Instituto Superior Técnico

Computational Intelligence for the IoT

2021/2022 - Second Semester, Second Period

Report Project 2:
Heuristic Optimization

Tomás Coheur - 93621

Contents

1 The Objective

For this project we needed to tackle a version of the *Traveling Salesman Problem*. We have 100 *Ecopoints* all throughout the city and we need to figure out fastest route between the ones that are full in less than 20 minutes. In our problem we start at the center of the city which is represented by the letter "C" and come back to it at the end. We are given a list with the *Ecopoints* we have to visit, represented by "E" and their number (ex: E1 for *Ecopoint* 1). We also have a file called "Project2_DistancesMatrix.xlsx" where we can find the distance between two *Ecopoints*. To answer this problem we will create two programs that can both calculate the fastest route. The first one will be based on a *Genetic Algorithm* using the python library *deap* and the second one will use the *Ant Colony Optimization* method using the library *pants*.

2 Genetic Algorithm

2.1 Creation of the population:

The first step to the *Genetic Algorithm* is the creation of the individuals and the population which is a group of individuals. For our problem we decided to have our individuals as a vector with the name of the *Ecopoints* to visit. Each individual will have a different vector from the previous one because we shuffle them so we have multiple routes in the first generation. The population is then filled with a number of individuals that we define and that we will specify when showing the results. In the figure bellow we have the code that we used to create our population (figure 1).

```
def shuffle_list():     # shuffle the ecopoints list to have different individuals
    random.shuffle(list_of_ecopoints)
    return list_of_ecopoints


def get_ecopoint():     # returns the next ecopoint from the list
    global counter  # counter to know when the individual has all ecopoints in its list
    if counter == nr_ecopoints_to_visit:    # we have to create a new individual
        shuffle_list()
        counter = 1
        return list_of_ecopoints[0]
    counter += 1
    return list_of_ecopoints[counter-1]


creator.create("FitnessMin", base.Fitness, weights=(-1.0,))     # we want the smallest value that our algorithm provides
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()
# Attribute generator
toolbox.register("distances", get_ecopoint)
# Structure initializers
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.distances, nr_ecopoints_to_visit) # individual is a list of ecopoints
toolbox.register("population", tools.initRepeat, list, toolbox.individual)  # population is a list of individuals
```

Figure 1: Code for the creation of the individuals and the population

2.2 Mating and Mutation:

After having our population created and having defined the fitness of our individuals, which in our case is the distance traveled between all the *Ecopoints* in order, we can start to mutate and cross our individuals. For the crossover we decided to go with an ordered crossover, so we used *tools.cxOrdered*, because we didn't want repeated values inside our individuals so we had to use a method that would remove a part of an offspring and then append a part of another offspring without putting two of the same values in the final product. For the mutation we went with a simple shuffle of indexes because once again we didn't want to have two of the same values in an individual. To do this we used the function *tools.mutShuffleIndexes*. The probability of mutation was tweaked a little, as was the probability of crossover, and we ended up keeping the values 0.5 and 0.2 for the crossing and the mutation probability respectively. We can see bellow the code used to run this two algorithms on our GA (2).

```
    # Apply crossover and mutation on the offspring
    for child1, child2 in zip(offspring[::2], offspring[1::2]):
        if random.random() < CXPB:
            toolbox.mate(child1, child2)
            del child1.fitness.values
            del child2.fitness.values

    for mutant in offspring:
        if random.random() < MUTPB:
            toolbox.mutate(mutant)
            del mutant.fitness.values
```

Figure 2: Code for the crossover and mutating algorithms

2.3 The Results:

The statement tells us that a garbage truck visits on average 20 *Ecopoints* and very rarely exceeds the 50. Because of this we will show results for routes that pass through 20 and 50 *Ecopoints* with populations of 1000 individuals and probabilities of crossing and mutating of 0.5 and 0.2 respectively.

