NOVA IMS

Optimization Algorithms

Travelling Salesperson Problem (TSP)

Ivan Fuertes e20211480
Luis Silvano r20201479
Miguel Tomé r20201644
Pedro Sousa r20201611

Index

1. Introduction

The purpose of this report is to apply the algorithms **(Iterated Local Search (ILS) and Genetic Algorithm with multiple crossover points)** to cities dataset, to realize which algorithm is better for collecting conclusions about the shortest route to travel across several cities (visiting each one only once) and then returning to their original city. This report analysis allowed our group to note that we have different manners of reaching globally optimal solutions using the algorithms. The report is intended for the discipline of Computational Methods for Optimization, which is given by professors Leonardo Vanneschi and Nuno Rodrigues.

The report structure is divided into 2 sections. The **first section**, Code Explanation, we will explain the code procedures. The **second section**, a Comparative Analysis of the results **received by the Iterated Local Search (ILS) and Genetic Algorithm** with multiple crossover points.

2. Code Explanation

```
import math
import random
from copy import deepcopy

import matplotlib.pyplot as plt
import numpy as np
```

Code 1: Importation of Packages in both Algorithms

First, we start by importing the packages (math, random, deepcopy, matplotlib.pyplot and numpy) for both the **Interacted Local Search** and the **Genetic Algorithm**. It is important to import the packages, because it makes it possible and easier to construct both algorithms.

As for the data, we used the data from the Travelling Salesperson Problem (TSP) which consists of 1291 cities (in this case, their coordinates). To test the code, we created a sample of the dataset, called “test data” with only 20 cities and their coordinates, to test the code and see if it works properly.

```
def transform_data(dataset: list) -> dict:
    datadict = {}
    city_name = [i for i in range(len(dataset))]
    for city, coord in zip(city_name, dataset):
        datadict[city] = coord
    return datadict
```

Code 2: Assignment of List of City Coordinates into Dictionary

Then, from the list of coordinates we created a dictionary assigning each coordinate of a city to a specific name, in this case, the name of the cities it’s a numerical value, being the first city “0” and the last the length of the dataset less 1. We integrated both the name and the coordinates in a for loop to be able to assign them in a dictionary.

2.1 ILS Algorithm

2.1.1 Debug and Plot Functions

```
def show_results(solutions: list, attempts: list = None):
    for i in range(len(solutions)):
        if attempts is None:
            print(
                f"Epoch: {i} --> "
                f"Fitness: {solutions[i].fitness} | "
                f"Route: {solutions[i].route}" )
        else:
            print(
                f"Epoch: {i} --> "
                f"Attempts: {attempts[i]} | "
                f"Fitness: {solutions[i].fitness} | "
                f"Route: {solutions[i].route}" )
```

Code 3: Epoch, Fitness, Route ILS Algorithm Results Function

For this first block we created a function that shows the results of the ILS algorithm, this being the number of the Epoch, the fitness of that solution and the route of that specific solution. For this, we did a for loop with the list of solutions as length, so, for example, if we have a list of 10 solutions, we will have 10 epochs. We added the attempt variable, during the creation of the algorithm to test it, and as a final model, this part has no purpose.

```python
def show_best_result(solutions: list):
    print("-------------\nBest Result (lower fitness):")
    print(f"Route: {solutions[-1].route}   |   Fitness: {solutions[-1].fitness}")
```

Code 4: The Best Result from Set of Solutions

The "show_best_result" *function* is very similar from the one before and **from the list of solutions (it chooses the best solution of the set of solutions)**. Since the fitness tends to decrease the last will the best one (that's why we use the index of -1, to select the last solution). Other than that, we print a commentary on the route and the fitness of the solution.

```python
def plot_connections(data: list,
                     route: list):
    plt.figure()
    plt.scatter(*zip(*data), color='red', s=12)

    for i in range(1, len(route)):
        x1, y1 = data[route[i - 1]]
        x2, y2 = data[route[i]]
        x_values = [x1, x2]
        y_values = [y1, y2]
        plt.plot(x_values, y_values, 'bo', linestyle="--")
        plt.text(x1 - 0.50, y1 + 0.50, route[i - 1])
        plt.text(x2 - 0.50, y2 + 0.50, route[i])

    plt.show()
```

Code 5: Data Preparation for the Routes Visualization

Where we wanted to visualize the route itself, we received the data and the route that we wanted to visualize. Then after defining a scatter plot of the coordinates of each city, and with the correspondent's name, we did a for loop to plot a "--" line between one point and the previous point, using the order given in the route list.

```python
def plot_fitness(solutions: list):
    plt.style.use('seaborn')
    fitness = [solution.fitness for solution in solutions]

    ax = plt.axes(xlim=(0, len(fitness)), ylim=(np.min(fitness) - 0.5, np.max(fitness) + 0.5))
    ax.set_title('Fitness Results', fontsize=18)
    ax.set_xlabel('Epochs', fontsize=16)
    ax.set_ylabel('Fitness', fontsize=16)
    ax.tick_params(labelsize=12)

    epochs = [i for i in range(len(solutions))]
    plt.plot(epochs, fitness)

    plt.show()
```

Code 6: Fitness Evolution Plot along the Epochs

Finally, for the last function of the block, all it was left was a plot to see the evolution of the fitness along the epochs. To do this, we started by defining a function that received a list of solutions and after creating the graph style. We saved the fitness of each solution from the list of solutions in a new list and before plotting the epochs and fitness, we prepared the graph by changing the size and label of both axes, the title and the tick parameters. For the epochs, we just associated the index of the list of solutions to an epoch. For example, the first solution is in index 0, so the epoch will be 0, the same can be said for the second, and so on, until the last solution.

2.1.2. Solution

```
class Solution:
    def __init__(self, route: list):
        self.route = route
        self.fitness = self.__get_fitness()

    def __manhattan(self, p1, p2):
        return int(math.fabs(p1[0] - p2[0]) + math.fabs(p1[1] - p2[1]))

    def __get_fitness(self):
        fitness = 0
        for i in range(len(self.route) - 1):
            fitness += self.__manhattan(data[self.route[i]], data[self.route[i + 1]])

        fitness += self.__manhattan(data[self.route[-1]], data[self.route[0]])
        return fitness
```

Code 7: Distance between the cities, their final fitness of the routes

For the class of solutions, we defined that each solution had a route when it received a list of cities. Then, to get the fitness of each solution, we used another function to calculate the fitness, but, before defining this function, we had to define the functions to calculate the distance between two points or cities, **using the Manhattan Distance Metric**. After that, we could define the function to get the fitness of each solution. First, we created the variable fitness (equal to 0, just to have a temporary value) and then for each value (or city) in the route (excluding the last since it doesn't have the following city), we **calculate the Manhattan Distance** between that city and the next and then and sum all these distances of the route into the variable fitness. Since we must go back to the beginning of the route, for each time the route is done, we add the distance between the last city and the first one to achieve the final fitness of the route and get the correct path.

