JULY 14, 2021

ECE 457A: ASSIGNMENT 3

DAIVIK GOEL, BOSCO HAN, LICHEN MA

Question 1

Formulate a fitness function used to evaluate a solution

The fitness function used to evaluate a solution is the inverse of the integral squared error (ISE):

$$\frac{1}{ISE}$$

$$ISE = \int_0^T (e(t))^2 dt$$

Where e(t) is the error signal in time domain.

Thus, to optimize the PID controller, we need to minimize the ISE value. In the code, this logic is handled by using the python control library.

Genetic Algorithm with population of 50, generations = 150, crossover probability = 0.6, mutation probability = 0.25

Crossover operator:

The crossover operation first generates a random uniform value between 0 and 1, if that number exceeds the cross_over_rate defined in the problem, one of the parents' chromosome is returned. Otherwise, one combination of the parents' KP, TD or TI values is mixed together and returned with equal probability.

There are a total number of 6 different combinations of two parents' KP, TD, and TI values should crossover occur. If crossover doesn't happen, all of either parent 1 or parent 2's genes are preserved and passed to the offspring.

	KP	**TD**	**TI**
Parent:	1	1	2
	1	2	1
	1	2	2
	2	1	1
	2	1	2
	2	2	1

Code of crossover operation:

```
    def crossover(self, c1, c2):
        # no crossover applied, one of the parent is returned:
        if random.uniform(0, 1) > self.crossover_rate:
            if random.uniform(0, 1) < 0.5:
                return c1
            else:
                return c2

        rand = random.uniform(0, 1)
        if rand < 1.0/6:
            return Chromosome(c1.kp,c1.td, c2.ti)
        elif rand < 2.0/6:
            return Chromosome(c1.kp, c2.td, c1.ti)
        elif rand < 3.0/6:
            return Chromosome(c1.kp, c2.td, c2.ti)
        elif rand < 4.0/6:
            return Chromosome(c2.kp, c1.td, c1.ti)
        elif rand < 5.0/6:
            return Chromosome(c2.kp,c1.td, c2.ti)
        else:
            return Chromosome(c2.kp,c2.td, c1.ti)
```

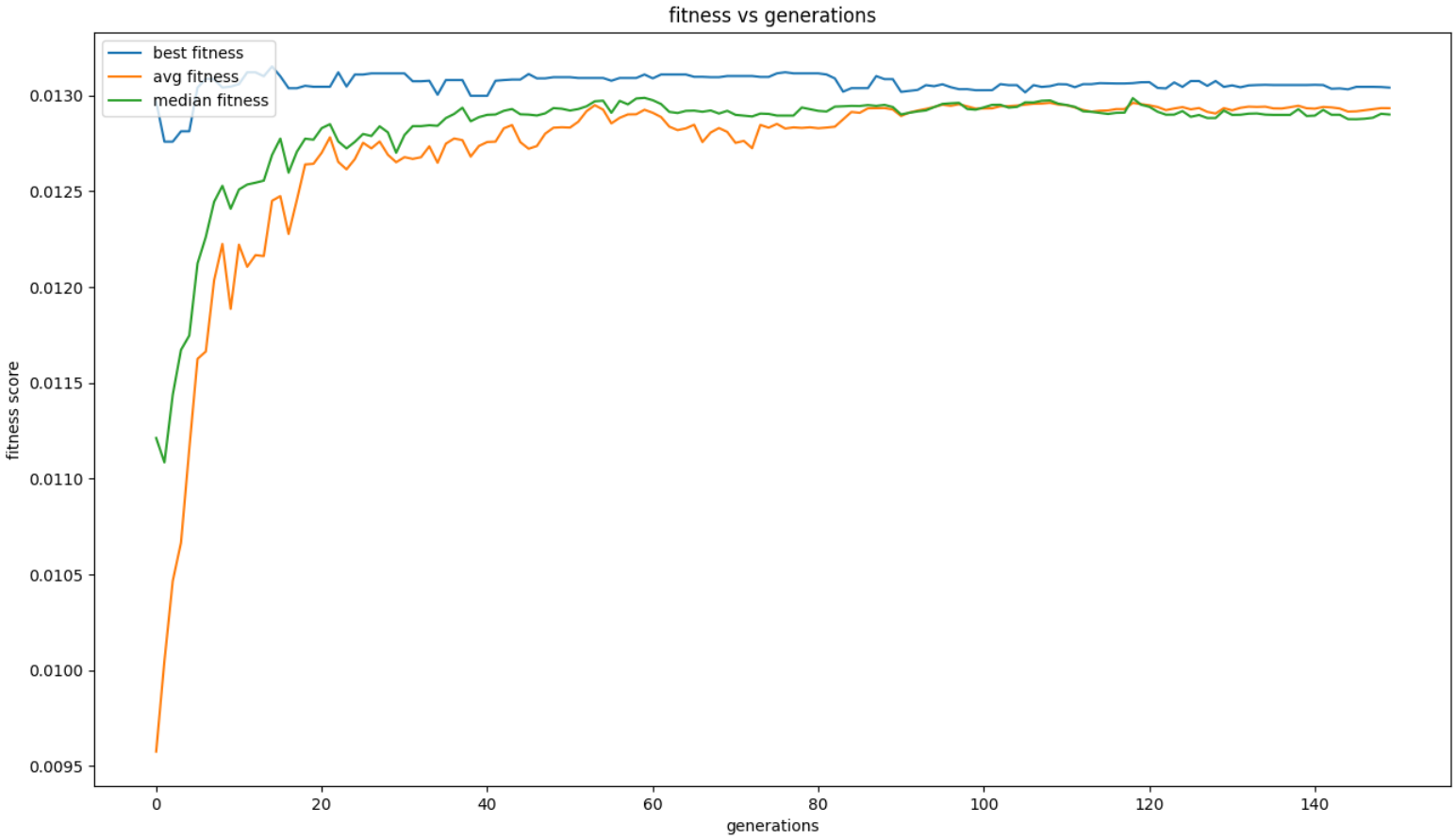
Mutation operator:

In mutation, we proceed only if random number generated is below the mutation probability provided. Thus, mutation will not occur all the time. Similar to the crossover operator, mutation adjusts values of KP, TD, TI to probe for a better fitness score of the PID controller. When mutation happens, it alters the values of KP, TD or TI of the chromosome under consideration with 1/3 probability for each case. The values changed via mutation are also adjusted according to the allowable range for KP, TD, and TI. The mutated genes are only accepted if they fall within the acceptable range. Finally, to prevent infinite loops from getting stuck with negative values for the tunable parameters, as a result of exploring too far, we regenerate new parameter values if they ever become negative.

