SmartEcoRoute

Daniel Lopes

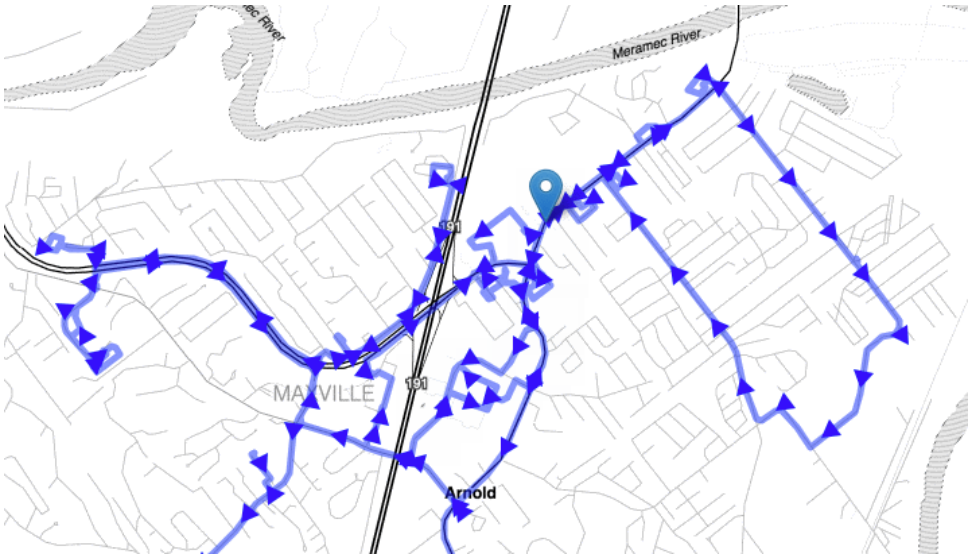
Department of Electrical Engineering,
Computer Engineering & Informatics

July 5, 2022

1 Objective

Routing problems are extremely common and known for their complexity and enormous quantity of combinations. The purpose of this work is to assign a route to garbage trucks of a certain location that will collect the recyclables for all Ecopoints that are full, in the most efficient way possible. The program has only 20mins to execute and obtain the best route, to avoid possible delays to the working hours. This intelligent system, given a list of up to 100 Ecopoints, returns the shortest route that starts from Central (C), runs through all the Ecopoints locations in the list (E_i) and returns to the central.

Also note that the data for the algorithms is already normalized and with the required format (distances matrix for every Ecopoint).



2 Implementation

For the implementation of this program two methods were tested: **Genetic Algorithms** and **Ant Colony Optimization**. The former is the most tunable one and with a lot of potential for improvement, whereas the latter is an algorithm which is usually very suited for problems like this and runs very efficiently for certain conditions.

Both algorithms will read the list of points to be considered and obtain the distances between them. Then they're supposed to process the data and obtain the shortest path that goes through all the points.

Let's now look at the implementation details for both algorithms.

2.1 Common Functions

Firstly we shall consider the following piece of code that contains common functions that were used for both algorithms and which helps the process of reading the data from the files (.csv for inputs and .xlsx for the distances matrix) and also calculating the runtime for a given function.

```
import time
import pandas as pd
import numpy as np


def read_points_and_distances(input_path='./input/list_of_points.csv',
                distances_path='./data/Project2_DistancesMatrix.xlsx'):
    points = None
    if 'csv' in input_path:
        points = pd.read_csv(input_path, header=None)
    elif 'xls' in input_path:
        points = pd.read_excel(input_path, header=None)

    distances = pd.read_excel(distances_path, header=0, index_col=0)
    distances.columns = range(distances.columns.size)
    distances.reset_index(drop=True, inplace=True)  # reset indexes

    return points, distances


def get_nodes(points, distances):
    if points is not None:
        nodes = points.to_numpy()[0]
        nodes = np.insert(nodes, 0, 0)
        # insert c in the list of nodes
        nodes = nodes.tolist()
    else:
        nodes = distances.columns.to_list()
    return nodes


def normalize_final_path(path):
    while path.index(0) != 0:
        path.append(path.pop(0))
    return path


def run_with_timer(func, *args):
    st = time.time()
    result = func(*args)
    duration = time.time() - st
    return result, duration
```