2.1.3. Local Search Algorithm

```
class StochaticHillClimber:
    def __init__(self, patience_threshold: int):
        self.threshold = patience_threshold
        self.solutions = []
        self.fitness = []
        self.attempts = []
```

Code 8: Stochastic Hill Climber Grouping Function

To start the **Local Search algorithm with the Stochastic Hill Climber** we firstly opened a class with the same name, to group all the functions referring to the Hill Climber. Inside, the first function triggers with the name of the same class. This function receives an integer as the patience threshold. Besides that, it creates 3 new empty lists, a solution list, a fitness list and the attempts list(used in the future for the purpose of allowing multiple trials until the patience threshold is met).

```
    def get_neighbor(self, sol):
        route = deepcopy(sol.route)

        # start from 1 to avoid changing the initial city
        i1 = random.randint(1, len(route) - 1)
        i2 = random.randint(1, len(route) - 1)

        # loop for avoiding to choose the same city
        while i2 == i1:
            i2 = random.randint(1, len(route) - 1)

        aux = route[i1]
        route[i1] = route[i2]
        route[i2] = aux

        return Solution(route)
```

Code 9: Route Neighbour

The next function, it is a function to get the neighbour from the route. We started by defining the route as a copy of the solutions and used the function route (defined previous) to consider the set of solutions as a route. Then, we chose two cities' indexes of the route to make the swap between them. In this we used the random function to choose a random integer from the list of cities that the salesperson will visit. To not allow the first city to change we started the random choice in the (index 1) jumping the first city of index 0. We used (len(route)-1) since the maximum value of the route length doesn't exist in the index of the same list. To attest that the cities that were randomly chosen are not equal, we did a while loop. This while loop runs when the cities are the same, and it will only leave the loop if the city2 is different than city1. For this we just assigned the variable inside the loop, and it will eventually leave when both cities are different. To make the swap we save the first city's index in an auxiliary variable, put the city of the first index as the second city chosen, and then the city of the second city as the first city, using the auxiliary variable (this variable had to be created, since we would lose the information about the first city when changing to the second). Finally, we just return the new route as a solution.

```
def fit(self, route: list):
    random.shuffle(route)
    s = Solution(route)
    self.solutions.append(s)
    self.fitness.append(s.fitness)
    self.attempts.append(0)
    attempts = 0

    while True:
        neighbor = self.get_neighbor(s)
        if neighbor.fitness <= s.fitness:  # <= because it is a minimization problem
            s = neighbor
            self.solutions.append(s)
            self.fitness.append(s.fitness)
            self.attempts.append(attempts)
            attempts = 0
        else:
            attempts += 1
            if attempts >= self.threshold:
                break
```

Code 10: Solution List Using Hill Climber Routes

At last, **to end the Hill Climber**, we defined a last function that also receives the route. First, we applied **the random shuffle** function to all cities inside the route, **and then save that route in a new variable as a Solution**. In the list of solutions created in the first function of the block, we append this route to the list also defined in the first function, we append the fitness of this route, and we set the attempts to 0 (the attempts function is used purely for testing, and to see if the algorithm was behaving correctly when we had in count the patience threshold). Then we used a while loop, by firstly define the neighbours with the function of neighbours created previously (where we get a random neighbour for the solution that we have). Then we test the fitness of that neighbour, and if it is smaller or equal, we will append it to the list of solutions and its fitness to the list of fitness. If not, we will add one unit to the number of attempts, and if the number of attempts is already equal or higher than the threshold defined at the beginning of the Hill Climber, we leave the while loop and end the Hill Climber.

It's important to say that each time we find a new solution the attempts reset, allowing the same number of attempts of getting a better solution every time it stagnates in a specific solution.

2.1.4 Iterate Local Search (ILS) Algorithm

```
class ILS:
    def __init__(self, algorithm, iterations: int):
        self.algorithm = algorithm
        self.iterations = iterations
        self.solutions = []
```

Code 11: Creating the Iterate Local Search (ILS) Algorithm

For the first block in the ILS algorithm, we created the class named ILS, and started with a function triggered by the same name. Although in the call of the function we can use another algorithm other than the Hill Climber, this part was made with the propose of using the Hill Climber as Algorithm, since it was the one, we were working on. With this said, we defined some variables with elements received in the function and add an empty list for the solutions.

```
    def perturbation(self, sol):
        route = deepcopy(sol.route)

        def random_cities():
            cities = []
            for num in range(6):
                while True:
                    city = random.randint(1, len(route) - 1)
                    if city not in cities:
                        cities.append(city)
                        break
            return cities

        indexes = random_cities()
        # this kind of swap works only with ranges of even number
        for i in range(len(indexes)//2):
            # start from the beginning
            i1 = 0 + i

            # start from the end
            i2 = len(indexes) - 1 - i

            # swap
            aux = route[indexes[i1]]
            route[indexes[i1]] = route[indexes[i2]]
            route[indexes[i2]] = aux

        return Solution(route)
```

Code 12: Performance of 6 Random Cities Swap

This second function has the purpose of performing a random six cities swap (when we have similar restrictions, then the swap performs the Hill Climber). We started by making a copy of the route (received as solution), then we created a secondary function called (*random_cities*) to choose the cities that we need to swap. Inside this secondary function, we open a new list and start a for loop of 6 iterations, and each time it saves the city index. Since we do not need to alter the first city, we did not allow the random function to pick this first city making sure the same city is not saved twice in the list of six cities. We also made sure that the chosen city was not in the list of cities using conditionals to test if a city was in fact already on the chosen list, then, it returns to the list of six cities.

After defining the secondary function, we saved the indexes of these cities in a new variable and started the process for the swap of these six cities. Inside a for loop, for every even index, we would save it as the same even index but counting from the right to the left. For example, the city of index 0 will swap with the city of index 5, the city of index 2 will swap

with city of index 3 and the city of index 4 will swap with city of index 1. After having this clear, the swap was very similar to the one made before in the Hill Climber to the neighbours, with the help of the auxiliary variable.

```
def fit(self):
    # generate random solution
    route = [key for key in data.keys()]
    random.shuffle(route)
    s = Solution(route)
    self.solutions.append(s)

    for i in range(self.iterations):
        # run local search algorithm
        self.algorithm.fit(s.route)
        # takes the best solution founded
        s = self.algorithm.solutions[-1]
        self.solutions.append(s)
        # apply the perturbation
        s = self.perturbation(s)

    # sort by fitness descencing mode (minimization problem)
    self.solutions = sorted(self.solutions, key=lambda x: x.fitness, reverse=True)
    print('Finished Training')
```

Code 13: Fitness Function

To **finish the ILS algorithm**, we created the *(fit)* function to be called as (*def fit()*) to perform all the previous functions defined before. First, we start by saving the route using the keys from the main data (it is a dictionary, so the keys are the name of the cities), then we shuffle the route to have a completely random route and not in the order that the current main data is, and we saved the current route as a Solution and append it to the list of solutions. After that, until the number of iterations is achieved (this being defined by the user previously), **we will run the local search algorithm,** take the best solution found in that algorithm, append it to the list of solutions and then apply the perturbation function. Finally, after having all the solutions, we sort them by fitness in a descending mode, and since it is a minimization problem, we wanted the lowest fitness on top, and we print a note to say that the algorithm has already run the solution.