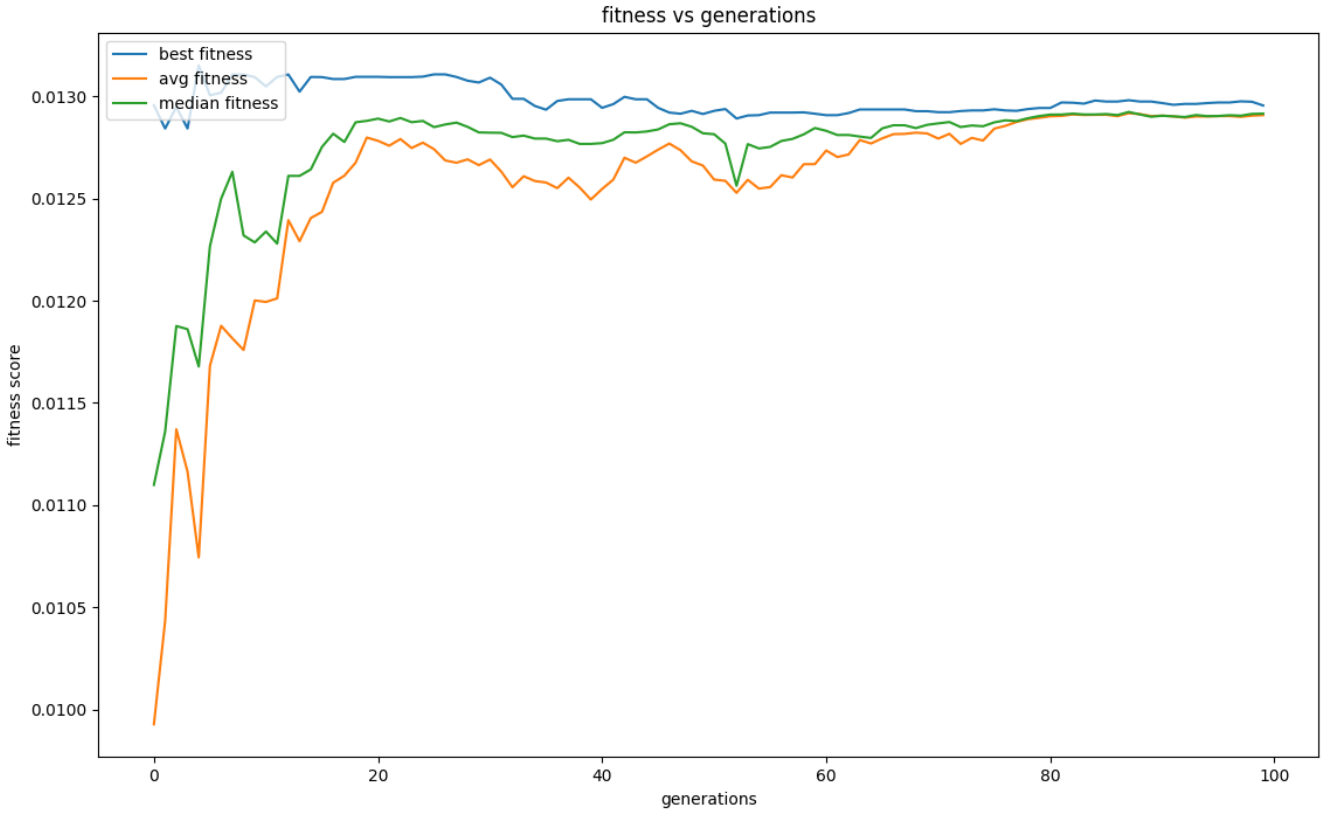
```
    def mutation(self, chromosome):
        if random.uniform(0, 1) > self.mutation_prob:
            return chromosome

        chromosome = self.mutate(chromosome)
        # only return if curr mutated chromosome is valid
        while not self.chromosome_valid(chromosome):
            chromosome = self.mutate(chromosome)

        return chromosome
```

```
    def mutate(self, chromosome):
        rand_num = random.uniform(0, 1)
        if rand_num < self.mutation_prob / 3:
            chromosome.kp = round(chromosome.kp + random.uniform(-0.1, 0.1), 2)
        if chromosome.kp < 0:
            chromosome.kp = round(random.uniform(2, 8) + random.uniform(-0.1, 0.1), 2)

        elif rand_num < self.mutation_prob * 2/3:
            chromosome.td = round(chromosome.td + random.uniform(-0.14, 0.14), 2)
        if chromosome.td < 0:
            chromosome.td = round(random.uniform(1.05, 9.42) + random.uniform(-0.14, 0.14), 2)

        elif rand_num < self.mutation_prob:
            chromosome.ti = round(chromosome.ti + random.uniform(-0.036, 0.036), 2)
        if chromosome.ti < 0:
            chromosome.ti = round(random.uniform(0.26, 2.37) + random.uniform(-
0.036, 0.036), 2)

        return chromosome
```

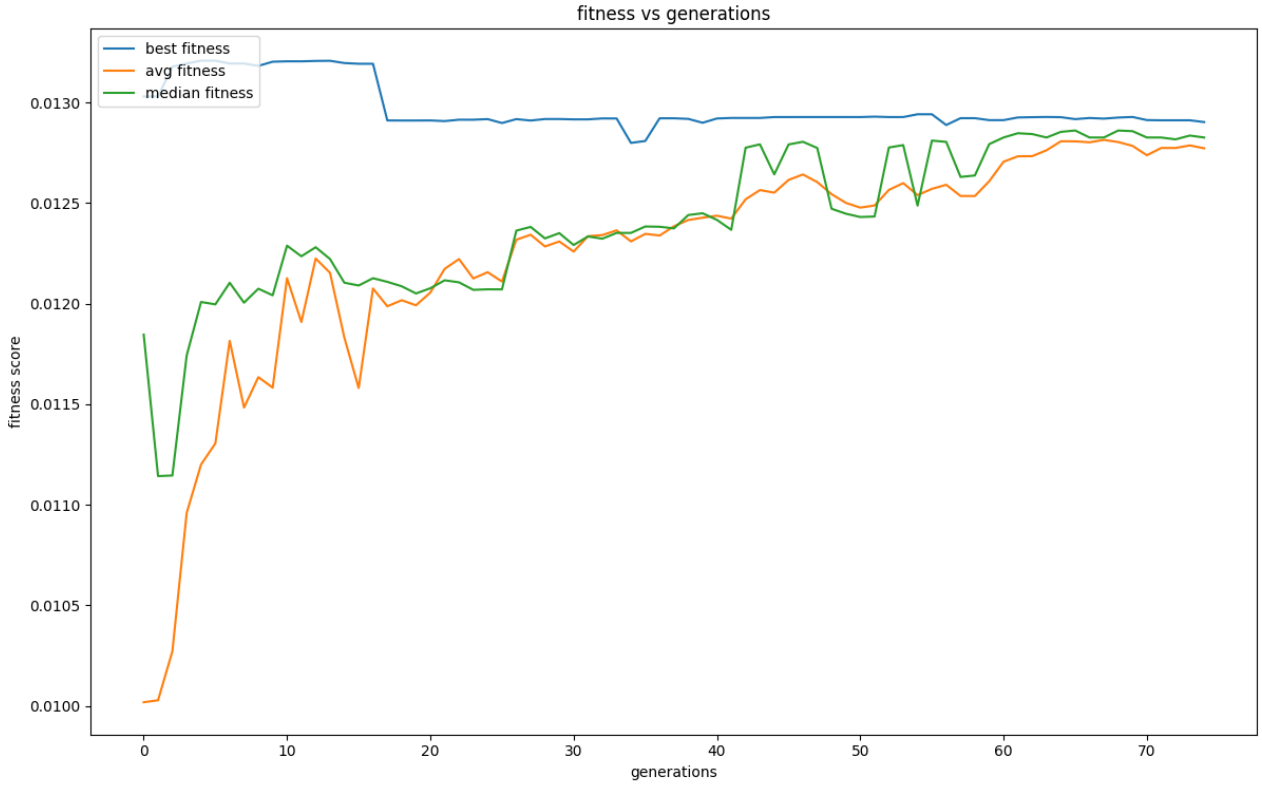
Plot of fitness vs generation (150 generations, 50 population)



