the lecture) we assign k=0.5 in order to get moderate solution.

Here is the code:

```
def diversity(gene, args1, population):
    counter_div = 0
    for i in range(0,args1.GA_POPSIZE):
        if(population[i] != gene):
            counter_div += edit_dis(gene.str, population[i].str)
    return counter_div

def genetic_diversity(population, args1):
    k = 0.5
    div_counter = 0
    for gene in population:
        gene.score = gene.fitness + k * diversity(gene, args1, population)
        div_counter += gene.score
    avg = div_counter / args1.GA_POPSIZE
    print("the genes avarge diversity:", avg)
```

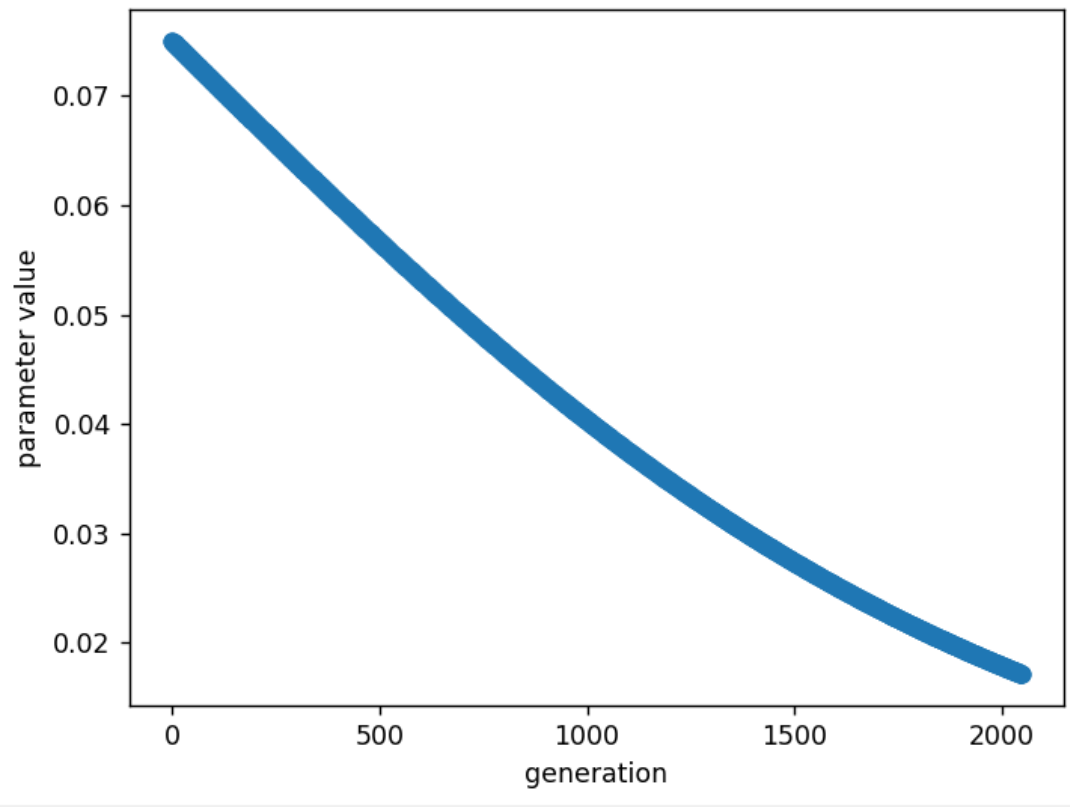
Q-3:

For static mutation rate: as we saw in the previous lab we assign the mutation rate to be all the time = 0.5, this doesn't encouraged a lot of diversity.