2.1.5. Execution

To have a better idea of the results we will show them and do a small exploration of the results and the functions we used to get them.

```
ils = ILS(algorithm=StochaticHillClimber(patience_threshold=10), iterations=1000)
ils.fit()
```

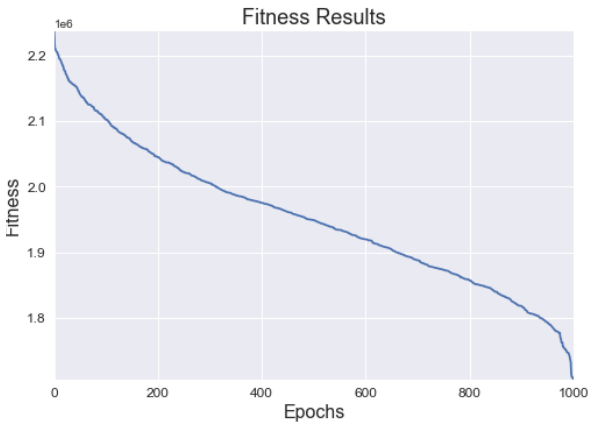
```
Finished Training
```

Code 14: Results Comparison (1000 interactions)

After some experimentations, **we decided to choose 1000 iterations to have a big base of comparison and results**. Although it takes time to get the result and its fitnesses, we thought it was a great point in the ratio of time to process good results. The same can be said about the number for the patience threshold since some trials made us believe that 10 was a great value to stay in. As an algorithm, we selected the Stochastic Hill Climber, as it was the one we worked on and prepared it.

```
------------
Best Result (lower fitness):
Route: [449, 1062, 206, 256, 81, 52, 935, 778, 518, 818, 786, 979, 824, 267, 271, 11, 1262, 275, 364, 776, 910, 643, 621, 19
6, 1146, 277, 891, 189, 461, 569, 989, 158, 625, 813, 141, 127, 933, 704, 882, 1229, 295, 91, 783, 866, 667, 138, 721, 779,
1155, 1095, 713, 166, 55, 77, 1148, 1157, 237, 120, 501, 360, 853, 880, 777, 186, 865, 1039, 469, 41, 747, 312, 300, 514, 71
2, 583, 752, 554, 750, 737, 88, 154, 1075, 1154, 382, 8, 290, 403, 304, 54, 192, 409, 1285, 216, 900, 1050, 288, 329, 422, 7
57, 912, 623, 914, 708, 994, 806, 398, 1272, 357, 1135, 957, 913, 894, 1215, 321, 1176, 1118, 619, 1017, 575, 107, 218, 976,
530, 356, 1266, 1284, 1187, 1260, 728, 666, 1042, 1034, 1045, 632, 618, 391, 897, 506, 1032, 160, 169, 952, 782, 707, 346, 7
24, 1287, 1282, 558, 92, 748, 450, 550, 680, 674, 522, 324, 281, 875, 698, 1212, 906, 323, 789, 1100, 660, 705, 586, 949, 82
7, 715, 214, 1041, 1153, 4, 429, 1063, 1029, 1083, 579, 115, 201, 798, 1053, 936, 1048, 814, 812, 887, 854, 598, 870, 1130,
1144, 1069, 801, 1283, 448, 874, 754, 955, 1106, 876, 1072, 689, 1021, 164, 960, 850, 279, 416, 182, 153, 66, 95, 555, 524,
424, 325, 872, 286, 1019, 307, 285, 546, 1090, 927, 1070, 560, 489, 867, 1085, 1013, 884, 399, 634, 1222, 1207, 830, 829, 92
1, 264, 1152, 1016, 345, 253, 1123, 225, 1091, 1167, 167, 940, 209, 200, 163, 1003, 1172, 1158, 220, 137, 505, 59, 342, 124
2, 1046, 947, 427, 211, 841, 654, 677, 774, 563, 926, 975, 581, 146, 998, 280, 1219, 741, 468, 603, 1203, 252, 670, 781, 54
8, 1248, 349, 1286, 36, 79, 394, 599, 890, 956, 816, 773, 497, 276, 612, 49, 508, 380, 1180, 901, 1231, 1232, 454, 1018, 91
5, 455, 296, 611, 790, 251, 982, 123, 964, 905, 860, 920, 521, 69, 641, 359, 199, 170, 791, 1269, 361, 941, 552, 1201, 205,
247, 843, 410, 405, 1261, 482, 526, 184, 292, 1265, 1196, 255, 1116, 185, 208, 460, 319, 1056, 172, 40, 6, 10, 499, 787, 64
8, 810, 751, 474, 1025, 1124, 697, 3, 442, 595, 585, 1131, 445, 996, 1178, 207, 202, 1150, 1030, 265, 291, 9, 580, 447, 108
7, 899, 1127, 213, 262, 1071, 1223, 1216, 883, 1225, 1259, 20, 531, 370, 7, 1290, 343, 839, 999, 268, 302, 378, 1182, 1082,
568, 510, 809, 1276, 1253, 846, 234, 733, 685, 542, 198, 686, 61, 953, 616, 574, 294, 1241, 446, 1084, 902, 928, 929, 1170,
1104, 851, 963, 1183, 1022, 681, 609, 244, 64, 476, 556, 540, 76, 148, 456, 396, 736, 723, 633, 863, 1119, 1163, 862, 1224,
838, 596, 978, 977, 1151, 162, 709, 665, 578, 925, 371, 1079, 113, 438, 577, 615, 1281, 339, 770, 602, 1147, 197, 126, 743,
1227, 327, 1236, 1230, 12, 298, 260, 588, 457, 527, 483, 390, 28, 513, 1233, 722, 340, 413, 533, 100, 562, 136, 191, 195, 46
4, 1251, 1139, 1179, 981, 212, 1210, 1111, 1169, 1086, 85, 885, 335, 1228, 1208, 1015, 328, 62, 87, 799, 408, 15, 980, 1089,
662, 74, 1094, 1288, 756, 760, 511, 570, 986, 729, 1120, 215, 134, 937, 690, 1114, 1192, 139, 744, 719, 699, 758, 14, 944, 3
89, 270, 549, 582, 749, 717, 766, 363, 1007, 561, 47, 60, 1121, 594, 99, 896, 1006, 143, 142, 204, 82, 147, 649, 942, 224, 2
99, 1149, 385, 909, 889, 1174, 958, 130, 726, 1257, 516, 362, 125, 673, 1243, 414, 498, 590, 877, 631, 71, 18, 374, 802, 10
8, 33, 532, 35, 1245, 490, 1031, 1040, 479, 434, 453, 630, 93, 515, 315, 444, 525, 1140, 727, 859, 122, 16, 331, 545, 90, 5
7, 1105, 836, 1247, 402, 105, 233, 46, 124, 26, 604, 34, 589, 376, 242, 970, 1037, 210, 283, 938, 1010, 1047, 322, 332, 775,
917, 868, 1068, 1204, 557, 696, 755, 541, 104, 44, 500, 512, 1028, 993, 825, 334, 544, 1240, 931, 140, 24, 38, 114, 68, 306,
70, 567, 706, 502, 228, 1185, 1027, 1054, 1133, 1184, 840, 1099, 764, 954, 759, 1067, 1066, 309, 592, 661, 150, 1044, 805, 1
195, 337, 492, 320, 833, 351, 610, 119, 605, 109, 844, 572, 656, 832, 78, 640, 236, 1080, 659, 246, 171, 369, 711, 520, 547,
1020, 725, 352, 491, 1065, 1175, 904, 397, 1264, 1267, 908, 716, 1023, 353, 39, 470, 392, 156, 762, 676, 1108, 1103, 1001, 1
12, 878, 266, 535, 73, 683, 792, 496, 617, 1275, 1036, 538, 308, 287, 672, 311, 1250, 797, 1, 487, 1268, 175, 1186, 131, 107
4, 553, 657, 916, 235, 1096, 106, 1078, 968, 528, 852, 845, 804, 529, 56, 537, 254, 133, 988, 684, 622, 393, 1218, 1033, 119
4, 419, 682, 29, 42, 377, 1235, 819, 240, 1165, 1092, 959, 922, 1134, 1205, 637, 664, 624, 366, 2, 1168, 1202, 395, 576, 96
2, 987, 695, 753, 373, 433, 365, 881, 1093, 679, 796, 620, 155, 945, 1128, 495, 63, 303, 1279, 772, 879, 458, 822, 1081, 67
8, 627, 179, 250, 30, 152, 418, 261, 504, 431, 652, 985, 990, 258, 893, 477, 428, 768, 157, 229, 565, 742, 420, 493, 735, 11
88, 1101, 1012, 481, 573, 523, 607, 871, 1214, 559, 354, 837, 1136, 1198, 785, 305, 187, 1097, 991, 869, 384, 795, 821, 101,
1278, 669, 423, 472, 226, 1213, 96, 415, 400, 13, 32, 1254, 701, 245, 181, 178, 100, 203, 732, 714, 857, 835, 1009, 919, 23,
471, 1159, 193, 435, 1011, 1141, 257, 121, 855, 417, 1189, 43, 961, 973, 1160, 1107, 888, 404, 129, 86, 1234, 338, 984, 272,
614, 808, 1162, 318, 274, 441, 53, 67, 536, 221, 873, 227, 1173, 430, 426, 440, 918, 98, 168, 693, 1026, 475, 864, 425, 956,
1117, 1211, 1043, 1137, 923, 383, 635, 807, 1237, 647, 232, 1166, 995, 950, 646, 1057, 1280, 1244, 350, 128, 97, 19, 80, 31,
645, 593, 551, 1220, 462, 102, 273, 465, 188, 132, 1035, 1064, 5, 282, 467, 1122, 1055, 1051, 600, 1190, 1058, 1270, 626, 67
1, 861, 1077, 608, 89, 946, 1008, 519, 1199, 834, 341, 731, 638, 636, 1191, 347, 238, 886, 1255, 1112, 765, 691, 811, 194, 9
83, 72, 826, 1005, 1098, 243, 159, 1132, 858, 27, 484, 289, 284, 924, 738, 1059, 480, 597, 355, 317, 375, 432, 118, 746, 104
9, 316, 793, 1004, 1052, 466, 263, 997, 566, 144, 943, 967, 734, 815, 720, 658, 539, 606, 451, 769, 1209, 692, 103, 22, 117,
367, 668, 75, 459, 1256, 401, 348, 951, 1060, 223, 1156, 222, 831, 571, 584, 110, 65, 116, 1129, 1061, 803, 411, 45, 974, 68
7, 1110, 387, 50, 173, 1142, 333, 388, 177, 37, 111, 84, 381, 1143, 443, 517, 965, 1109, 823, 694, 543, 1252, 1277, 58, 165,
230, 800, 145, 239, 1138, 1289, 488, 820, 1102, 259, 1076, 248, 1088, 407, 439, 486, 842, 784, 939, 1000, 930, 688, 847, 17,
25, 217, 1239, 452, 473, 21, 1193, 1171, 1164, 948, 892, 903, 1038, 313, 740, 219, 249, 135, 161, 1125, 745, 503, 507, 1206,
1217, 895, 650, 485, 703, 655, 0, 269, 174, 379, 788, 1226, 767, 761, 601, 326, 1271, 856, 971, 1161, 1197, 330, 421, 849, 1
126, 1014, 628, 629, 1258, 478, 463, 1249, 372, 780, 730, 911, 663, 1024, 700, 613, 509, 344, 1238, 494, 297, 1263, 932, 96
9, 1113, 1246, 94, 564, 436, 644, 1073, 710, 534, 48, 412, 907, 639, 310, 1273, 1221, 1200, 848, 591, 293, 651, 718, 83, 76
3, 1274, 406, 51, 183, 1145, 176, 190, 702, 368, 794, 972, 278, 386, 817, 992, 642, 934, 1115, 358, 437, 151, 771, 336, 675,
1181, 1002, 828, 301, 314, 1177, 653, 898, 241, 149, 587, 739, 231]   |   Fitness: 1706326
```

