מעבדה 2 חלק ב

סאמר חרעובה – 209050202

עבד אלרחמן אבו חוסין – 208517631

Section 1:

- Firstly we created a problem set for Baldwins effect that is based on our classic class DNA:
    1. Selects 1,0,? As described in the Baldwin experiment 0.25,0.25,0.5 respectively

```
class baldwin_effect(DNA):
    # our object is the initial position , we added 2 parameters that are required
    def __init__(self):
        DNA.__init__(self)

    def create_object(self, target_size, target):
        numTrue = math.floor(0.25 * target_size)
        numQmark = target_size - 2 * numTrue
        places_to_select = [i for i in range(target_size)]
        self.object = [None] * target_size
        Qmarkplaces = random.sample(places_to_select, numQmark)
        places_to_select = list(numpy.setxor1d(numpy.array(places_to_select), numpy.array(Qmarkplaces)))
        true_places = random.sample(places_to_select, numTrue)

        self.object = ['?' if i in Qmarkplaces else '1' if i in true_places else '0' for i in range(target_size)]

    def character_creation(self, target_size=0):
        return chr(random.randint(0,1))
```

- Then we added 2 fitness functions :
    a. fixed_distance : calculated number of fixed correct/incorrect placements of bits
    b. baldwinss: fitness function given in the lecture

```
def fixed_distance(self, object, target, target_size=0):
    correct = incorrect = 0

    for i in range(len(target)):
        if object[i] == target[i]:
            correct += 1
        elif object[i]!='?':
            incorrect += 1
    return correct, incorrect

def baldwinss(self, pop_size, tries, num_tries):
    return 1 + ((pop_size - 1) * tries / num_tries)
```

- created a memetic algorithm called PureMA:

```python
class PureMA(GA_LAB1):
    def __init__(self, target, tar_size, pop_size, problem_spec, crosstype, fitnesstype, selection,
                 serviving_mechanizem, mutation, gene_dist, mutation_probability=0):
        GA_LAB1.__init__(self, target, tar_size, pop_size, problem_spec, crosstype, fitnesstype, selection,
                 serviving_mechanizem, mutation, gene_dist,0)
        self.correct_fixed=[]
        self.incorrect_fixed=[]
        self.generations=[]
        self.learning_bits=[]

    def algo(self, i):...

    def stopage(self, i):...

    def evolve(self,i):...

    def baldwin(self,individual):
        for index,i in enumerate(individual.object):
            individual.object[index]=individual.character_creation() if i=='q' else i
    def correctness(self):...
    def local_search(self,num_tries):...
```

- baldwin(self,individual): replaces every '?' with either 1 or 0
- correctness: function that calculates percentages of fixed correct/incorrect bits
- local_search: the described local search given in the lecture

```python
def correctness(self):
    correctly_fixed = numpy.array([None] * self.pop_size)
    incorrectly_fixed = numpy.array([None] * self.pop_size)
    learnt_bits = numpy.array([None] * self.pop_size)
    for index, pop in enumerate(self.population):
        correct, incorrect = pop.fitnesstype['fixed'](pop.object, self.target)
        learnt_bits_num = self.target_size - correct - incorrect
        correctly_fixed[index], incorrectly_fixed[index] = correct, incorrect
        learnt_bits[index] = learnt_bits_num
    return correctly_fixed.mean() / self.target_size, incorrectly_fixed.mean() / self.target_size, learnt_bits.mean() / self.target_size
```

```python
def local_search(self, num_tries):
    tries = 0
    for index, pop in enumerate(self.population):
        for n in range(1, num_tries + 1):
            temp = self.prob_spec()
            # copy string
            temp.object = [i for i in pop.object]
            self.baldwin(temp)
            # check distance from target
            temp.calculate_fittness(self.target, self.target_size, 0, self.selection)
            # if distance is zero then calculate n (here we call it tries as in tries left) and send it to fitness function given in lecture
            if not temp.fitness:
                self.population[index] = temp
                del pop
                tries = num_tries - n
                break
        # fitness given in lecture : 1+(19+....)  explained in "baldwin" in fitness class
        self.population[index].fitnesstype['baldwin'](self.pop_size, tries, num_tries)
```

In local search , every individual is searched up to 1000 times , as

Section 1.a:

