Lab4 – part 1

Zinat Abo Hgool 206721714

Elie Hadad 207931536

Introduction:

In this assignment we were asked to reprocess the Hillis experiment, in this experiment Hillis tried to minimize the number of the comparators in a sorting network.

So based on Hillis co-evolutionary algorithm, in our implementation the "hosts" represented by configurations of the sorting network, and "parasites" represented test data providing it to a host as an input.

Initializing :

First, we based our algorithm on the concept of genetic algorithm that we saw in the previous labs. In this assignment we consider two sizes of input one is k=6 and k=16.

The parasites population consist of new genes, every one of them has an initial fitness and an array of length k of random values.

Next, we initialize the host (we called it in our implementation "population"), we defined for every input size (6 or 16) two variables, minlength and maxlength according to the table below that shows the minimal possible number of comparators of input with size n.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Upper bound	0	1	3	5	9	12	16	19	25	29	35	39	45	51	56	60

using loop we generated new gene every one of them has initial array of size 2*k (k is random number between minlength and maxlength) this array is basically the sorting network **every two adjacent elements of this array are comparator.**

Below the full code:

```
def init_population(self): #create popsize citizens
      for i in range(100):
          gene = Struct(np.random.randint(0,self.nwsize,self.nwsize), 0)
          self.parasites.append(gene)
      for j in range(self.args.GA_POPSIZE):
          nw = []
          for k in range(randrange(self.minlength, self.maxlength)):
              y = randrange(0, self.nwsize)
              nw.append(y)
              x=randrange(0,self.nwsize)
              while(x==y):
                  x = randrange(0, self.nwsize)
              nw.append(x)
          gene = Struct(nw, 0)
          self.population.append(gene)
```

Calculating the fitness:

the concept of fitness in our implementation consists of two elements:

*Parasite fitness: for every test in the parasite, we applied the sorting networks that we generated previously on this test, using a function that we called " sortByNetwork" this function takes a network and test and checks if this network returned the sorted version of the test array (we will explain more about this function later). For every sorting network that doesn't succeed in sorting the array we give its fitness penalty of 1.

*Host fitness (we called it population):

This calculation has the same concept of the previous one, but here we run the sorting network over every test array in the parasite, using the function " sortByNetwork" and if the sorting network wasn't successful in sorting the test array, we increased the fitness of the network by 1. Another important factor that we took in calculating the fitness of the network is the depth, in order to reach optimal sorting network, we need

to calculate the depth of the desired sorting network and penalize the fitness of the network if its depth exceed the desire one.

Below is the code of what we explained above:

```
def calc_fitness(self, population: list[Struct]):

    for test in self.parasites:
        fitness = 0
        for network in self.population:
          if self.sortByNetwork(network.arr,test.arr):
              fitness+=1
        test.fitness=fitness/self.args.GA_POPSIZE

    for network in self.population:
        fitness = 0
        depth=len(network.arr)
        for test in self.parasites:
            if self.sortByNetwork(network.arr,test.arr):
                fitness+=1
        network.fitness=(fitness/100)*depth
```

sortByNetwork function:

this is helper function that we constructed in order to check if given a sorting network can produce a sorting version of the given test array, for every comparator in the sorting network we checked the indices in the test array and swap between them if needed (namely the first one should be the smaller one).

Below the code:

```
def sortByNetwork(self,network,test):
    test2=[]
    for i in test:
        test2.append(i)
    for i in range(0,len(network)-1,2):
        if test2[network[i]]>test2[network[i+1]]:
            temp = test2[network[i + 1]]
            test2[network[i + 1]] = test2[network[i]]
            test2[network[i]] = temp
    for i in range(len(test2)-1):
        if(test2[i]>test2[i+1]):
            return True

    return False
```

Mate:

The concept of mate function is based on the one we saw in the previous labs. First, we call elitism function with elite rate = 0.1

Then, we preformed a uniform crossover, the same implementation like in lab 1. Note: in case the two genes don't have the same size, we took the smallest sized one.