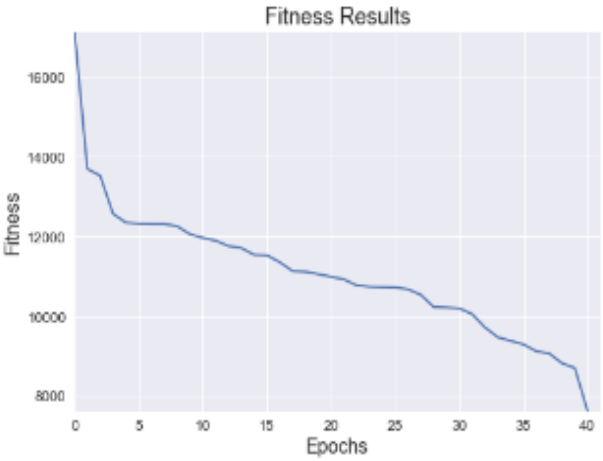
Code 16 Interactions Algorithm Route



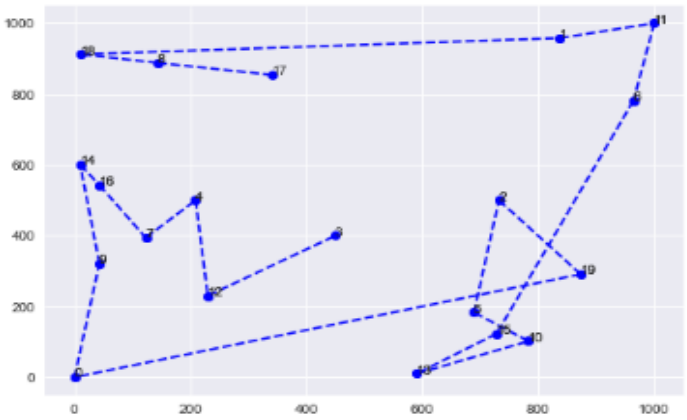
Code 15: Evolution of the fitness per epoch (1000 interactions)

Using the main data to get the results, we noticed that the decrease of the fitness is not constant along the epochs. It starts decreasing rapidly in the beginning and slows downs the decrease until after the 900º epoch. After that the decrease of the fitness happens quickly again. To summarize, we started with an initial fitness of 2236994 and after 1000 iterations we ended up with fitness of 1706326. On the left, we can see the route made by the algorithm in the 999º iteration has the best fitness of all, this was created by using the function defined previous “show_best_result”. And on the right, created with “plot_fitness”, that shows the evolution of the fitness per epoch, showing a curvy line, accentuated at the beginning and the end (or as said a faster decrease in the fitness).

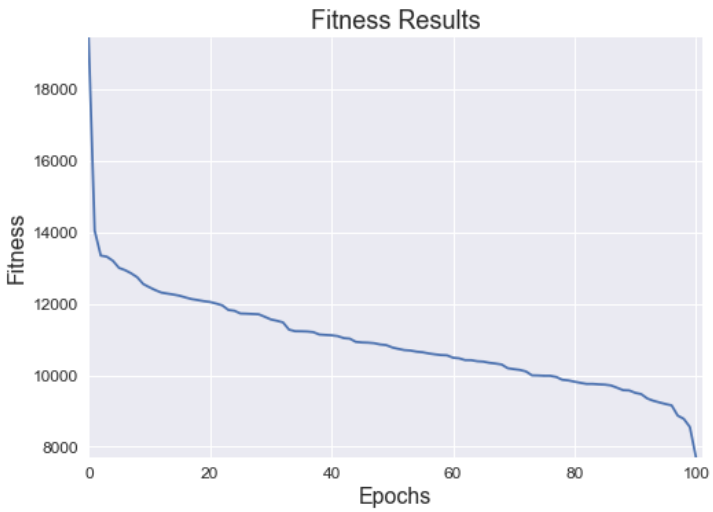
We wanted to analyse the behaviour of the algorithm from close, so we created the test dataset, both to test the algorithm and to better visualize some functions, that with the main data were not so clear.