In terminal demo:

```
correctly fixed: 0.3046  incorrectly fixed: 0.2496  learning_bits: 0.44580000000000003
 Best:?,?,1,1,1,0,1,1,0,1,0,1,0,0,1,?,?,0,0,?, ,fittness: 780  Mean: 0 ,Variance: 780.0 Time :  19.1007318  ticks: 19.100881338119507
```

- In code we used the correctness function mentioned and explained above

A live demo of the code :

```
set population size:1000
chose algorithem :  1:GA
2:PSO 3:Minimal conflicts
 4:first fit
 5: baldwins
 6:MA+GA5
choose surviving strategy :  Elite: 1 ,Age: 21
if you want to use cx make sure that the string doesn't have 2 matching letters !
type bit string: 11101011110001010101
choose cross function :  One Cross: 1  Two Cross: 2  Uniform: 3  PMX: 4   CX: 53
choose selection function :  RAND: 0  SUS: 1  RWS: 2  tournement:32
choose fitness function :  0:Distance  1:Bul Pgia   1
choose mutation scheme:  random mutation: 1 ,swap_mutate: 2 ,insertion_mutate: 32
correctly fixed: 0.3046  incorrectly fixed: 0.2496  learning_bits: 0.44580000000000003
 Best:?,?,1,1,1,0,1,1,0,1,0,1,0,0,1,?,?,0,0,?, ,fittness: 780  Mean: 0 ,Variance: 780.0 Time :  19.1007318  ticks: 19.100881338119507
correctly fixed: 0.37315  incorrectly fixed: 0.2663  learning_bits: 0.36055000000000004
 Best:?,1,0,0,0,0,1,1,1,?,0,1,0,0,0,1,?,1,1,0, ,fittness: 540  Mean: 0 ,Variance: 540.0 Time :  37.4028209  ticks: 37.40307664871216
correctly fixed: 0.43955  incorrectly fixed: 0.28769999999999996  learning_bits: 0.27275
 Best:1,1,1,?,1,0,1,1,1,1,0,0,0,1,1,1,?,1,0,1, ,fittness: 270  Mean: 0 ,Variance: 270.0 Time :  53.750146599999994  ticks: 53.75067663192749
correctly fixed: 0.50815  incorrectly fixed: 0.30505  learning_bits: 0.18680000000000002
 Best:1,1,1,0,1,0,1,1,1,1,?,0,1,1,1,1,1,1,0,1, ,fittness: 210  Mean: 0 ,Variance: 210.0 Time :  68.7085964  ticks: 68.7093415260315
correctly fixed: 0.57725  incorrectly fixed: 0.30924999999999997  learning_bits: 0.1135
 Best:1,1,1,1,1,0,1,0,1,1,1,0,0,0,0,1,0,1,0,1, ,fittness: 120  Mean: 0 ,Variance: 120.0 Time :  81.6918596  ticks: 81.69241499900818
correctly fixed: 0.6362  incorrectly fixed: 0.3069  learning_bits: 0.05689999999999999
 Best:0,1,1,0,1,0,1,1,1,1,0,0,1,1,0,1,0,0,0,1, ,fittness: 90  Mean: 0 ,Variance: 90.0 Time :  92.8193369  ticks: 92.82009077072144
correctly fixed: 0.70025  incorrectly fixed: 0.28095  learning_bits: 0.0188
 Best:1,1,1,0,1,0,1,1,1,1,0,0,0,1,0,1,0,1,0,1, ,fittness: 0  Mean: 0 ,Variance: 0.0 Time :  102.6636808  ticks: 102.66389751434326
correctly fixed: 0.7602  incorrectly fixed: 0.2378  learning_bits: 0.002
 Best:1,1,1,0,1,0,1,1,1,1,0,0,0,1,0,1,0,1,0,1, ,fittness: 0  Mean: 0 ,Variance: 0.0 Time :  111.0569465  ticks: 111.05719780921936
correctly fixed: 0.8268000000000001  incorrectly fixed: 0.17304999999999998  learning_bits: 0.00015000000000000001
 Best:1,1,1,0,1,0,1,1,1,1,0,0,0,1,0,1,0,1,0,1, ,fittness: 0  Mean: 0 ,Variance: 0.0 Time :  119.54254639999999  ticks: 119.54275870323181
correctly fixed: 0.8768499999999999  incorrectly fixed: 0.12315000000000001  learning_bits: 0.0
 Best:1,1,1,0,1,0,1,1,1,1,0,0,0,1,0,1,0,1,0,1, ,fittness: 0  Mean: 0 ,Variance: 0.0 Time :  127.6230936  ticks: 127.62356424331665
correctly fixed: 0.9157  incorrectly fixed: 0.0843  learning_bits: 0.0
 Best:1,1,1,0,1,0,1,1,1,1,0,0,0,1,0,1,0,1,0,1, ,fittness: 0  Mean: 0 ,Variance: 0.0 Time :  135.6059852  ticks: 135.60673594474792
correctly fixed: 0.9416  incorrectly fixed: 0.058399999999999994  learning_bits: 0.0
 Best:1,1,1,0,1,0,1,1,1,1,0,0,0,1,0,1,0,1,0,1, ,fittness: 0  Mean: 0 ,Variance: 0.0 Time :  142.3185529  ticks: 142.31911754608154
correctly fixed: 0.9544499999999999  incorrectly fixed: 0.04555  learning_bits: 0.0
 Best:1,1,1,0,1,0,1,1,1,1,0,0,0,1,0,1,0,1,0,1, ,fittness: 0  Mean: 0 ,Variance: 0.0 Time :  148.0759771  ticks: 148.07672238349915
correctly fixed: 0.9633499999999999  incorrectly fixed: 0.03665  learning_bits: 0.0
 Best:1,1,1,0,1,0,1,1,1,1,0,0,0,1,0,1,0,1,0,1, ,fittness: 0  Mean: 0 ,Variance: 0.0 Time :  152.4843422  ticks: 152.4847469329834
[0.44580000000000003, 0.36055000000000004, 0.27275, 0.18680000000000002, 0.1135, 0.05689999999999999, 0.0188, 0.002, 0.00015000000000000001, 0.0, 0.0, 0.0, 0.0, 0.0]
 number of generations :  13
Overall runtime : 237.6686723

 run again ? press y for yes n for no
```