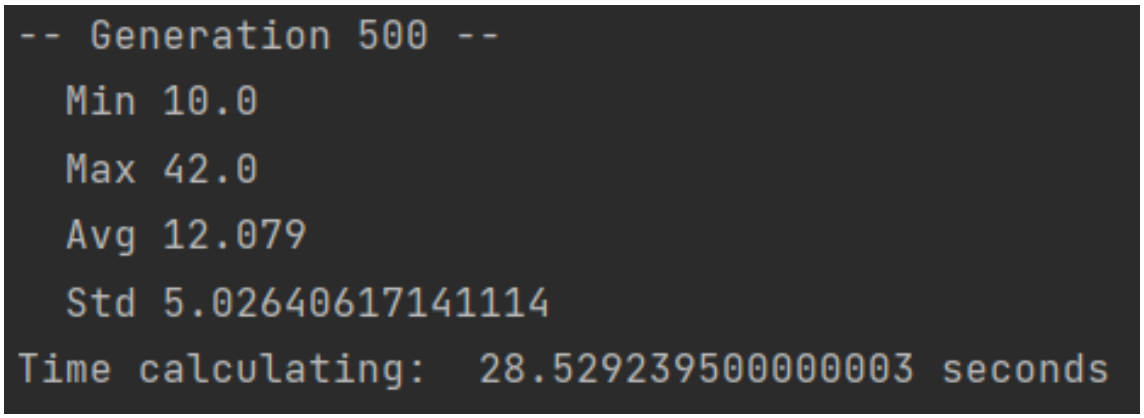

Figure 3: Results for a visit of 20 Ecopoints by the GA

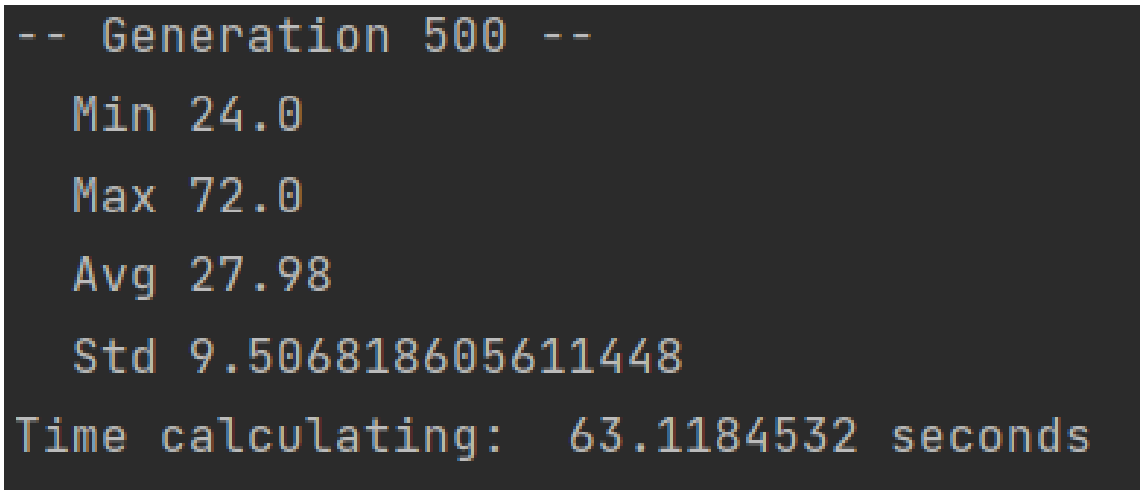

Figure 4: Results for a visit of 50 Ecopoints by the GA

Because we didn't see any improvement on changing the probabilities of mutation and crossing we decided to keep them as they were at the beginning. The high value of the individuals is due to the fact that there will be a better chance to have good individuals in the early stages which would give us the best route in less generations. Keep in mind that we only end the run of our GA after the 500 generations to give it time to get the best route even if it was already found generations ago.

3 Ant Colony Optimization

3.1 Creation of the World

The Ant Colony Optimization, as the name indicates, has for origin the way ants in a colony travel and always find the fastest route from point A to B using pheromones. These will stay in the shortest path longer because they are more condensed and ants will then choose that route. The generation of this algorithm

with *pants* was pretty easy because we just had to give a starting point, a list of points to visit and a function that calculates the distance between two points. This is what we can see bellow in figure 5 where *world.data(0)* is the command to set the value for the staring point which in our case is "C".

```
13  def get_distance(actual_ecopoint, next_ecopoint):
14      return distance_file[actual_ecopoint][next_ecopoint]
15
16
17  world = pants.World(list_of_ecopoints, get_distance)
18  world.data(0)
```

Figure 5: Code for the creation of the world for our ants

After creating the world we just had to create a *pants.solver*, that takes our world and gives us a solution for the problem.