Code 18: Evolution of the fitness per epoch (40 interactions, test-data)



Code 17: Best Visualization Result of the Route (test-data)



Code 19: Evolution of the fitness per epoch (100 interactions)

After some testing and experimenting with the test data (and the main data), we noticed that the shape of the plot doesn't change much, independently of how many iterations we run (epochs). The first iterations decrease a lot of the fitness (until close to the median distribution of the evolution of the fitness). Then until the finals iterations, the fitness decreases very slowly. In the final iterations, the fitness starts to decrease a lot again, until the end of the iterations. We have an example of the test data with 40 iterations and 100 iterations to show our point.

2.2. GA Algorithm

2.2.1. Debug and Plot Functions

```
def show_best_result(solutions: list):
    print(f"-------------\nBest Result (lower fitness):\nFitness: {solutions[-1].fitness}")
    print(f"Route: {solutions[-1].chromossome}")
```

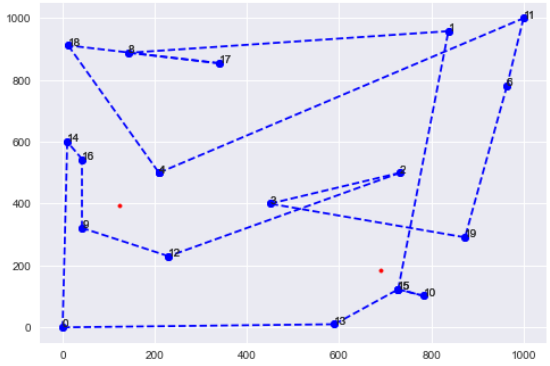
Code 20: The Best Result from Set of Solutions

To apply the GA algorithm, we started by creating some functions to plot and visualize the results. Between these functions we have a function that shows the best result. This function contains the route with the lowest fitness, and returns the route itself, to understand which is the best route.

```
def plot_connections(data: list,
                     route: list):
    plt.figure()
    plt.scatter(*zip(*data), color='red', s=12)

    for i in range(1, len(route)):
        x1, y1 = data[route[i - 1]]
        x2, y2 = data[route[i]]
        x_values = [x1, x2]
        y_values = [y1, y2]
        plt.plot(x_values, y_values, 'bo', linestyle="--")
        plt.text(x1 - 0.50, y1 + 0.50, route[i - 1])
        plt.text(x2 - 0.50, y2 + 0.50, route[i])

    plt.show()
```

Code 21: Data Preparation for the Routes Visualization

Then we defined another function that plotted the coordinates into a scatter plot and connected all the dotes according to the best route, like what was done in the ILS algorithm defined previously. For this function we received the data itself and the route that we wanted to visualize. Next, after defining a scatter plot of the coordinates of each city (and with the correspondent's name), we did a for loop to plot a "--" line between a point and the previous point, using the order given in the route list. The results of this function are better visualized when used with test data since it has fewer cities and therefore it is more perceptible.



Code 22: Best Visualization Result of the Route

```python
def plot_fitness(solutions):
    plt.style.use('seaborn')

    fitnesses = [sol.fitness for sol in solutions]

    # general figure options
    plt.figure(figsize=(15, 7))
    ax = plt.axes(xlim=(0, len(fitnesses)), ylim=(np.min(fitnesses) - 1, np.max(fitnesses) + 1))
    _, = ax.plot([], [], lw=2)
    ax.set_title('GA results', fontsize=18)
    ax.set_xlabel('generations', fontsize=16)
    ax.set_ylabel('Fitness', fontsize=16)
    ax.tick_params(labelsize=12)

    generations = [i for i in range(len(fitnesses))]

    plt.plot(generations, fitnesses)

    plt.show()
```

Code 23: Fitness Evolution Plot along the Generations

Our final plot function was a simple function to plot all the fitness along with the different generations (defined later in the code). To do this we saved the fitness of the solutions in a list and defined the characteristics of our plot (size, type, title and labels). We also defined the length of both x-axes with the size of the list created previously and regarding the y-axes, from the minimum and maximum fitness with 1 unit of space to space out from the border. After that, we define the generations. Since each fitness in the list was generated from a set of generations, we only used the function range to attribute the first fitness to generation 0, the second fitness to generation 1 and so on, until the last fitness in the list.

2.2.2. Individuals

```python
class Individual:
    def __init__(self, chromossome):
        self.chromossome = chromossome
        self.fitness = self.__get_fitness()

    def __manhattan(self, p1, p2):
        return int(math.fabs(p1[0] - p2[0]) + math.fabs(p1[1] - p2[1]))

    # takes as an argument a distance metric
    def __get_fitness(self):
        fitness = 0
        for i in range(len(self.chromossome) - 1):
            fitness += self.__manhattan(data[self.chromossome[i]], data[self.chromossome[i + 1]])

        return fitness
```

Code 24: Distance between the cities, their final fitness of the routes

In this part we need to define an individual from the population, giving a way of having a chromosome (to continue the genetic algorithm), a way to get fitness and a way to calculate the distance between two points or cities. For the chromosome, everyone can receive one when it receives a list, and then saves the correspondent key value from the dictionary and uses that to calculate the fitness. But, for this calculation, we firstly defined a function to calculate the **distance between two points using the Manhattan Distance Metric** (the sum of absolute subtraction of each x [with index 0] and y [with index 0] and x and y of index 1. After having a way of calculating the distance we defined a way of getting the fitness using the chromosomes and the distance between one chromosome of index i and the next chromosome of index i+1. We add all these results with a for loop until we used all the chromosomes of a specific individual.

2.2.3. The Genetic Algorithm

```
class GA:
    def __init__(self, generations, population_size, elitism):
        self.generations = generations
        self.pop_size = population_size
        self.elitism = elitism
        self.solutions = list()
        self.population = list()
```

Code 25: Defining the Genetic Algorithm (GA)

After defining all the necessary functions, we could start defining the Genetic Algorithm. We define that, when called, the GA must have or receives the number of generations that wanted to perform to improve the fitness, a number for the population size of selected parents and the elitism number. This, with the purpose of diminishing the potential flaw of missing or losing the best individuals in the population, we saved x individuals for the next generation (being x the number defined). After we save the information, we got for the Genetic Algorithm and created two empty lists, one for the solutions and another for the population.

```
    def create_population(self):
        for i in range(self.pop_size):
            chromossome = [key for key in data.keys()]
            # shuffle it to make it random
            random.shuffle(chromossome)
            # copy the first element so the route list starts and ends on the same element
            chromossome.append(chromossome[0])

            indiv = Individual(chromossome)

            self.population.append(indiv)

        self.population = sorted(self.population, key=lambda x: x.fitness)
```

Code 26: Information, Fitness of Chromosomes

After receiving the necessary information for the GA, we defined a function to create the population with the size defined in the previous step. Then, **we shuffle each chromosome randomly** to improve the results and get a higher diversity. To start and end in the same element, we simply copy the element of index 0 and append this copy in the final of the chromosome. In each chromosome, we classified as an individual to then, being able to get information about distance and fitness. Finally, we added these new individuals (the chromosomes with the characteristics of the Individuals defined in 2.2.1.) and had the full list of the population with the different chromosomes as elements, that we sorted accordingly to their fitness (the highest the fitness the first).

```
    # selecting a parent
    def tournament(self):
        k = 2
        candidates = random.sample(self.population, k=k)  # the same individual can be chosen multiple times
        candidates = sorted(candidates, key=lambda x: x.fitness)
        return candidates[0]
```