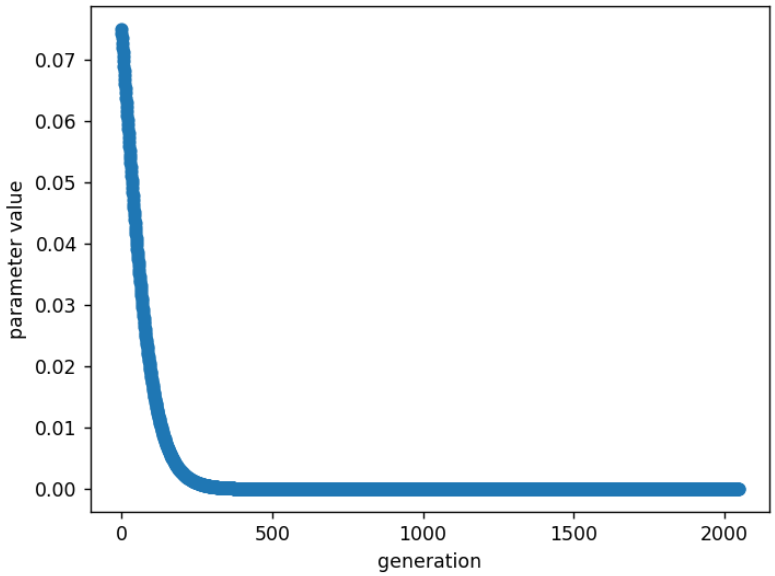
For Adaptive mutation(logistic decay): we used the non linear logistic decay that we saw in the lecture, we assign values to the parameters according to that we want a graph that is decreasing , here is the code:

```python
def logistic_decay(popsize):
    p_max = 0.075
    min = 0.025
    r = -0.001
    x = [0] * 2048
    y = [0] * 2048
    for i in range(0, popsize):
        mutate_rate = ((2 * (p_max ** 2) * e ** (r * i)) / (p_max + p_max * e ** (r * i)))
        x[i] = i
        y[i] = mutate_rate
    plt.scatter(x, y)
    plt.rcParams.update({'figure.figsize': (10, 8), 'figure.dpi': 100})
    plt.title('The distribution of the genes fitnesses')
    plt.xlabel('Fitness')
    plt.ylabel('Num of genes')
    plt.show()
    return y
```

Here is the graph when we assign r = - 0.001



Here when we assigned r = -0.02



We found out that using adaptive mutation we have more diversity in the generation, here is the code when we updated the mate function, this implementation apply also the triggered hyper mutation, such that when the parameter value drops down the min limit we fix the probability of the mutation :

Snip of the mate method

```
            buffer[i] = Struct(population[i1].str[0: spos1] + population[i2].str[spos1:s
        if(self.args.CROSS_TYPE == "Uniform"):
            gene = ""
            for j in range(self.tsize):
                x =randint(0, sys.maxsize) % 2
                if x == 0:
                    gene = gene + population[i1].str[j]
                else:
                    gene = gene + population[i2].str[j]
            buffer[i] = Struct(gene, 0)
        # adding the trigered hyper mutation condition if it drops down the min limit
        min = 0.25
        if y[i] < min:
            y[i] = min
        if random() < y[i] * random():
            self.mutate(buffer[i])
```

Q-4:

Threshold Speciation:

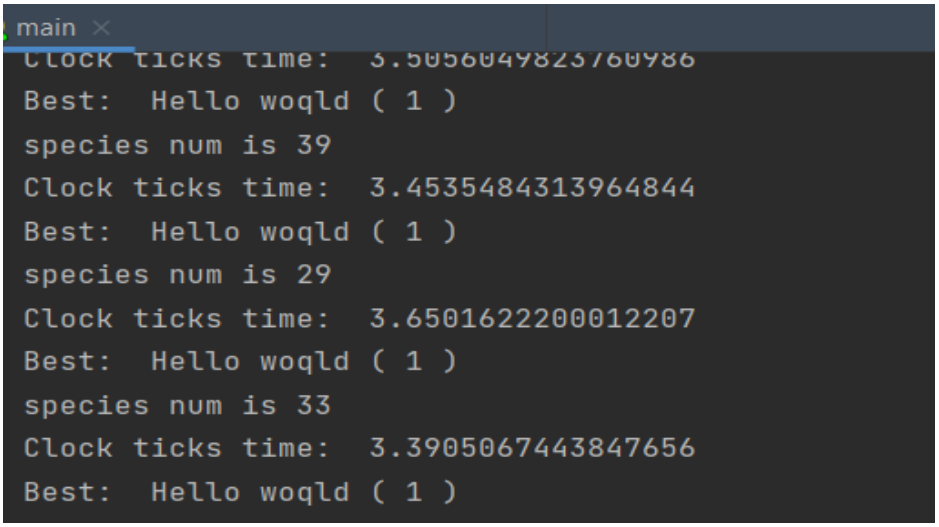
in order to implement this algorithm: we first add an attribute (species) for each genome in the population(initialized with -1) so that each genome is mapped to specific specie. Then, initialize the first genome in the population to be in the first specie, after this we iterate over all the genome in the population and checked if it apply the condition of the species threshold for every genome that are already mapped to this specie (that we saw in the lecture) if yes, it can join this specie, if no it creates a new specie.