Listing 1: Python - Common functions

2.2 Genetic Algorithms

Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. In simple words, they simulate "survival of the fittest" among individual of consecutive generation for solving a problem. Each generation consists of a population of individuals and each individual represents a point in search space and possible solution. [1]

To be able to solve this problem with genetic algorithms, one must define what is an Individual of the population and create the appropriate fitness, mutation and the crossover functions.

The objective of this algorithm will be to minimize the fitness function, i.e. to obtain the lowest distance.

Let's consider an individual as a vector of ecopoints: [0,1,5,7,44] (*chromossome*) where the order matters and the vector represents the path to be taken. To generate individuals we can simply apply a random sample to the points being considered and obtain multiple combinations of those points.

The fitness function is a simple sum of all of the distances between the points in an Individual, considered as a circle or a loop, i.e. it also accounts for the return distance. Consider the following example:

```
individual = [1, 3, 4, 6]
distances: [1->3: 1, 3->4: 2, 4->6: 4.1, 6->1: 1]
fitness: 1 + 2 + 4.1 + 1 = 8.1
```

In this example, the **fitness** is the sum of distances between 1 and 3, 3 and 4, 4 and 6 and also the return distance from 6 to 1.

For the **mutation** function the method that is being used is the **interchanging** method, which means that a mutation is the switch of position between two genes.

Finally, the **crossover** function is *not* a straight up single or two point crossover, because the Individuals are **ordered** and **unique**, which means that if we applied a simple crossover function, the offspring would contain duplicate points which goes against the requirement of passing through an Ecopoint only once. For this reason, the crossover function to be considered here was the **Single-point Ordered Crossover** [2], which is a very simple function that solves all of the mentioned concerns. It is true that Deap contains a function that supposedly does an ordered crossover (cxOrdered)[3], it assumes that each element is boolean, or binary, which is not the case for this problem since it was decided to not use binary encoding. With this in mind, the simpler approach works perfectly well.

All of these functions and concepts were implemented in the following piece of code.

```python
from deap import tools, creator, base
from lib import utils
import matplotlib.pyplot as plt
import random
import time

points, distances = utils.read_points_and_distances(input_path="")
nodes = utils.get_nodes(points, distances)
possible_genes = nodes

def get_distance(index_a: int, index_b: int):
    return distances.loc[index_a, index_b]

def fitness_func(individual):
    """Calculate the fitness of an individual
    by summing the distances between each gene (eco)"""
    total_dist = 0
    if len(individual) >= 2:
        for eco1, eco2 in zip(individual[::1], individual[1::1]):
            dist = get_distance(eco1, eco2)
            total_dist += dist
        return_home_dist = get_distance(individual[-1], individual[0])
        total_dist += return_home_dist
    return [round(total_dist, 2)]

def mutate(individual, indpb):
    """Interchange single gene position for another"""
    prob = random.random()
    if prob < indpb and len(individual) >= 2:
        pos_i = random.randint(0, len(individual)-1)
        pos_f = random.randint(0, len(individual)-1)
        temp = individual[pos_f]
        individual[pos_f] = individual[pos_i]
        individual[pos_i] = temp
    return individual

def generate_individual():
    """Generate random sample of genes from possible genes"""
    return random.sample(possible_genes, len(possible_genes))

def cxOnePointOrdered(ind1, ind2):
    """Custom single-point ordered crossover"""
    size = min(len(ind1), len(ind2))
    k = random.randint(0, size-1)
    c1, c2 = ind1[:k],  ind2[:k]
    for val1, val2 in zip(ind1, ind2):
        if val2 not in c1:
            c1.append(val2)
        if val1 not in c2:
            c2.append(val1)
    return c1, c2
```

Listing 2: Python - GA initial functions and Imports

The next step in implementing this Genetic Algorithm is very standard when using the **Deap** library.