Code 27: Parent Selection Method

Finally arriving at a different step from the Genetic Algorithm given in class, as method of parent selection we had to use the tournament with k=2. For this we defined a function called tournament, saved the information of k=2 and selected the k random candidates from the population we created in the previous step. It is important to notice that in this selection

the same individual of the population can be chosen twice. After having the candidates, we sort them by fitness again, and return the one with lowest fitness.

```
def mutation(self, parent):
    # we can make a while to ensure both the start and end of the route are not changed
    while True:
        new_chromossome = deepcopy(parent.chromossome)

        p1 = random.choice(new_chromossome)
        p2 = random.choice(new_chromossome)

        id1 = new_chromossome.index(p1)
        id2 = new_chromossome.index(p2)

        if id1 == 0 or id1 == len(new_chromossome) or id2 == 0 or id2 == len(new_chromossome):
            pass
            # if the swap ids contain either the start or end points of the route, we repeat
        else:
            break
            # if they do not, we can accept those ids, exiting the while loop and creating our new route
    new_chromossome[id1] = p2
    new_chromossome[id2] = p1

    offspring = Individual(new_chromossome)

    return offspring
```

Code 28: Mutation Performance for Change the position of two random cities

After we did the first list of parents, we performed a mutation to change the position of two random cities between each other. Inside a while loop, first we selected a random city from the population and saved as parent1 and then another one as parent2 and we saved the index of each one before changing them. To leave the while loop we only accept if the cities selected were neither the first nor the last. If so, we would repeat the selection until they are neither the last nor the first. After leaving the while loop, we had two cities, so we simply put the second city in the index of the first and the first one in the index of the second. Then we end up with the offspring of this mutation and return this individual.

```
def crossover(self, parent1: Individual, parent2: Individual):
    subset = random.sample([i for i in range(len(parent1.chromossome))],
                           k=random.randint(1, len(parent1.chromossome)))
    new_chromossome = [None] * len(parent1.chromossome)

    for index in subset:
        new_chromossome[index] = parent1.chromossome[index]

    for index in range(len(new_chromossome)):
        if new_chromossome[index] is None:
            new_chromossome[index] = parent2.chromossome[index]

    offspring = Individual(new_chromossome)

    return offspring
```

Code 29: Crossover Method

As the method of crossover, we went with the order crossover as asked by the teachers. This crossover consists of saving a subset of a chromosome and using it in the same index for the offspring and filling the rest of the index with the values of the parent 2. With this code selection, the random subset of the chromosome of the parent1 with random size save them into a variable. We created a new chromosome with the same length as the parent1 where this new chromosome's index corresponds to the ones present in the subset. We filled with those values and for the indexes that are not present in the subset, we fill with the values parent2. Lastly we classified this new chromosome as an individual and return this offspring.

```python
def apply_operators(self, parents):  # apply mutation, crossover and elitism

    if len(set(parents)) == 1:
        print(
            "The following loop (line 190) is going to be infinite because all the parents are equal: "
            "\nwhile p2 == p1:"
            "\n\tp2 = random.choice(parents)"
            "\n\nFinishing program...")
        exit()

    new_population = []

    while len(new_population) < self.pop_size:
        # generates a random number [0,100]
        prob = random.random() * 100
        if prob < 30:  # mutation
            p = random.choice(parents)
            p = self.mutation(p)
            new_population.append(p)
        else:  # crossover
            p1 = random.choice(parents)
            p2 = random.choice(parents)

            while p2 == p1:
                p2 = random.choice(parents)

            for i in range(2):  # we generate 2 offsprings
                son = self.crossover(p1, p2)
                new_population.append(son)

    # apply elitism
    # first remove the bad solutions
    new_population = sorted(new_population, key=lambda x: x.fitness)
    for i in range(self.elitism):
        del new_population[-1]

    # then add the old top solutions
    for i in range(self.elitism):
        new_population.append(self.population[0])

    # sort it again
    self.population = sorted(new_population, key=lambda x: x.fitness)
```

Code 30: Applying (mutation, crossover, elitism) GA Algorithm

After having the selection method specified, the mutation and the crossover we needed to apply them to a new population. In first place, we started by saying that if the number of parents is only one, it would not work. To not create a new empty list for the new population and start filling the values of this list until it has the same size as the initial population, we needed to make a notice about it.

Since we do not want both methods to happen at the same time, we defined a probability of each one occurring, choosing a random number from 0 to 1 and multiplying by 100. For the mutation, it had 30% chance of happening, and if it did, we would choose one random parent from the set of parents, apply the mutation to it and then appended the result of the mutation to the new population.

On the other hand, the crossover had 70% chance of happening, and if so, it selected two random parents from the set of parents, to make sure they were different. We did a while loop that only closed when parent1 and parent2 were different, in each time they were equal it would randomly choose another parent as parent2. After having two distinct parents, we applied the crossover twice to get two offspring and then append them to the new population (separately).

To apply the elitism, we first sorted them in descending order accordingly to their fitness, and for a range of x numbers (defined previously as the number for elitism), we deleted the last x individuals from the new population list, since they had the worst fitness. Next, we added the best solution ten times to the new population and to finish we ordered the new population again, from the highest to the smallest.

```
def fit(self):
    for i in range(self.generations):
        if i == 0:
            self.create_population()
            print(
                f"Best individual at start --> fitness: {self.population[0].fitness}")
            print(f"route: {self.population[0].chromossome}")

        parents = list()

        while len(parents) < self.pop_size:
            parents.append(self.tournament())

        self.apply_operators(parents)
        self.solutions.append(self.population[0])

        print(f"'Generation {i} | fitness: {self.population[0].fitness}")
        print(f"route: {self.population[0].chromossome}")

    print("Done!")
```

Code 31: Fitness of each Generation (GA)

To finish the algorithm, we defined the last function to perform the GA itself using all the functions defined before, to return the fitness of each generation and then to return the best one at last. On the code, we put everything inside a loop with the number of generations as the number of iterations. For the first generation, we decided to print the fitness of the population to know the initial fitness, where we started and the route of cities. Secondly, we created an empty list for the parents, and until this list has the size of the pop_size (argument defined in the initialization of the function "GA"), we append individuals using the tournament method. Thirdly we apply the mutation, crossover and elitism using the function in the previous block (*apply_opperators*) and return the first element of the population (or new population after going inside the apply operators), that is the element(route) with the highest or the best fitness. Finally, we printed the generation that we were in, this route fitness has and the route itself. To complete, we left the for loop and we made a not to say the algorithm ended. As said, all this code will be repeated until we get x number of generations, being x, the number of generations defined in the arguments of the function "GA".