In the above simulation we used :

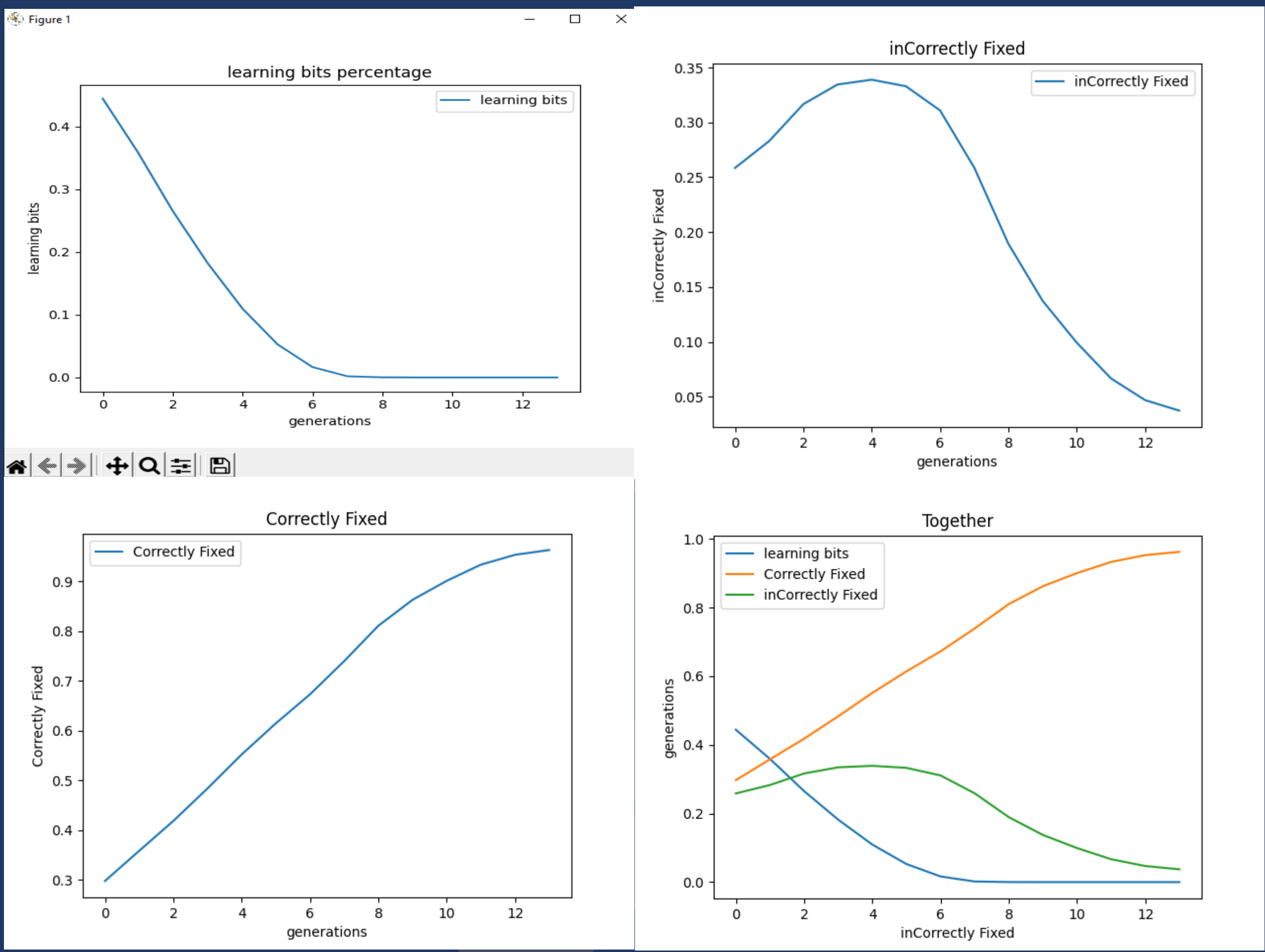
1. Swap mutation
2. RWS selection
3. 11101011110001010101 string made of 20 bits