We first register all of the functions and strategies mentioned above.

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0, ))
creator.create("Individual", list, fitness=creator.FitnessMin)
toolbox = base.Toolbox()
toolbox.register("gen_ind", generate_individual)
toolbox.register("individual", tools.initIterate,
                 creator.Individual, toolbox.gen_ind)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("evaluate", fitness_func)
toolbox.register("mate", cxOnePointOrdered)
toolbox.register("mutate", mutate, indpb=0.1)
toolbox.register("select", tools.selTournament, tournsize=3)
```

Listing 3: Python - GA registering Deap functions into toolbox

With everything correctly configured, we start to run the generations. The function for running a single generation is as follows:

```
def run_generation(cxpb, mutpb, hof, pop, g, logbook):
    print("-- Generation %i --" % g)

    offspring = toolbox.select(pop, len(pop))
    offspring = list(map(toolbox.clone, 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

    invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
    fitnesses = list(map(toolbox.evaluate, invalid_ind))
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    pop[:] = offspring
    hof.update(pop)
    best_ind = hof[0]
    logbook.record(gen=g, best_individual_distance=best_ind.fitness)
    print(logbook.stream)
```

Listing 4: Python - GA single generation function

Finally, the last portion of code needed for this algorithm is the main portion where it calls all of the generations, logs the improvements and chooses the best individual while plotting the results in a line chart. It is also where we can configure how many generations we want or where we can limit the time that this algorithm runs for.

```python
def run_ga(cxpb=0.6, mutpb=0.3, gen=1000, pop_size=500, max_time=120, threshold=20):
    total_time = 0
    hof = tools.HallOfFame(1)
    random.seed(64)
    pop = toolbox.population(n=pop_size)
    print("Start of evolution")
    fitnesses = list(map(toolbox.evaluate, pop))
    for ind, fit in zip(pop, fitnesses):
        ind.fitness.values = fit

    g = 0
    logbook = tools.Logbook()
    logbook.header = ["gen", "best_individual_distance"]
    while g < gen and total_time < max_time - threshold:
        g = g + 1
        _, time = utils.run_with_timer(
            run_generation, cxpb, mutpb, hof, pop, g, logbook)
        total_time += time
    print(f"-- End of (successful) evolution (Generations: {g}) --")
    best_ind = hof[0]
    return best_ind, logbook

def plot_ga_logbook(lb: tools.Logbook):
    generations = lb.select("gen")
    distances = lb.select("best_individual_distance")
    normalized_distances = [float(d.values[0]) for d in distances]
    normalized_generations = [int(g) for g in generations]
    plt.scatter(normalized_generations, normalized_distances, color="orange")
    plt.plot(normalized_generations, normalized_distances)
    plt.xlabel("Generation")
    plt.ylabel("Distance")
    plt.title("Genetic Algorithm Evolution")
    plt.show()

if __name__ == '__main__':
    st = time.time()
    # run for a limited amount of time or generations
    best_ind, logbook = run_ga(0.6, 0.3, 100, 500, 60*20)
    path = utils.normalize_final_path(best_ind)
    print("==== Best Result ====")
    print(f'Path: {path}')
    print(f'Distance: {float(best_ind.fitness.values[0]):.2f}')
    plot_ga_logbook(logbook)
    print(f"Took {time.time() - st:.2f} seconds to execute.")
```

Listing 5: Python - GA main execution

2.3 Ant Colony Optimization

Ants live in community nests and the underlying principle of ACO is to observe the movement of the ants from their nests in order to search for food in the shortest possible path. Initially, ants start to move randomly in search of food around their nests. This randomized search opens up multiple routes from the nest to the food source. Now, based on the quality and quantity of the food, ants carry a portion of the food back with necessary pheromone concentration on its return path. Depending on these pheromone trials, the probability of selection of a specific path by the following ants would be a guiding factor to the food source. [4]

Having this in mind, the implementation of this algorithm was more straightforward. This means that by using the ACO-Pants library this algorithm can be up and running in a few lines of code.

This makes the use of this algorithm extremely practical for novice developers.

Consider the following code as the complete implementation of the ACO algorithm.