Here is a bit of the code:

```
for i in range(1, args1.GA_POPSIZE):
    flag = 1
    for spec in all_species:
        if edit_dis(spec[0].str, population[i].str) < spec_threshold :# if the new genom uphold the threshold condition with the first genom in the c
            for k in range(1, len(spec)):#check the other genoms in this species
                if edit_dis(spec[k].str, population[i].str) >= spec_threshold:# if the condition doesn't hold for specific genom we create new specie
                    flag = 0
                    break
            if flag == 1:
                spec.append(population[i])
                population[i].species = spec
                break

    if(population[i].species == -1):
        curr_specie += 1
        population[i].species = curr_specie
        specie1 = []
        specie1.append(population[i])
        all_species.append(specie1)
```

This algorithm took a lot of time because it has many loop that checks every genome in the population, we assigned different values to the species threshold, when it is very high we had few species, and vice versa. We finally found that the optimal value of species threshold = 3, in order to try and reach species count = 30



K-means:

Our algorithm uses many methods, we first pick K random centroids, then we map every genome in the population to the centroid that is the closest to it. After doing this we update the centroids to be the average of the distance between the old centroid and all the genomes in the specific cluster.

Here is a bit of the code:

```
def K_means(population, args1, k):
    centroids = getRandomCent(population,k)
    old_centroids = None
    while(centroids != old_centroids):
        old_centroids = centroids
        assign_members(population, centroids, args1)
        for i in range(args1.GA_POPSIZE):
            centroids = update_centroids(population, k, old_centroids,args1)
    return centroids
```

To find the k optimal we implement the Silhouette Coefficient

And it gives us the result k=3

Here is a bit of the code:

```
def calc_silhouette(population, args1):
    sil_avg = []
    labels = []
    range_n_cluster = [2,3,4,5,6,7,8]
    for k in range_n_cluster:
        kmeans = K_means(population,args1,k)
        for i in range(args1.GA_POPSIZE):
            labels.append(population[i].centroid)
        sil_avg.append(silhouette_score(population, labels, metric='euclidean'))
    plt.plot(range_n_cluster, sil_avg, 'bx-')
    plt.xlabel('Values of K')
    plt.ylabel('Silhouette score')
    plt.title('Silhouette analysis For Optimal k')
    plt.show()
```

When we apply k-means with k=3 we get the results in less time compared with threshold speciation.

Q-5:

Random immigrants:

This algorithm replaces the worst genomes by randomly generated genomes.

This is our code:

```
def immigrants(self, buffer):
    index = 0
    for i in range(self.args.GA_POPSIZE- self.esize, self.args.GA_POPSIZE):
        buffer[i] = buffer[index]
        index += 1
        if randrange(sys.maxsize) < sys.maxsize:
            self.mutate(buffer[i])
```

Q-6:

Three problems: (Bull's eye/ N queens/ Bin packing)

Completeness: for the three problems in lab 1 and in this lab we got an answer they all were complete.

Optimality: in lab 1 we didn't really got the optimal of bull's eye and N-queens even it was the worse in bin packing, but in this lab we got slightly improved result in the optimally of the three problems compared to the previous lab.

Convergence: for the three problems in lab 1 we got slow convergence compared to what we now have got.

Time complexity: also here for the three problems in lab 1 we got slow time complexity compared to what we now have got.