2.2.4. Execution

An important note to the execution of the algorithm we have a possibility to experiment the algorithm with a sample of the main data that we called test data (composed of 20 cities). Some functions, as the scatter plot with the best route, are better shown in using the test data. So, to show our conclusions we will talk about the main data, to have a better visualization and understanding that we thought it was the one which would have a better performance.

```
ga=GA(generations=1000, population_size=100, elitism=10)  # parameters for main_data
ga.fit()
```

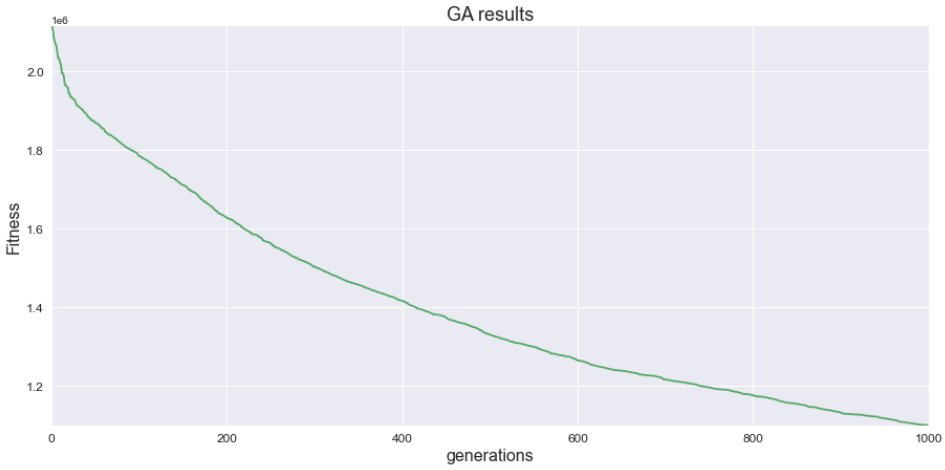
Code 32: Applying GA [1000 generations, 10 elitism, 100 population size]

After some tries and experimentations, we ended up with a total number of 1000 generations. Although it takes some time to get the result and its fitnesses, we believed that it is the best result in a ratio of time with good results. The same can be said about the number for elitism. After deciding that a maximum of 10% of the population size was the way to go in order not to lose diversity, we performed some trials until 10 which led us to believe that 10 was the best result. For the population size, since the main data counts with 1291 cities, each route or chromosome is composed of 1291 elements, so we used the maximum population size, one should use, 100.

```
Best Result (lower fitness):
Fitness: 1099338
Route: [536, 506, 589, 653, 613, 30, 59, 48, 112, 792, 758, 666, 901, 819, 729, 685, 640, 787, 688, 1238, 449, 1227, 1237, 3
59, 1232, 429, 470, 1007, 975, 844, 416, 497, 495, 1276, 1265, 497, 1231, 578, 689, 828, 841, 870, 805, 689, 494, 602, 907,
1215, 755, 594, 71, 102, 51, 55, 422, 837, 910, 910, 1116, 1067, 1014, 902, 449, 395, 473, 448, 476, 933, 989, 969, 974, 36
7, 283, 830, 150, 860, 882, 686, 707, 738, 657, 272, 302, 273, 433, 459, 738, 914, 832, 833, 965, 915, 838, 792, 598, 444, 4
55, 504, 588, 962, 983, 976, 538, 609, 44, 50, 84, 505, 870, 891, 975, 1123, 837, 1188, 925, 983, 1099, 147, 880, 1092, 116
5, 1122, 964, 132, 5, 319, 412, 72, 54, 20, 106, 120, 148, 845, 829, 739, 581, 974, 1028, 1282, 1229, 317, 361, 1249, 722, 6
81, 858, 887, 613, 1037, 228, 1159, 1183, 1037, 1126, 875, 300, 371, 497, 1090, 925, 1045, 1032, 1040, 1078, 1038, 854, 718,
628, 553, 357, 1256, 654, 752, 649, 879, 995, 994, 1111, 495, 852, 833, 1132, 1048, 1068, 937, 937, 151, 117, 191, 190, 77,
106, 387, 1246, 512, 649, 966, 978, 1000, 196, 144, 844, 898, 1081, 1053, 960, 827, 1125, 527, 570, 704, 174, 176, 1008, 83
7, 295, 285, 329, 600, 608, 443, 391, 1038, 117, 847, 974, 973, 693, 689, 583, 52, 101, 844, 1073, 165, 182, 1004, 1039, 100
3, 940, 917, 148, 847, 390, 395, 358, 711, 1164, 1142, 578, 421, 272, 801, 993, 1111, 872, 458, 420, 290, 274, 437, 826, 53
4, 589, 589, 476, 805, 153, 154, 647, 450, 477, 922, 986, 837, 362, 283, 457, 445, 306, 656, 435, 498, 1131, 1129, 1045, 97
5, 994, 792, 739, 568, 469, 872, 1106, 983, 1071, 451, 355, 354, 662, 646, 579, 689, 546, 644, 837, 623, 707, 840, 794, 975,
922, 285, 318, 1229, 512, 736, 645, 877, 966, 938, 278, 487, 1130, 1131, 1054, 130, 504, 605, 782, 459, 503, 411, 744, 106,
110, 171, 164, 121, 380, 390, 381, 438, 392, 466, 828, 1161, 891, 936, 438, 925, 38, 38, 453, 482, 496, 758, 927, 1127, 121
2, 1189, 811, 1139, 1211, 677, 1167, 1134, 969, 181, 653, 503, 527, 585, 611, 67, 585, 798, 258, 1173, 1129, 1051, 1153, 103
7, 965, 1180, 238, 184, 1102, 1224, 1163, 1158, 977, 165, 175, 1054, 1036, 1040, 416, 296, 340, 1197, 1217, 929, 858, 816, 1
013, 928, 338, 1290, 1245, 574, 806, 1071, 1220, 1222, 690, 552, 434, 697, 873, 920, 904, 1208, 1194, 1046, 1063, 1184, 113
2, 847, 20, 194, 1001, 755, 604, 483, 390, 45, 470, 365, 310, 388, 5, 852, 868, 549, 685, 780, 780, 594, 696, 57, 60, 111, 1
075, 1102, 986, 983, 873, 851, 192, 980, 617, 690, 1177, 1060, 54, 480, 542, 665, 84, 508, 593, 90, 511, 469, 427, 630, 678,
646, 741, 184, 899, 273, 1079, 1041, 1001, 143, 127, 996, 1009, 1072, 870, 850, 834, 854, 696, 1027, 865, 507, 414, 1276, 46
4, 108, 441, 651, 732, 625, 671, 786, 994, 840, 718, 991, 1117, 1008, 602, 578, 661, 647, 915, 731, 712, 482, 870, 599, 472,
482, 1008, 922, 1180, 1097, 605, 526, 703, 795, 946, 208, 219, 185, 802, 703, 501, 416, 471, 647, 690, 585, 352, 347, 495, 1
229, 750, 741, 938, 406, 524, 583, 437, 539, 900, 891, 987, 1118, 1021, 905, 327, 346, 416, 804, 1075, 998, 578, 404, 265, 9
76, 977, 608, 977, 966, 918, 532, 100, 100, 591, 21, 67, 61, 155, 919, 922, 1033, 976, 1096, 1060, 227, 253, 813, 776, 741,
890, 924, 782, 734, 935, 870, 795, 871, 783, 756, 623, 782, 674, 673, 354, 118, 106, 612, 21, 272, 388, 148, 216, 1213, 700,
388, 123, 127, 844, 600, 553, 471, 474, 336, 352, 1229, 413, 404, 493, 354, 352, 578, 805, 1172, 237, 208, 871, 153, 885, 80
8, 812, 717, 772, 660, 710, 691, 93, 51, 132, 652, 945, 586, 449, 833, 851, 1012, 927, 1220, 1148, 1128, 979, 192, 156, 759,
812, 791, 575, 664, 581, 958, 1063, 250, 847, 43, 578, 832, 362, 168, 911, 761, 685, 687, 619, 924, 884, 785, 819, 1054, 84
5, 840, 604, 829, 1152, 157, 761, 679, 476, 706, 134, 213, 819, 560, 571, 1134, 462, 68, 741, 661, 970, 973, 1166, 1124, 15
5, 699, 1042, 912, 1273, 673, 628, 339, 728, 567, 273, 260, 403, 410, 151, 119, 645, 499, 48, 740, 834, 750, 131, 117, 111,
805, 1036, 531, 545, 1237, 265, 442, 485, 32, 35, 416, 1238, 739, 407, 50, 813, 335, 1248, 266, 406, 16, 1, 1213, 697, 458,
233, 172, 751, 762, 789, 544, 431, 19, 50, 139, 97, 819, 689, 805, 139, 195, 946, 960, 1031, 1164, 1101, 792, 837, 804, 699,
1131, 1062, 604, 976, 412, 85, 135, 847, 178, 106, 269, 365, 364, 182, 187, 1108, 1186, 923, 966, 921, 1092, 794, 526, 757,
498, 405, 374, 14, 457, 939, 1102, 844, 849, 922, 279, 26, 45, 91, 110, 269, 342, 1258, 503, 944, 227, 55, 261, 395, 849, 12
00, 1048, 1072, 648, 905, 770, 884, 807, 185, 124, 609, 804, 990, 843, 113, 155, 950, 734, 520, 856, 868, 809, 834, 737, 73
7, 1075, 207, 186, 760, 669, 673, 1189, 793, 78, 138, 869, 833, 823, 431, 435, 1054, 925, 783, 989, 905, 888, 834, 705, 151,
482, 413, 673, 279, 368, 369, 368, 400, 523, 983, 594, 616, 578, 343, 294, 486, 197, 964, 408, 458, 416, 440, 395, 356, 855,
870, 1094, 1168, 1083, 469, 492, 1284, 448, 747, 184, 183, 985, 323, 287, 277, 1229, 418, 805, 718, 524, 707, 420, 938, 390,
455, 1191, 762, 692, 582, 473, 401, 498, 704, 30, 544, 639, 722, 750, 703, 811, 1124, 564, 656, 371, 973, 100, 124, 155, 128
3, 1219, 1188, 1135, 892, 181, 834, 728, 702, 61, 87, 109, 1127, 1133, 780, 14, 452, 736, 1076, 1123, 814, 334, 1274, 485, 7
99, 80, 121, 534, 873, 211, 96, 100, 235, 248, 996, 710, 955, 905, 181, 918, 602, 1055, 1002, 981, 993, 671, 1197, 356, 459,
782, 705, 1060, 593, 177, 99, 540, 122, 121, 159, 125, 745, 588, 583, 471, 480, 745, 905, 623, 951, 400, 581, 650, 751, 730,
328, 724, 742, 1006, 946, 487, 742, 873, 887, 926, 555, 819, 1094, 403, 793, 915, 311, 450, 967, 659, 647, 530, 406, 940, 98
4, 711, 364, 419, 1235, 646, 571, 750, 600, 522, 826, 691, 604, 999, 844, 844, 1148, 195, 261, 154, 843, 999, 190, 411, 593,
1159, 1063, 1017, 616, 582, 543, 571, 1009, 974, 776, 683, 93, 4, 25, 438, 383, 1261, 578, 384, 382, 594, 946, 982, 922, 84
1, 965, 938, 798, 946, 893, 652, 976, 951, 779, 887, 850, 782, 686, 428, 489, 1236, 275, 532, 739, 420, 409, 381, 342, 336,
911, 911, 1115, 784, 790, 512, 106, 176, 183, 614, 431, 368, 457, 91, 92, 163, 778, 672, 640, 486, 41, 263, 301, 946, 880, 5
43, 384, 989, 424, 517, 886, 847, 1036, 256, 980, 874, 140, 847, 263, 313, 89, 876, 1129, 861, 671, 24, 391, 567, 833, 969,
496, 1206, 1195, 754, 598, 66, 148, 881, 836, 788, 927, 904, 949, 571, 598, 481, 1167, 600, 429, 503, 735, 739, 518, 521, 12
55, 933, 869, 946, 570, 738, 1054, 1079, 658, 810, 515, 387, 1051, 1056, 155, 176, 853, 430, 311, 524, 880, 1002, 910, 790,
1264, 1257, 983, 382, 843, 975, 828, 948, 153, 154, 120, 797, 873, 1058, 801, 412, 204, 234, 1056, 1061, 1074, 1187, 932, 34
9, 578, 705, 605, 497, 648, 54, 412, 462, 1168, 919, 662, 699, 44, 214, 247, 805, 821, 429, 534, 395, 1062, 1160, 649, 412,
765, 751, 400, 547, 753, 536]
```