```
import pants
from lib import utils
import matplotlib.pyplot as plt
import time

points, distances = utils.read_points_and_distances(input_path="")
nodes = utils.get_nodes(points, distances)

def get_distance(index_a: int, index_b: int):
    return distances.loc[index_a, index_b]

def run_iteration():
    world = pants.World(nodes, get_distance)
    solver = pants.Solver()
    solution = solver.solve(world)
    return solution

def run_for_duration(func, seconds, *args):
    threshold = 20
    total_time = 0
    result_1st, duration_1st = utils.run_with_timer(func, *args)
    total_time += duration_1st
    func_results = [result_1st]
    while (total_time + duration_1st) < (seconds - threshold):
        result, duration = utils.run_with_timer(func, *args)
        func_results.append(result)
        total_time += duration
    return func_results
```

```python
def plot_aco_iterations(results):
    x = range(1, len(results) + 1)
    y = [p.distance for p in results]
    plt.scatter(x, y, color="green")
    plt.plot(x, y)
    plt.xlabel("Iteration")
    plt.ylabel("Distance")
    plt.title("Ant Colony Optimization x Iterations")
    plt.show()


if __name__ == "__main__":
    st = time.time()
    # run many times to obtain the best possible result in a given time
    time_to_run = 20 # 20 seconds (example)
    results = run_for_duration(run_iteration, time_to_run)
    best_aco = None
    for res in results:
        if (best_aco is None or res.distance < best_aco.distance):
            best_aco = res