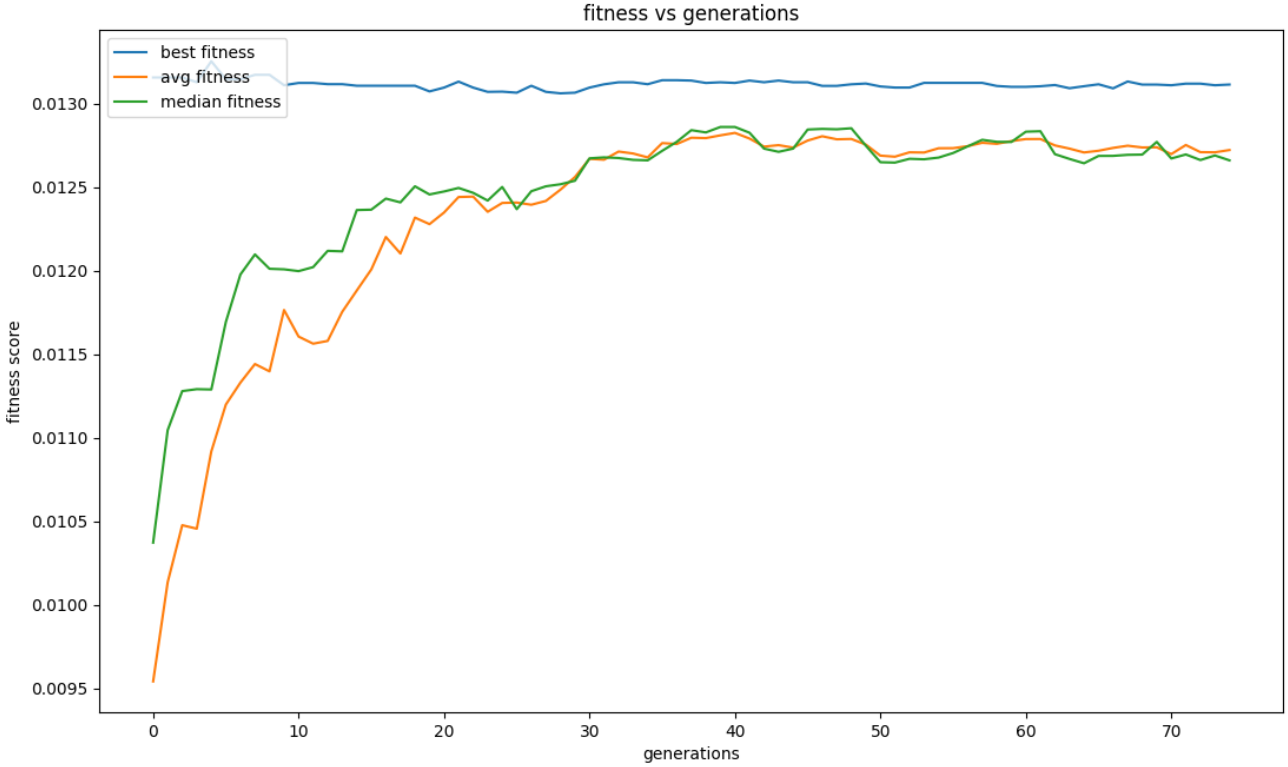
plot of fitness vs generation (100 generations, 50 population)



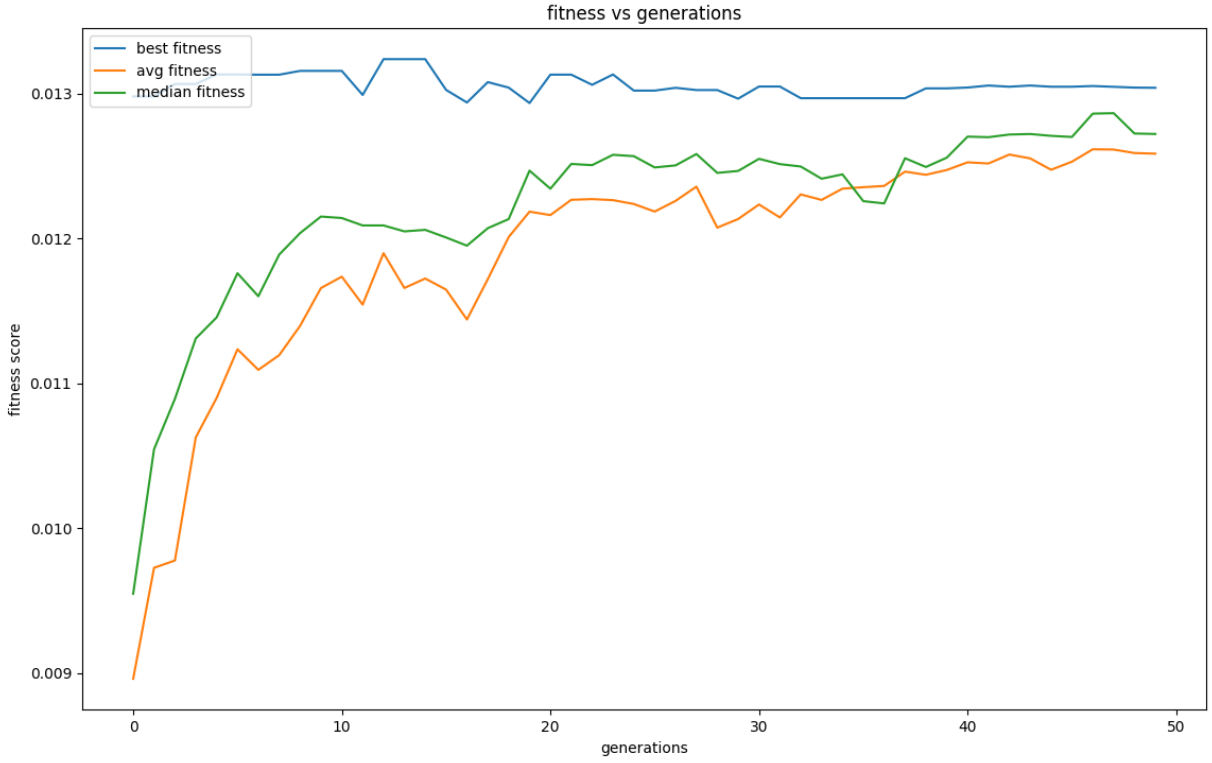
plot of fitness vs generation (75 generations, 50 population)



plot of fitness of fitness vs generation (75 generation, 100 population)