Below the code:

```
def elitism(self, population: list[Struct], buffer: list[Struct]):
    flag = 0
    temp = []
    while len(temp) < self.esize:
            temp.append(population[flag])
            flag += 1
    # temp = population[:self.esize].copy()
    self.population[:self.esize] = temp
    for i in range(self.esize):
       self.population[i].age += 1
```

```
def mate(self):
    self.elitism(self.population, self.buffer)
    for i in range(self.esize, self.args.GA_POPSIZE):

        i1 = randint(0, (self.args.GA_POPSIZE/2) - 1)
        i2 = randint(0, (self.args.GA_POPSIZE/2) - 1)

        gene = []
        for j in range(min(len(self.population[i1].arr),len(self.population[i2].arr))):
            x =randint(0, sys.maxsize) % 2
            if x == 0:
                gene.append(self.population[i1].arr[j])
            else:
                gene.append(self.population[i2].arr[j])
        self.population[i] = Struct(gene, 0)
        if random.random() < self.args.GA_MUTATION:
            self.mutate(self.population[i])
```

Mutate:

With mutation rate = 0.5, we choose a random comparator and delete its values then we choose random position and inserted new comparator.

Below the code:

```
def mutate(self, member: Struct):
    ipos = randint(0,len( member.arr)-1)
    del member.arr[ipos]
    ipos = randint(0, len(member.arr) - 1)
    if ipos%2==1:
        ipos-=1
    y=randrange(0,self.nwsize)
    x = randrange(0, self.nwsize)
    while x==y:
        x = randrange(0, self.nwsize)

    member.arr.insert(ipos,x)
    member.arr.insert(ipos+1, y)
```

Print sorted function:

This is a helper function we constructed in order to print the arrays in the parasite sorted according to the given sorting network.

Below the code:

```
def print_sorted(self,network):
    for tests in self.parasites:
        test=tests.arr
        test2 = []
        for i in test:
            test2.append(i)
        for i in range(0, len(network) - 1, 2):
            if test2[network[i]] > test2[network[i + 1]]:
                temp = test2[network[i + 1]]
                test2[network[i + 1]] = test2[network[i]]
                test2[network[i]] = temp
        print("sorted array: ",test2)
```

Main.py:

Get input:

This function initializes all the arguments we needed to implement our algorithms.

run:

looping for max iteration, first we calculated the fitness and sort the population by the fitness. As long as there is no fitness equals to 0 we call the mate function.

This is the code:

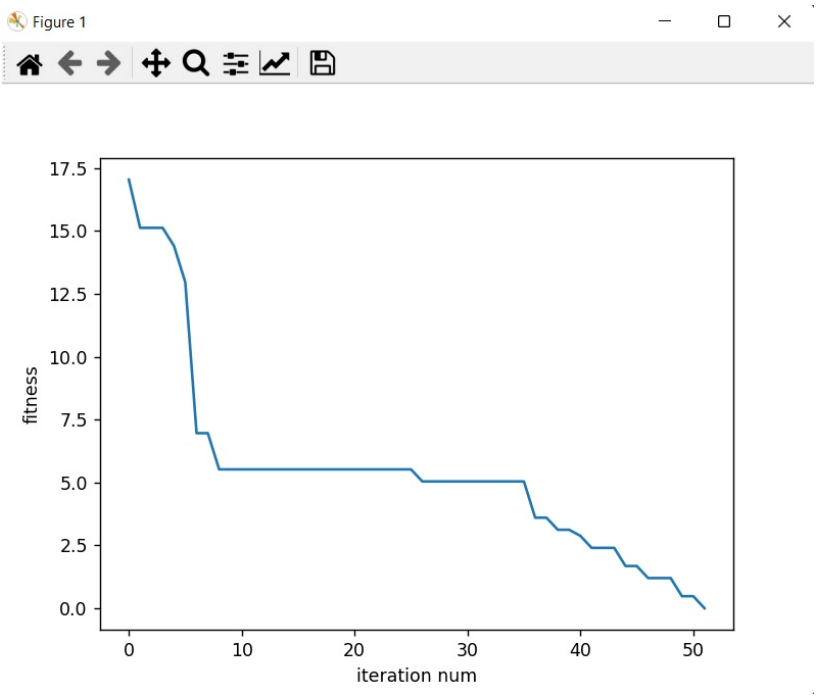
```
def GArun(cross_type):
    start_qt = time.time()  # the start time of clock ticks
    start_elapsed = timer()  # the start time of elapsed
    args1 = get_input()
    ga = Genetic5(args1)
    esize = int(args1.GA_POPSIZE * args1.GA_ELITRATE)
    ga.init_population()
    run(ga,args1, esize, start_elapsed)
def get_input():
    popsize = 600
    maxIter = 200

    args = Var(maxIter,popsize,0.1,0.5)
    ARGS=args
    return args

if __name__ == '__main__':
    GArun(ARGS)
```

```
    for i in range(args1.maxiter):
        iteration_time = time.time()
        ga.calc_fitness(ga.population)
        ga.sort_by_fitness(ga.population)
        print("fitness",ga.population[0].fitness)
        ga.print_best(ga.population)
        lst.append(ga.population[0].fitness)
        popfit=ga.population
        if ga.population[0].fitness == 0:
            print()
            print("Best string: ", ga.population[0].arr)
            ga.print_sorted(ga.population[0].arr)
            print(f'found in {i} iterations out of {args1.maxiter}')
            elapsed_time = timer() - start_elapsed
            print("Elapsed time: ", elapsed_time)
            fig, ax = plt.subplots()
            gennumarr = []
            for i in range(len(lst)):
                gennumarr.append(i)
            ax.plot(gennumarr, lst)
            print(gennumarr)
            print(lst)
            plt.xlabel('iteration num')
            plt.ylabel('fitness')
            plt.show()
            break
        ga.mate()
        clock_ticks = time.time() - iteration_time
        print("Clock ticks time: ", clock_ticks)
```