    path = best_aco.tour
    path = utils.normalize_final_path(path)
    print("==== Best Result ====")
    print(f'Path: {path}')
    print(f'Distance: {best_aco.distance:.2f}')
    plot_aco_iterations(results)
    print(f"Took {time.time() - st:.2f} seconds to execute.")
```

Listing 6: Python - ACO complete implementation

By looking at this implementation there are a couple of things that standout.

First the way this library works is by creating a world with the nodes being considered and the function to obtain the distance between two nodes. With the world created, a simple solver will then run on top of that world and perform the ACO algorithm behind the curtains and try to find the best path possible, with the lowest distance.

Note that the ACO algorithm implemented in the mentioned library already accounts for the return distance, which means that we do not have to worry about it right away as we did with the Genetic Algorithm.

This algorithm can be implemented manually of course, but for the purpose of this project, the objective is mainly to compare the performance of both algorithms and assess which one is better suited for the problem at hands.

Finally, we can also see that it contains functions for continuously repeating the execution of the solver for a given time, so that randomness plays in our favor and we can obtain the best result possible; and of course there's also the function for plotting these results, eventhough the plot will have a very random appearance.

3 Results & Evaluation

On this section, we evaluate and compare the results obtained for each algorithm and decide on which is the most promising for this problem.

Also note that for the genetic algorithm the crossover and mutation probabilities are 60% and 30%, respectively. This was chosen after a couple of experiments where it proved to be the most effective in achieving the best results quicker.

3.1 Different Inputs

The following tables contain the results for each of the techniques on differently sized inputs. In the case of the genetic algorithm, it was ran a single time against multiple combinations of generation and population sizes, but not limited by time. For the Ant colony optimization, a single solver iteration was executed.

Input	Population & Generation	Runtime	Best Distance	Best Path
[4, 6, 8, 12, 66, 2]	500 & 100	3.09 secs	8.0 km	[0, 66, 4, 8, 2, 12, 6]
[4, 6, 8, 12, 66, 2]	100 & 10	0.09 secs	8.2 km	[0, 12, 2, 8, 4, 6, 66]
[4, 6, 8, 12, 66, 2, 55, 1, 5, 19, 20]	100 & 100	1.10 secs	9.5 km	[0, 6, 55, 2, 4, 8, 5, 12, 1, 19, 20, 66]
[4, 6, 8, 12, 66, 2, 55, 1, 5, 19, 20]	500 & 100	4.45 secs	9.0 km	[0, 55, 2, 4, 8, 5, 12, 1, 19, 6, 66, 20]
First 30 EcoPts	500 & 100	9.52 secs	21.4 km	–
First 30 EcoPts	500 & 500	48.04 secs	17.7 km	–

Table 1: Different input results for GA

Input	Runtime	Best Distance	Best Path
[4, 6, 8, 12, 66, 2]	0.05 secs	8.0 km	[0, 66, 2, 4, 8, 12, 6]
[4, 6, 8, 12, 66, 2, 55, 1, 5, 19, 20]	0.11 secs	9.0 km	[0, 20, 66, 6, 19, 1, 12, 5, 8, 4, 2, 55]
First 30 Ecopoints	0.62 secs	12.2 km	–

Table 2: Different input results for ACO

Interestingly, we can see that at the start, with a few points and 500 individuals and 100 generations for the GA, both algorithms behave similarly, however when we add more ecopoints, the Genetic Algorithm starts lacking behind the ACO even with 500 individuals and 500 generations which means it would require further tuning to get slightly closer to the ACO.

3.2 Route through all 100 Ecopoints