As we can see , the results are amazing , in 6 generations the algorithm finds the best individuals , as we have made the algorithm work until fixed bits converge to 1 and about 0 .

,so the Baldwin effect actually does work wonders

Section 1.b:

The graphs of the above simulation :



Results of the graphs:

1. As we can see the correctly fixed bits increase with time up until they get to 100%
2. And the number of incorrectly fixed bits increases in a couple of generation , mainly due to the mutation type that we selected , and then rapidly decreases to zero
3. Learning bits decrease significantly , we can see that by the 6'th generation they almost came to be 0 and then we can also see that the algorithm has already found the solution by generation 6

Section 2 :

Added a generic implementation of MA with options for a learning algorithm and a fitness for learning :

```
class HybridMA(PureMA):
    def __init__(self, target, tar_size, pop_size, problem_spec, crosstype, fitnesstype, selection,
                 serviving_mechanizem, mutation, gene_dist, frequency, intensity,learning_algo,learning_fitness, mutation_probability=0):
        PureMA.__init__(self, target, tar_size, pop_size, problem_spec, crosstype, fitnesstype, selection,
                        serviving_mechanizem, mutation, gene_dist, 0)
        self.intensity=intensity
        self.frequency=frequency
        self.learning_algo=learning_algo
        self.learning_fitness=learning_fitness
        # selection parameter in previous classes is used for understanding which individuals will learn
    def search(self):...
    def algo(self, i):
        # self.calc_fitness()  # evaluate all individuals in population
        self.search()
        self.propablities_rank_based()
        self.evolve(i)  # evolve a new population
        self.population, self.buffer = self.buffer, self.population  # // swap buffers
        self.population = sorted(self.population)
        cf, icf, learning_bits = self.correctness()
        # for graphs
        self.correct_fixed.append(cf)
        self.incorrect_fixed.append(icf)
        self.learning_bits.append(learning_bits)
        self.generations.append(i)
        print("correctly fixed:", cf, " incorrectly fixed:", icf, " learning_bits:", learning_bits)
        self.solution = self.population[0]
```

Section 2.a:

Added intensity and frequency ,so that we can use them in the upcoming learning algorithms:

```
    def __init__(self, target, tar_size, pop_size, problem_spec, crosstype, fitnesstype, selection,
                 serviving_mechanizem, mutation, gene_dist, frequency, intensity,learning_algo,learning_fitness, mutation_probability=0):
        PureMA.__init__(self, target, tar_size, pop_size, problem_spec, crosstype, fitnesstype, selection,
                        serviving_mechanizem, mutation, gene_dist, 0)
        self.intensity=intensity
        self.frequency=frequency
        self.learning_algo=learning_algo
        self.learning_fitness=learning_fitness
        # selection parameter in previous classes is used for understanding which individuals will learn
```