Results:

After implementing our algorithms, we run it multiply time to check the output we got, and we concluded the following:

N=6



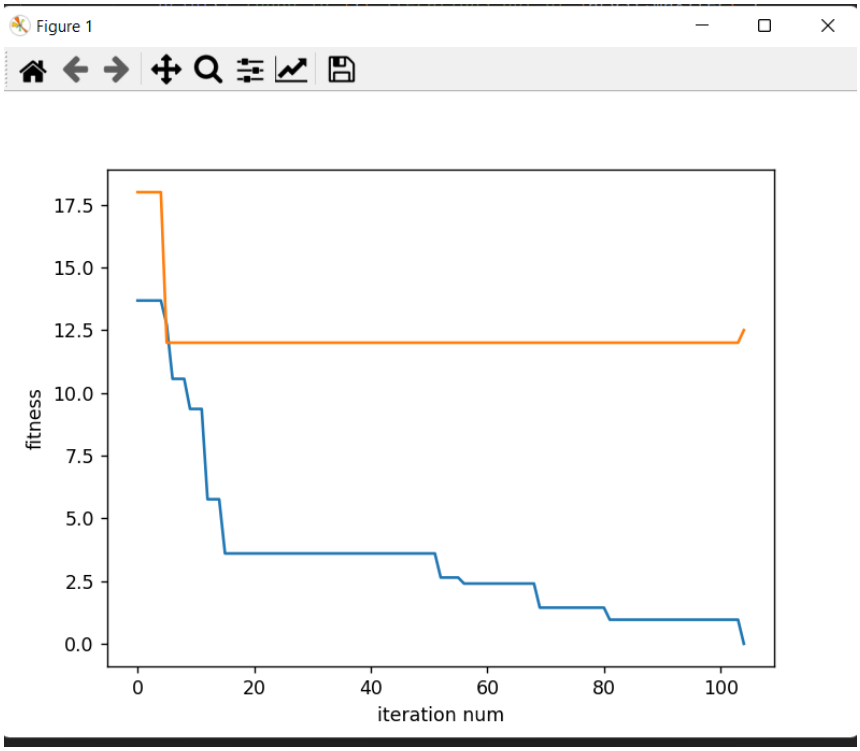
This is the graph of fitness in case when n=6, we can see the improvement in fitness along with iteration number.