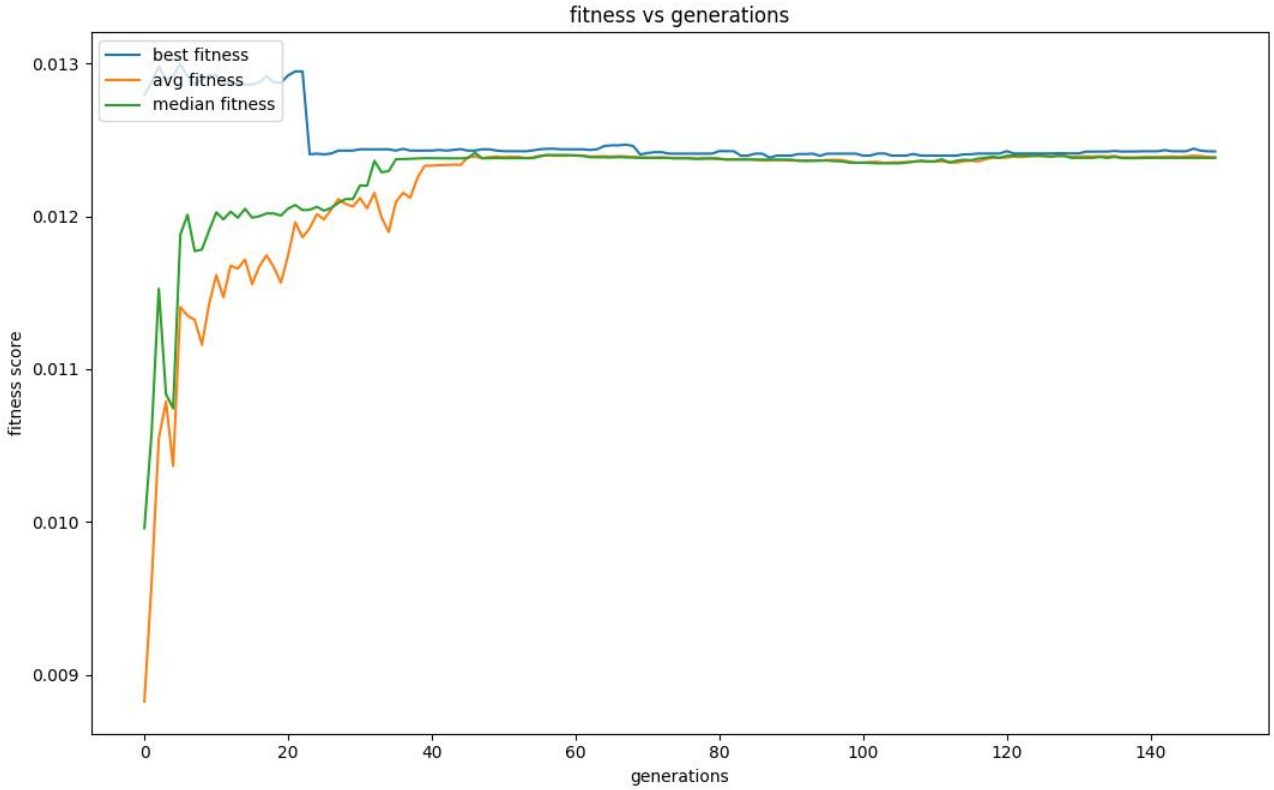


plot of fitness of fitness vs generation (50 generation, 80 population)



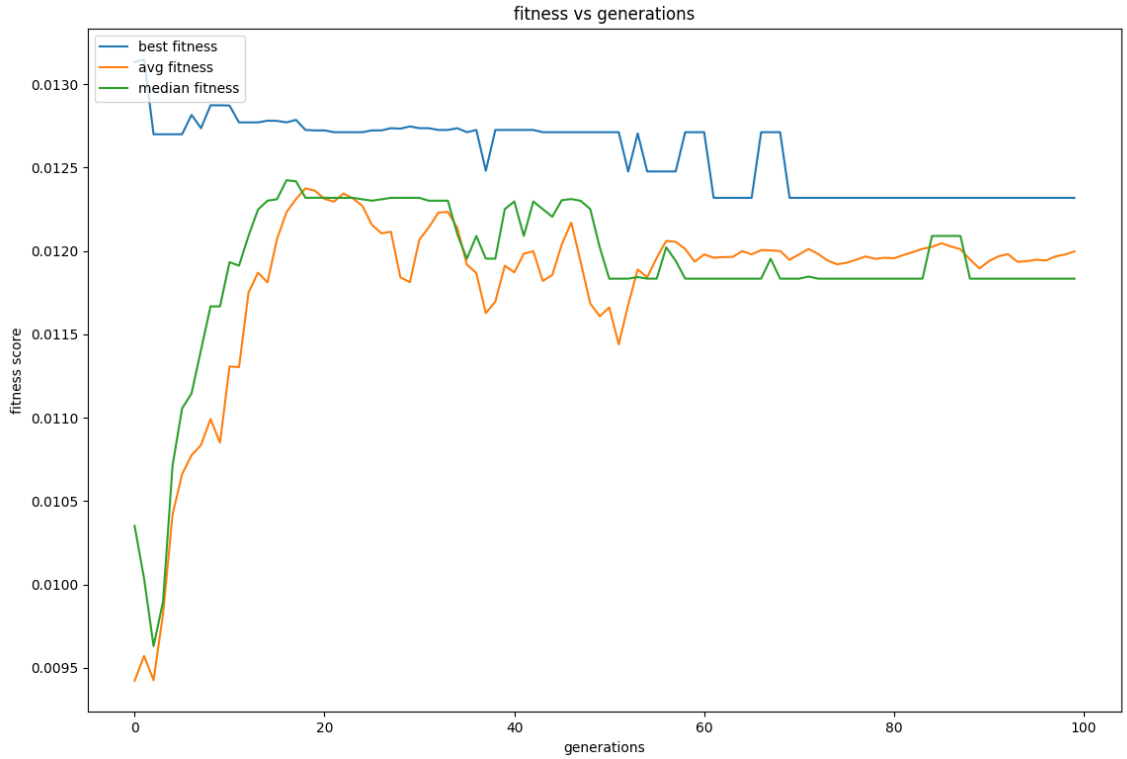
Plot of fitness of fitness vs generation (150 generation, 30 population)

It can be seen that with less population, the stability of the fitness values will not be as good as that of a very populated genome. In previous runs, the population were at least above 50, thus we did not have any sudden drops of the fitness score.



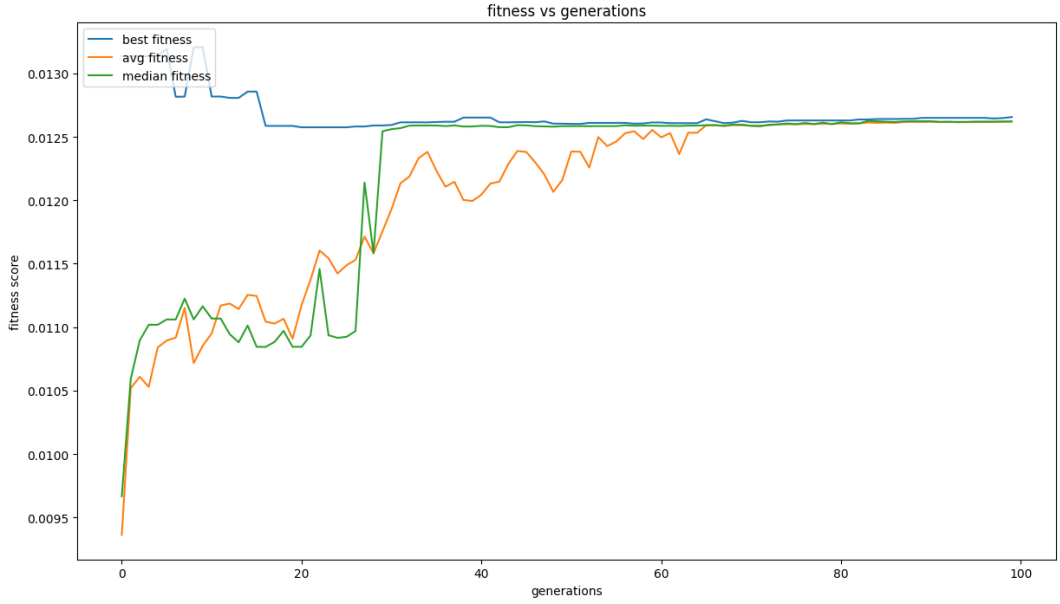
Effect of lowering mutation probability = 0.05, 50 population 100 generations

Because of a much lower mutation probability, the genes fitness scores are more likely to become stagnant, as offsprings tend to keep the parents' genes without any changes.