Section 2.b:

i. -> learning_fitness
ii. -> algo_huristic

```
class Agent:
    fitnesstype = fitness_selector().select

    def __init__(self):
        self.object = None
        self.learning_fitness=0
        self.algo_huristic=None
        self.age = 0
        self.fitness = 0
```

Section 2.c:

1. Hill Climbing algorithm:
   a. We used a function that creates all neighbors of each individual
   b. Then sorted them according to the learning fitness and chose the best individual
   c. We iterate over them according to the number of tries left for each individual (intensity)

```
def hill_climbing(self, pop_size, hill_probability=None):
    for i,pop in enumerate(self.population):
        tries = 0
        best_neighbour=self.best_neighbour(pop)
        while best_neighbour.learning_fitness!=0 and self.intensity-tries>0:
            best_neighbour=self.best_neighbour(best_neighbour)
        self.population[i]=best_neighbour
def best_neighbour(self,citizen):
    neighbours=[]
    for i in range(len(citizen)):
        for j in range(i + 1, len(citizen)):
            neighbour = citizen.copy()
            neighbour.object[i] = citizen.object[j]
            neighbour.object[j] = citizen.object[i]
            neighbour.learning_fitness=neighbour.Learning_fitness(self.target, self.target_size, self.learning
            neighbours.append(neighbour)
    neighbours=sorted(neighbours,key=lambda x:x.learning_fitness)
    return neighbours[0]
```

2. Random walk:
    a. We get the probability of up or down movement from the user .
    b. We then create a pattern of up and down indexes (walk)
    c. Then we iterate over the selected individual's neighbors until we meet our stoppage point

```
def random_walk(self, pop_size, hill_prabability):
    prob = [hill_prabability, 1 - hill_prabability]
    pos = [2]
    random_array = numpy.random.random(self.target_size)
    # create arrays of random places
    down = random_array < prob[0]
    up = random_array >= prob[1]
    for i, j in zip(up, down):
        down2 = j and pos[-1] > 1
        up2 = i and pos[-1] < self.target_size
        pos.append(pos[-1] - down2 + up2)
    # pos is an index array of random walk (positions to walk to )
    for i,citizen in enumerate(self.population):
        tries=0
        best=self.neighbours_given_pos(pos,citizen)
        while best.learning_fitness != 0 and self.intensity - tries > 0:
            best = self.best_neighbour(best)
        self.population[i] = best

def neighbours_given_pos(self,pos,citizen):
    neighbours = []
    for i in pos:
        # get the member
        for j in pos:
            if j != i:
                neighbour = citizen.copy()
                neighbour.object[i] = citizen.object[j]
                neighbour.object[j] = citizen.object[i]
                neighbour.learning_fitness = neighbour.Learning_fitness(self.target, self.target_size,
                                                                        self.learning_fitness)
                neighbours.append(neighbour)
    neighbours = sorted(neighbours)
    return neighbours[0]
```

Section 2.d:

We used the a heuristic that we used in section 1 as it best suits this problem

Section 2.e :

Implementing k-gene-exchanges was easy as we used the base engine of section 1 which allowed us to use previous functions with a small change , that being using k instead of another parameter .

```
def evolve(self, i):
    GA_LAB1.mate(self, i) if not self.k else self.k_gene_exchange(i,self.k)

def k_gene_exchange(self, gen,k):
    esize = self.serviving_genes(gen)
    # cross function for intial GA algo
    self.cross(esize, gen, self.population, k)
```

Section 2.f :

- **Completeness:** this algorithm always got to the solution just like most of our algorithms ,given the right parameters it is complete
- **Optimality :** it always get’s the solution and not a close approximation of the solution
- **Convergence speed :** as we have seen in section 1 the algorithm already gets the solution in less steps than the best solution from Lab 1 , as it takes about 4 to 6 generations to find the solution which is much better than 10 to 15 generations.
- **Speed of runtime :** as we have seen through all of this experiment ,we added a lot more calculations than in Lab1 algorithms ,which means of course that given the same parameters would yield the lab1 algorithms to be faster , yet if we lower the population size we can get a close fight between the two .
-