We printed the best network and the sorted arrays of the parasites as the following:

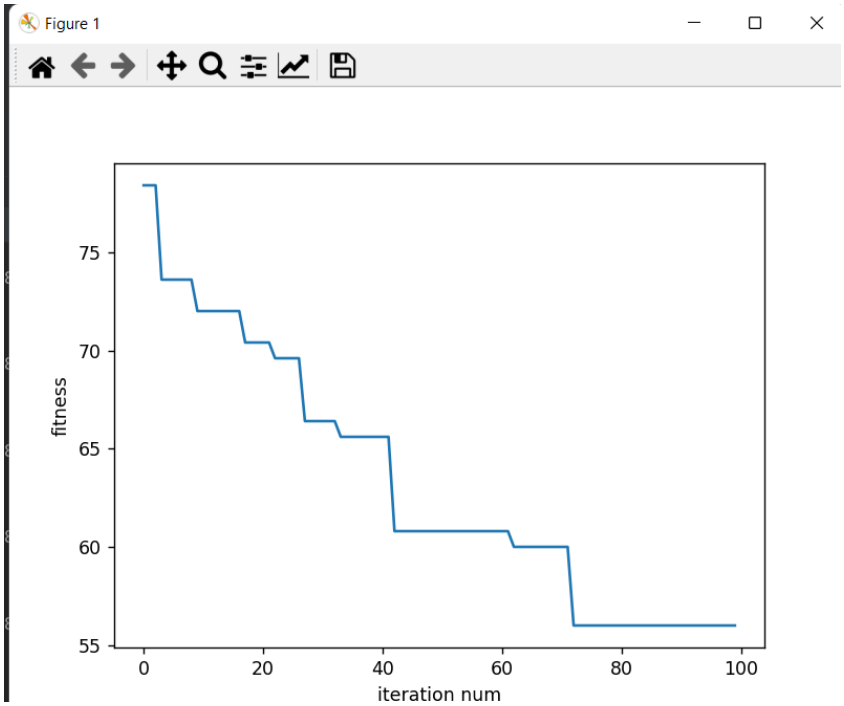
```
Clock ticks time:  0.556736946105957
fitness 1.6800000000000002
Best:  [1, 3, 0, 4, 0, 5, 0, 1, 3, 5, 2, 4, 1, 2, 1, 3, 2, 2, 4, 5, 2, 3, 3, 4] ( 1.6800000000000002 )
Clock ticks time:  0.5422646999359131
fitness 1.6800000000000002
Best:  [1, 3, 0, 4, 0, 5, 0, 1, 3, 5, 2, 4, 1, 2, 1, 3, 2, 2, 4, 5, 2, 3, 3, 4] ( 1.6800000000000002 )
Clock ticks time:  0.543872594833374
fitness 1.2000000000000002
Best:  [3, 2, 1, 5, 0, 4, 0, 1, 2, 5, 0, 3, 3, 4, 1, 3, 1, 2, 4, 5, 2, 3, 3, 4] ( 1.2000000000000002 )
Clock ticks time:  0.5482540130615234
fitness 1.2000000000000002
Best:  [3, 2, 1, 5, 0, 4, 0, 1, 2, 5, 0, 3, 3, 4, 1, 3, 1, 2, 4, 5, 2, 3, 3, 4] ( 1.2000000000000002 )
Clock ticks time:  0.537628173828125
fitness 1.2000000000000002
Best:  [3, 2, 1, 5, 0, 4, 0, 1, 2, 5, 0, 3, 3, 4, 1, 3, 1, 2, 4, 5, 2, 3, 3, 4] ( 1.2000000000000002 )
Clock ticks time:  0.5477437973022461
fitness 0.48
Best:  [0, 4, 0, 2, 3, 5, 0, 1, 2, 5, 0, 3, 1, 4, 1, 3, 1, 2, 4, 5, 2, 3, 3, 4] ( 0.48 )
Clock ticks time:  0.5519585609436035
fitness 0.48
Best:  [0, 4, 0, 2, 3, 5, 0, 1, 2, 5, 0, 3, 1, 4, 1, 3, 1, 2, 4, 5, 2, 3, 3, 4] ( 0.48 )
Clock ticks time:  0.7592735290527344
fitness 0.0
Best:  [3, 2, 3, 4, 0, 5, 0, 1, 2, 5, 0, 3, 1, 4, 1, 3, 1, 2, 4, 5, 2, 3, 3, 4] ( 0.0 )

Best string:  [3, 2, 3, 4, 0, 5, 0, 1, 2, 5, 0, 3, 1, 4, 1, 3, 1, 2, 4, 5, 2, 3, 3, 4]
sorted array:  [0, 1, 1, 3, 3, 5]
sorted array:  [0, 0, 0, 3, 4, 4]
sorted array:  [1, 1, 2, 2, 4, 5]
```