Effect of lowering crossover rate = 0.15, 50 population 100 generations

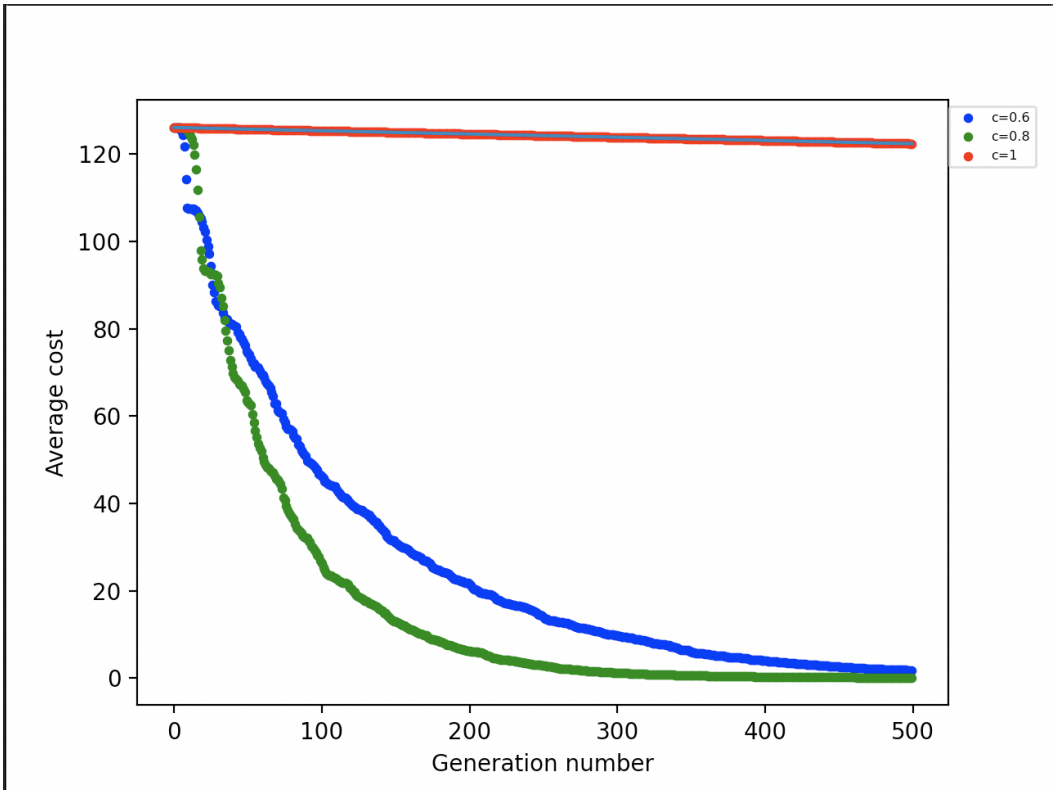
In this case, it seems like the general population took longer to find and converge at a good fitness value. Similar to mutation, this is because offsprings are less likley to have their genes altered (mix and matched) from their parents, and the parents' exact genes are more likely to be passed on to the children making genetic variations take longer.



Question 2

Using the pseudocode provided in Figure 3 to build the adaptive (1+1)-ES for minimizing the 10 dimensional sphere function. The source code takes into accound the domain of [-5.12,5.12] as well as simulates for 500 generations and averages the 50 cost values at each generation. For further reference please look at adaptive.py and its corresponding README to gain further insight into how the following graphs were generated.

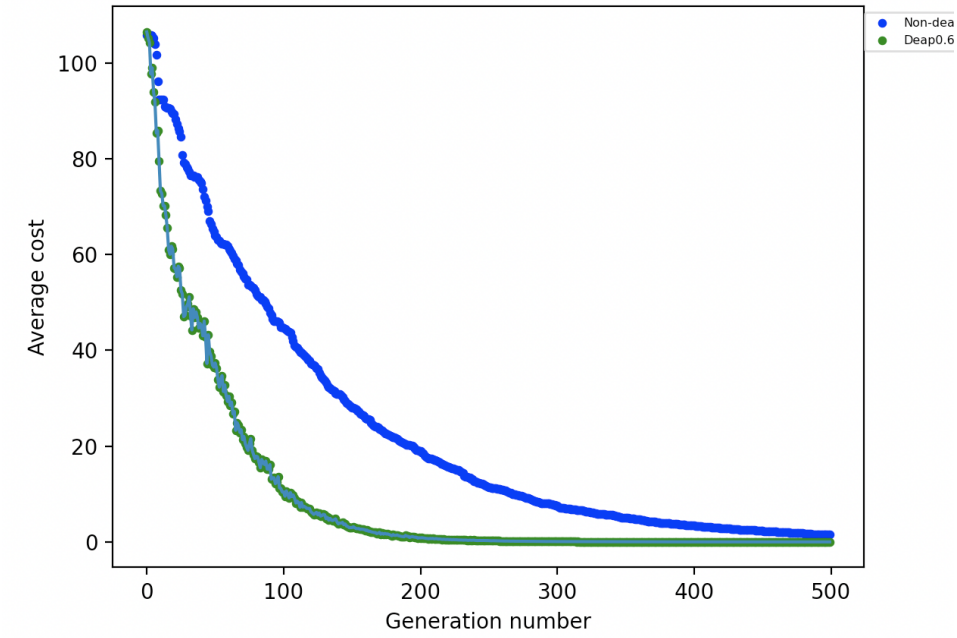
Plot of fitness of average cost values vs generation number (for c=0.6,0.8 & 1.0)