3.2 The Results:

Like the *Genetic Algorithm* we will show two results for our tests of the *Ant Colony Optimization*.

```
Distance:  9
['E9', 'E12', 'E1', 'E17', 'E7', 'E19', 'E10', 'E11', 'E15', 'E3', 'E14', 'E6', 'E20', 'C', 'E18', 'E2', 'E16', 'E4', 'E13', 'E8', 'E5']
Time calculating:  0.40078680000000005 seconds
```

Figure 6: Results for a visit of 20 Ecopoints by the ACO

```
Distance:  17
['E8', 'E50', 'E24', 'E16', 'E13', 'E36', 'E4',
Time calculating:  1.1854179999999999 seconds
```

Figure 7: Results for a visit of 20 Ecopoints by the ACO

If we want to compare with the GA we can see that our ACO is much faster to give results and that it normally gives us lower distances. However there is a problem to keep in mind with our results. As we can see in the figure 6 there is a C in the middle of the best route which means that the ants went through the center again when they couldn't. We did not find a way to fix this and it could alter the values of the distance a bit.

4 The Final Test

The final test of these algorithms is to run them using a list containing the total number of *ecopoints* that exist in the *Oeiras Municipality*. So, bellow we can see the length of the shortest route that runs through all 100 *Ecopoints* for both our algorithms.

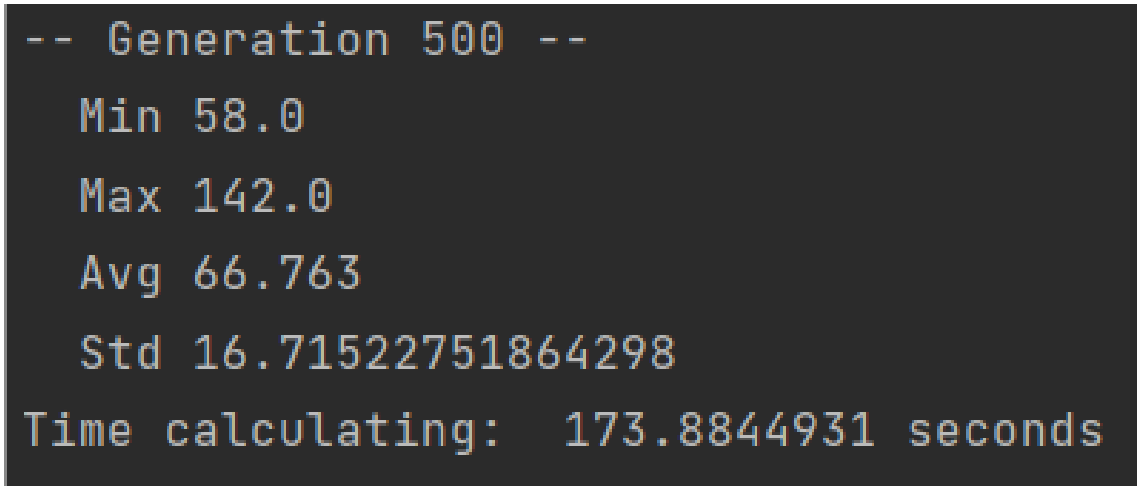

Figure 8: Results for a visit of 100 Ecopoints by the GA

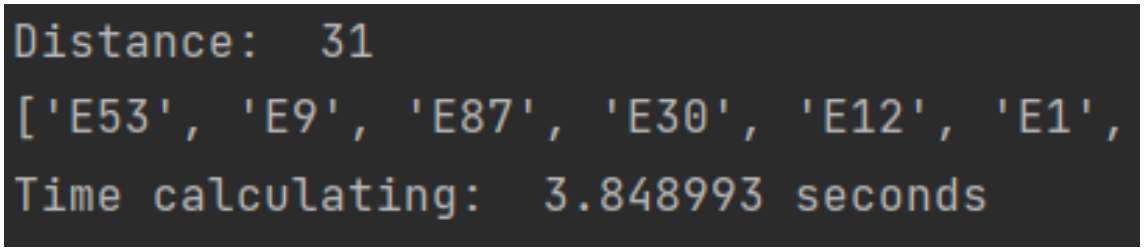

Figure 9: Results for a visit of 100 Ecopoints by the ACO

As we can observe if we compare the two images is that the figure 9 shows us a way lower completion time than the the figure 8. Indeed the GA takes almost three minutes to complete the calculations while the ACO only takes 4 seconds. More than that is the fact that the *Ant Colony Optimization* gives us a way better distance than the *Genetic Algorithm* even though this last one takes much more time to complete.

5 Conclusion

As we have seen the *Ant Colony Optimization* is way faster than the *Genetic Algorithm* to resolve a *Travelling Salesman* type problem while also giving better

results. Even though we did not test our GA for a 20 minute long period to see what would the output be, it already takes more than two minutes to finish 500 generations that don't even give us good results. Because of all this I would say that the best algorithm to implement and to give to the *Oeiras Municipality* is the *Ant Colony Optimization* that is very fast and gives the garbage trucks more than enough time to finish their shifts on time.