Having in mind what we saw in the previous section, we can start taking a guess on which of these techniques will be more efficient and effective in finding a better path that goes through all Ecopoints.

To run these algorithms for all ecopoints and in order to perform a tryout that simulates real world conditions, these algorithms were left running for 20min. In the case of GA this means more generations, and in the case of ACO it means that given its randomness, it might be more likely to obtain a better result, eventually.

We can by start looking at the results for the **genetic algorithm:**

This algorithm obtained the lowest distance of **42.5 km** with an execution time of 20min, **4128 generations** and 500 individuals.

The evolution of the generations can be found in the following figure.

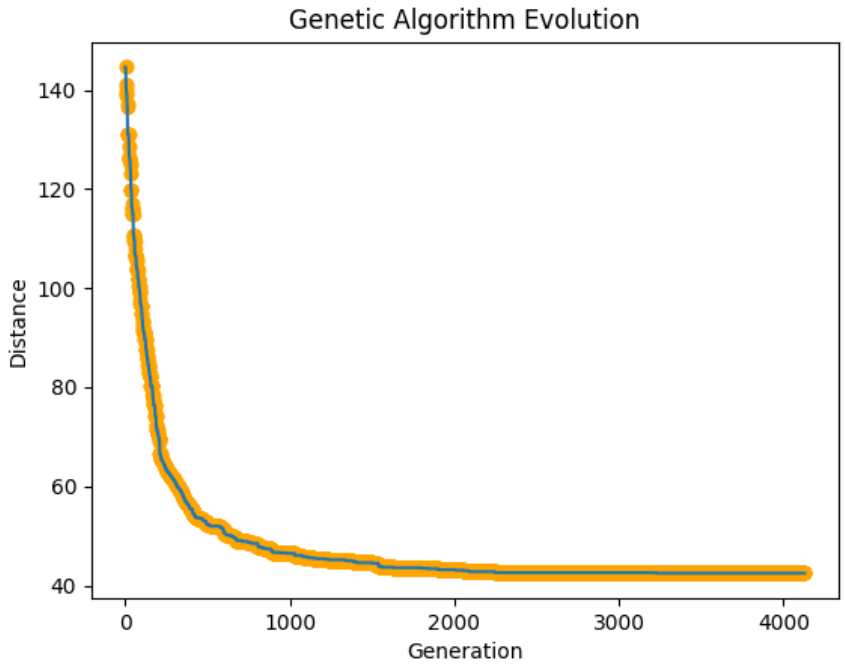


Figure 1: GA Performance Evolution

In the case of the **ACO algorithm**, 20 minutes of runtime turned out to reflect in around 230 executions of the algorithm.

Given the fact that this algorithm performs extremely well out-of-the-box, its random properties is what allowed it to obtain a slightly better result, given enough tries. This can be easily explained by looking at Figure 2 which tells us that the algorithm does not improve over time, rather it might approach lower values, eventually.

Looking at the Figure, we can see that ACO obtained very good results right at the start, with the lowest distance being **29.40km**, an extremely high improvement over the GA algorithm.

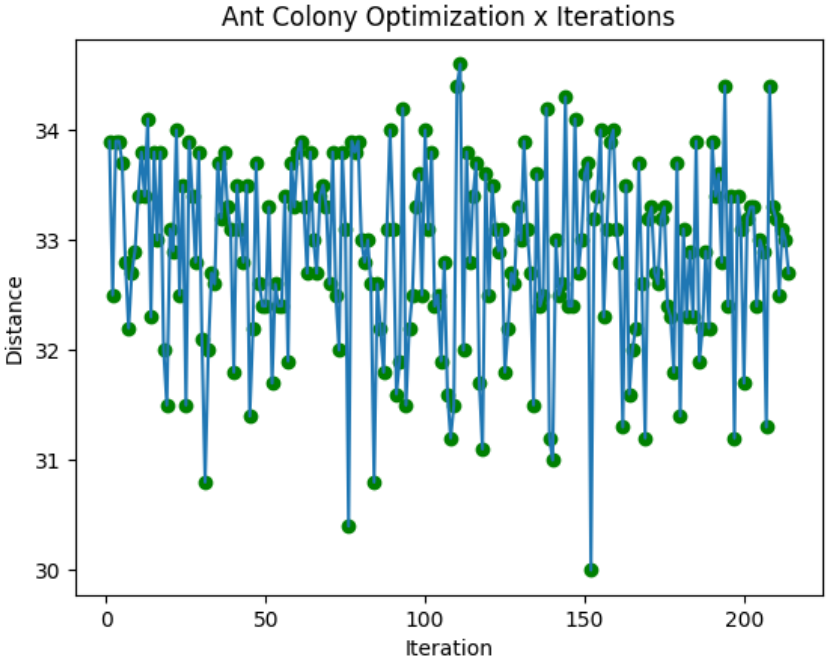


Figure 2: ACO Performance Per Iteration

3.3 Conclusions (Best approach)

This experiment allowed for some interesting conclusions regarding the use of these techniques in daily operations.

It is true that Genetic Algorithms are extremely versatile and can adapt to almost any type of Heuristic Optimization problems. These types of algorithms are very useful for problems with large search spaces where it's not viable/possible to tryout every possible combination. However, in some cases GAs need a lot of tuning and customization to be a viable approach.

For problems like path finding, in scenarios like the one being considered in this project or, for example, network optimization, Swarm intelligence might have the upper hand here. At least that's what was seen in this problem.

It was very clear that ACO was a better fit for this problem since it obtained much better results with a single iteration, which takes a very small amount of time to run, as opposed to GAs. Even more so if we let it get the most out of the 20mins and perform multiple iterations.

In my opinion, and based on the experimental observations, Ant Colony Optimization which is a Swarm Intelligence algorithm, is the clear winner when it comes to finding the most efficient route for collecting recyclable materials from Ecopoints.

4 References

[1] “Genetic Algorithms - GeeksforGeeks,” GeeksforGeeks, Jun. 29, 2017 https://www.geeksforgeeks.org/genetic-algorithms/ (accessed Jun. 28, 2022).

[2] IoT Evolutionary Computation Slides, João Paulo Carvalho

[3] “Evolutionary Tools — DEAP 1.3.1 documentation,” Readthedocs.io, 2022. https://deap.readthedocs.io/en/master/api/tools.html?highlight=logbookcrossover (accessed Jun. 28, 2022).

[4] “Introduction to Ant Colony Optimization - GeeksforGeeks,” GeeksforGeeks, May 15, 2020. https://www.geeksforgeeks.org/introduction-to-ant-colony-optimization/ (accessed Jun. 28, 2022).