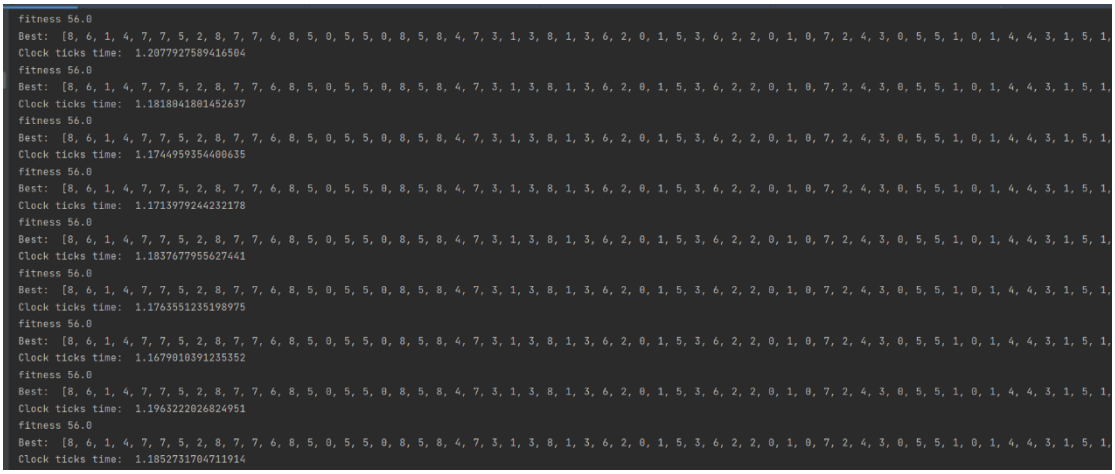
```
sorted array:  [1, 2, 3, 3, 4, 4]
sorted array:  [0, 0, 0, 0, 1, 3]
sorted array:  [0, 2, 2, 3, 4, 5]
sorted array:  [0, 1, 2, 3, 5, 5]
sorted array:  [0, 1, 2, 2, 3, 4]
sorted array:  [0, 0, 1, 1, 1, 3]
sorted array:  [0, 0, 4, 4, 5, 5]
sorted array:  [0, 2, 2, 2, 3, 4]
sorted array:  [0, 1, 1, 2, 3, 5]
sorted array:  [0, 0, 1, 1, 2, 5]
sorted array:  [1, 2, 2, 3, 4, 4]
sorted array:  [0, 1, 2, 3, 4, 4]
sorted array:  [0, 0, 1, 1, 3, 5]
sorted array:  [0, 0, 2, 2, 2, 2]
sorted array:  [1, 1, 1, 3, 3, 5]
sorted array:  [0, 0, 3, 3, 5, 5]
sorted array:  [0, 1, 1, 2, 3, 5]
sorted array:  [0, 0, 1, 1, 5, 5]
sorted array:  [2, 2, 2, 3, 3, 4]
sorted array:  [0, 1, 3, 4, 5, 5]
sorted array:  [0, 1, 2, 3, 4, 4]
sorted array:  [1, 1, 2, 3, 3, 5]
sorted array:  [2, 2, 4, 4, 4, 5]
sorted array:  [0, 1, 1, 2, 4, 4]
sorted array:  [1, 1, 3, 4, 4, 4]
sorted array:  [1, 2, 2, 2, 4, 5]
sorted array:  [0, 0, 3, 3, 5, 5]
found in 51 iterations out of 300
Elapsed time:  33.156448
```

This is a graph with the best fitness in blue and the depth in orange



N=16





Note:our algorithm takes a lot of time when running on n>10

So we had stop it before it reached fitness=0

But we saw that there was improvement in the fitness in each generation