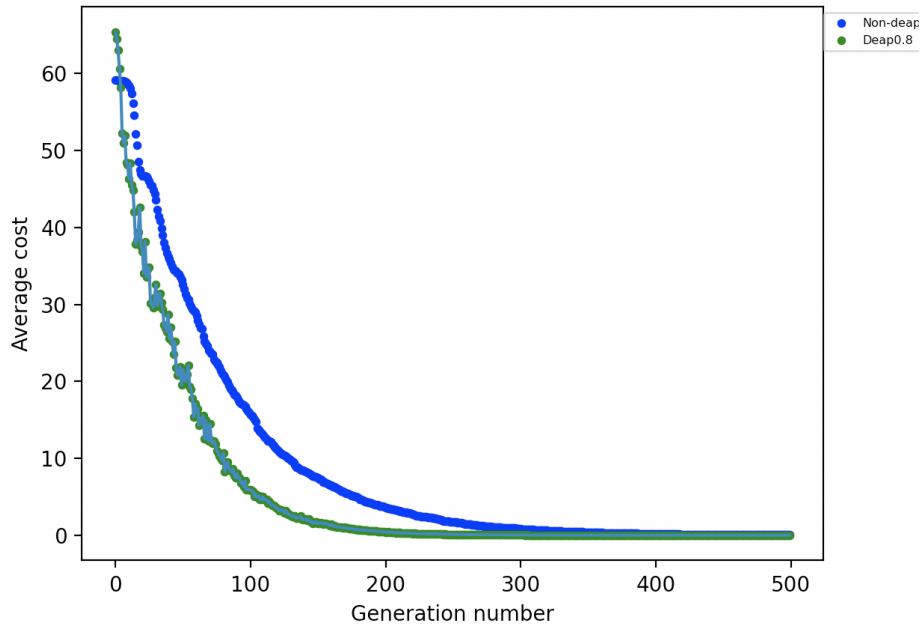


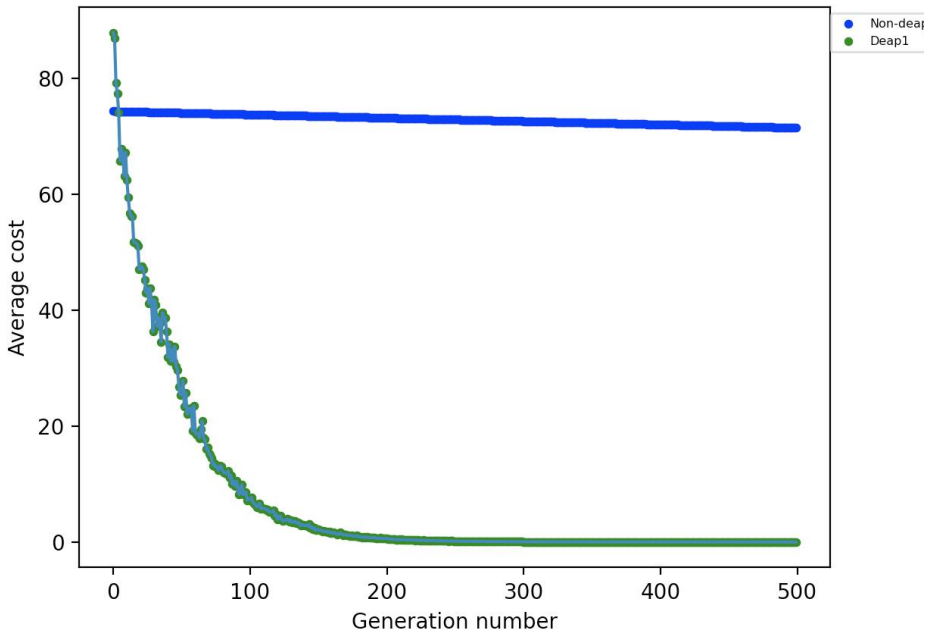
We can see from the graph that the value of c=0.8 gives the best performance in our adaptive (1+1)-ES. This follows from the 1/5 success rule of Rechenberg which states that ratio of successful mutations to all mutations should be 1/5 - c is used to increase or decrease the search space/variance to maintain this ratio.

After implementing and testing the ES code, we can use the DEAP platform to do the same thing and compare results. The DEAP implementation uses the one plus lambda algorithm with lambda set to 1 to mimic the setup for our (1+1)-ES algorithm. See the one_plus_one_deap function in adaptive.py for the code and specifics regarding the implementation. In order to compare the DEAP and non-DEAP solutions, the two functions are executed 50 times and a

graph of Average cost vs Generation number is created for c=0.6, 0.8 and 1 for comparison. The figures are presented below:





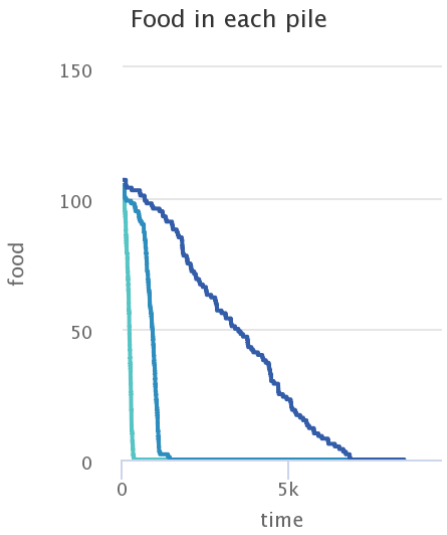


We can see that the DEAP implementation of the algorithm performs much better than the ES code implemented by this group above. The average cost improves (drops) much faster for the DEAP code and reaches a more optimal solution at a much earlier generation number.

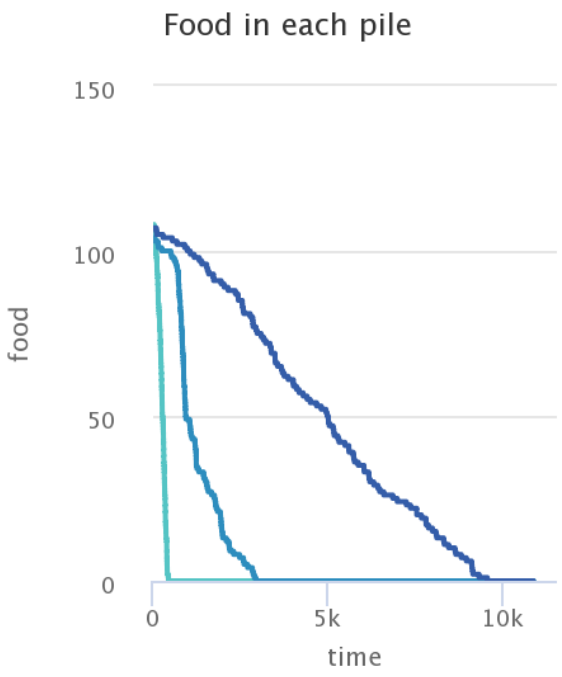
Question 3

Using the NetLogo web interface we get to see the ANT model in action. For this question we will try varying the population size, diffusion rate and evaporation rate to see what effect it has on the time it takes to get the food from the piles.

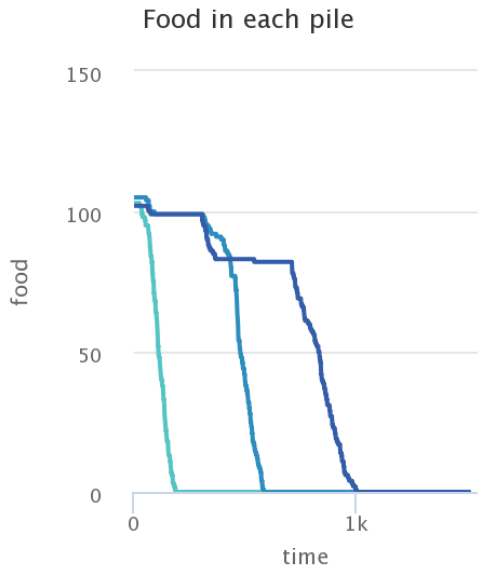
This is our base case with population set to 30, diffusion rate set to 40 and evaporation set to 10.



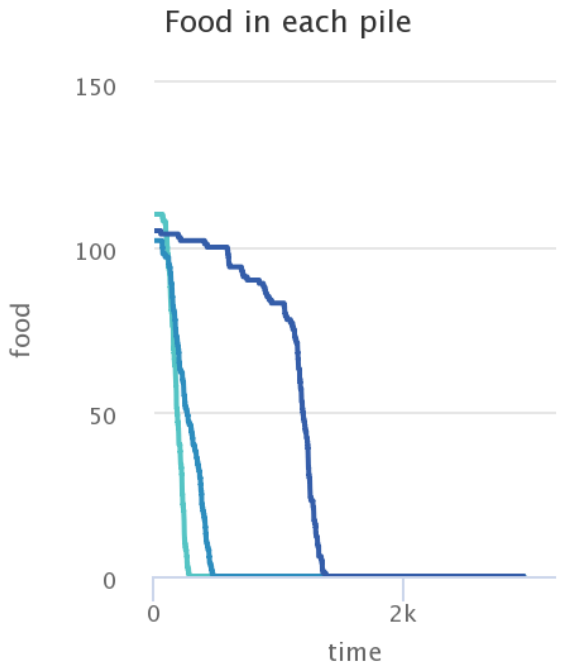
In increasing our population to 50 and keeping the rest of the parameters the same we see longer time for pile 2 and 3 to be destroyed. In fact, it almost looks like the time increase is directly proportional to the increase in population. As the population went up 1.6x the time for the piles to be finished also seemed to go up the same amount.



In increasing our population to 100 and keeping the rest of the parameters the same we get the following graph. The time for each of the piles drastically decreases.

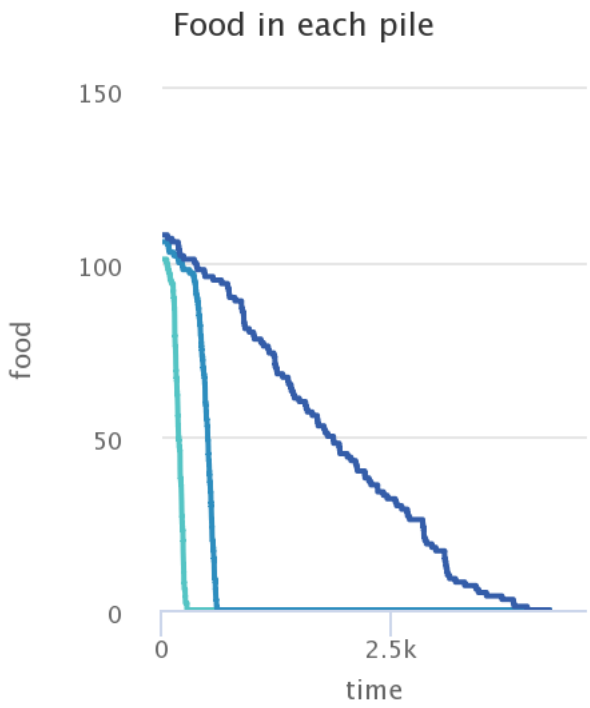


In increasing the diffusion rate to 80 while having a 100 population and the evaporation to 10 like was previously we get the following graph.



With the diffusion rate increased we see the food in the piles overall decrease quicker with the exception pile 2 which takes longer.

Increasing the evaporation rate from 10 to 20 while keeping all other parameters the same gives us the following graph.

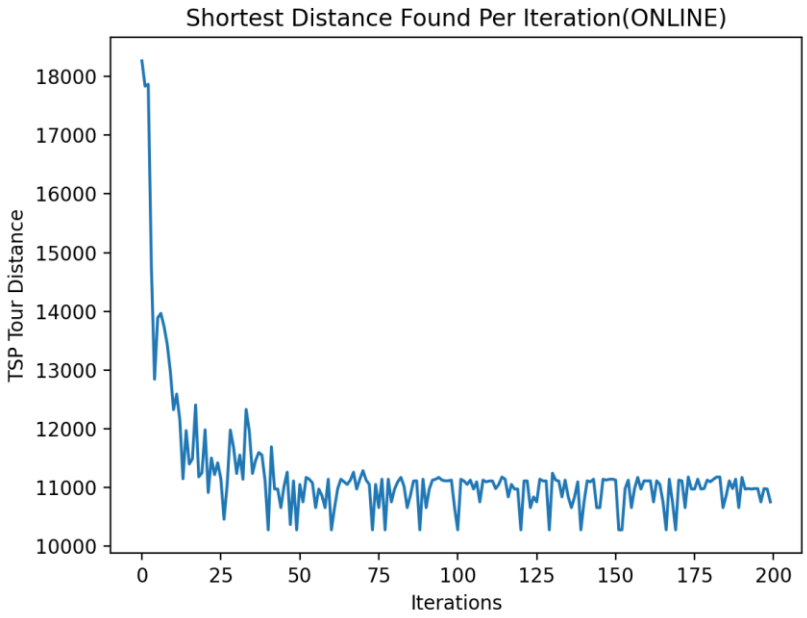


The graph resembles the previous one but piles 3 takes around 3 times the amount of time to be fully eaten.

Below are the results from the code implementation. The code itself can be found in the attached repo and it has a readme for your convenience. These are the parameters we are using in the base case. We change them as required in the different parts.

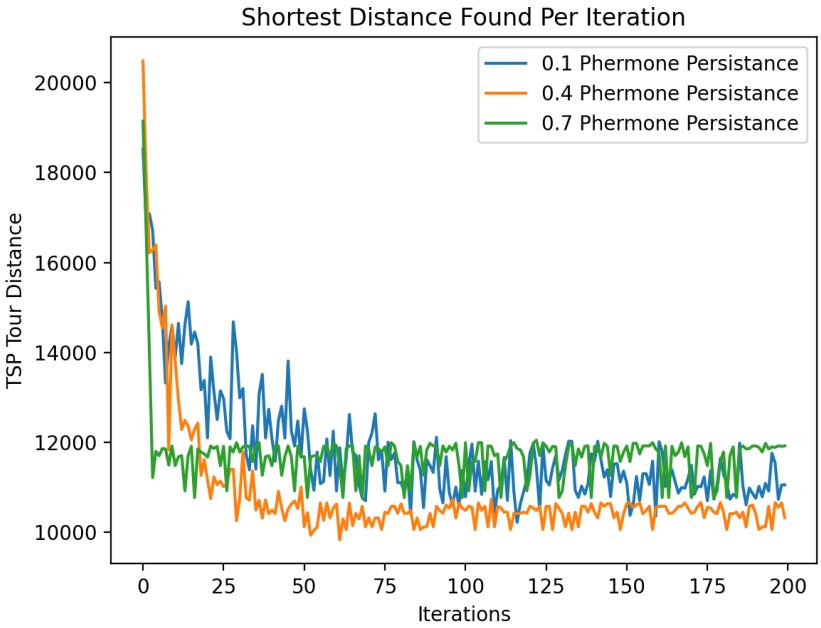
```
alpha,beta,base_ph ant_pop,phermone_decay,state_transition,online_phermone,Q,iter =
1,1,1,10,0.4,0.5,True, 5000,200
```

For our base case with the values seen above we got this as our graph:



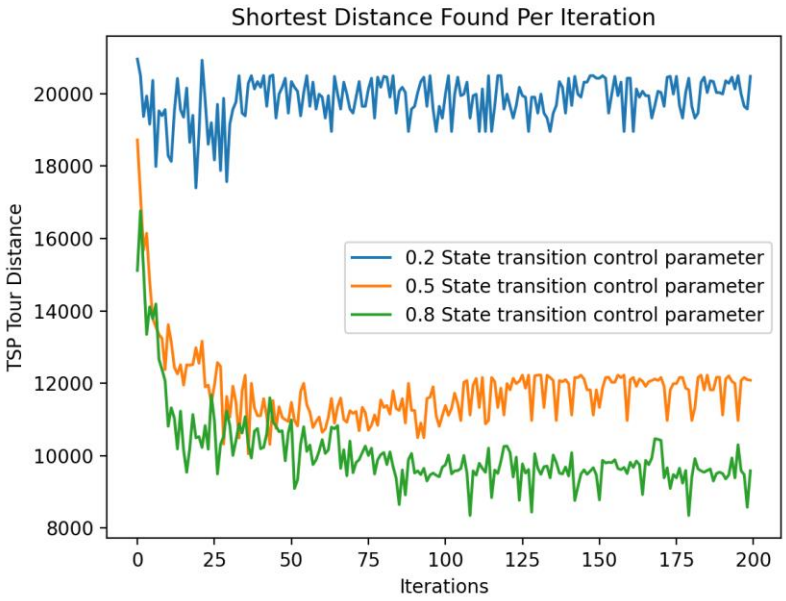
As you can see, as the iterations increase, the found distance also massively decreases until we reach a point of small variations. This is probably due to the increasing pheromones highlighting the shortest path for future iterations.

When we change the values of the Pheromone Persistence, we get the following graph:



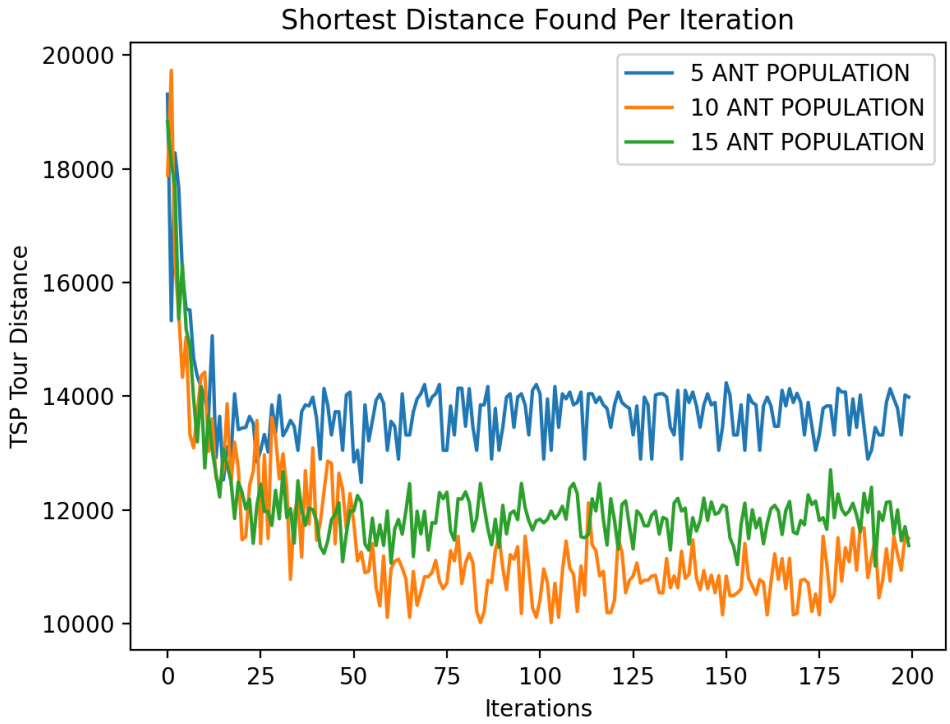
All the graphs reach the same point of small variations, but we see that a higher pheromone persistence reaches it much faster. This makes sense given a higher persistence should lead to more ants following previously found shorter routes.

When we change the state transition parameter, we get the following:



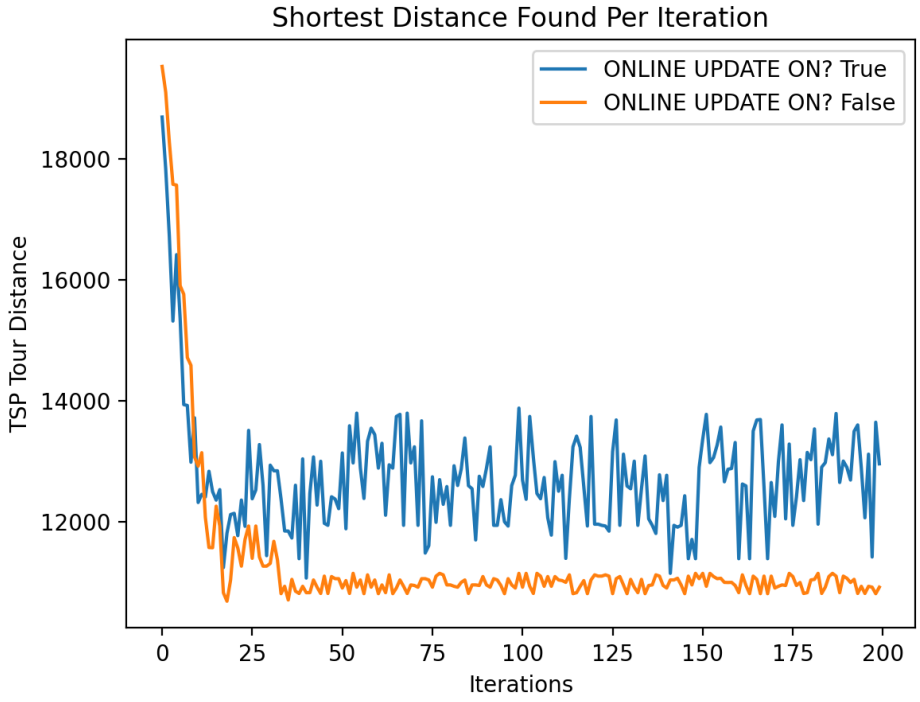
In our code a smaller state transition values favors a pheromone-based city selection rather than probability. We see a drastic difference as the one that chooses solely based on the highest pheromone stays constantly at the top of the distance. This seems to indicate that the a certain level of randomness in the ant's selection is required to get the best solution possible.

When we change the population size of the ants, we get the following graph:



This graph shows us that the slope of the curves to get to each population respective resting points is similar. However, the rest points themselves are greatly affected especially with the 5 ants resting almost 2000-3000 more distance than the 10 and 15 population ants. It seems that population of ants are required to reach a certain value which once reach renders the impact of the difference in ants to be minimal.

Finally, when switching between offline updating and online delayed updating we get the following graph.



Although the slopes to their respective resting points are similar, we see that offline updating is quite lower consistently than online updating. We also see less variation in the offline updating line while we have a lot more variation with online updating. This seems to make sense as with offline updating since we are only updating the pheromone of the best ant in the iteration, the paths that the ants take in future iterations should and do have a distinct advantage. We see this reflected in the graph above.