Code 33: Generations Algorithm Route

In the main data, we started with a fitness of 2122914 before the first generation and ended it with 109938 after 1000 generations. This route was obtained by using the function we created previously to show the best result or the generation with the lowest fitness. We can conclude, from each generation, that the difference in the route was very small and as referred previously we wanted to made sure that the first and the last cities were the same.



Code 34: Evolution of main data their fitness (1000 Generations) GA

After seeing the route, what was left to do was to see the evolution of the fitness along the multiple generations. As expected, it shows a harsh decrease in the beginning (around the first 20 generations) and then the decrease of the fitness happens very smoothly but continuously.

Example of the test data:

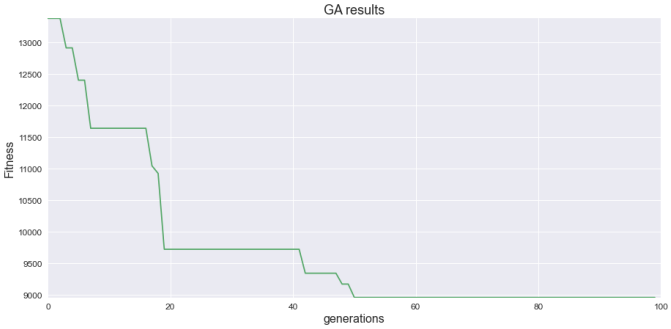
```
'Generation 0 | fitness: 13376
route: [6, 8, 12, 2, 3, 19, 16, 17, 18, 13, 5, 10, 15, 1, 9, 14, 0, 4, 7, 11, 6]

'Generation 1 | fitness: 13376
route: [6, 8, 12, 2, 3, 19, 16, 17, 18, 13, 5, 10, 15, 1, 9, 14, 0, 4, 7, 11, 6]

'Generation 2 | fitness: 13376
route: [6, 8, 12, 2, 3, 19, 16, 17, 18, 13, 5, 10, 15, 1, 9, 14, 0, 4, 7, 11, 6]

'Generation 3 | fitness: 12908
route: [6, 8, 12, 2, 3, 19, 16, 17, 0, 13, 5, 10, 15, 1, 9, 14, 18, 4, 7, 11, 6]

'Generation 4 | fitness: 12908
route: [6, 8, 12, 2, 3, 19, 16, 17, 0, 13, 5, 10, 15, 1, 9, 14, 18, 4, 7, 11, 6]

'Generation 5 | fitness: 12396
route: [6, 8, 12, 2, 3, 19, 16, 14, 0, 13, 5, 10, 15, 1, 9, 17, 18, 4, 7, 11, 6]

'Generation 6 | fitness: 12396
route: [6, 8, 12, 2, 3, 19, 16, 14, 0, 13, 5, 10, 15, 1, 9, 17, 18, 4, 7, 11, 6]

'Generation 7 | fitness: 11634
route: [6, 19, 12, 2, 3, 8, 16, 14, 0, 13, 5, 10, 15, 1, 9, 17, 18, 4, 7, 11, 6]

'Generation 8 | fitness: 11634
route: [6, 19, 12, 2, 3, 8, 16, 14, 0, 13, 5, 10, 15, 1, 9, 17, 18, 4, 7, 11, 6]

'Generation 9 | fitness: 11634
route: [6, 19, 12, 2, 3, 8, 16, 14, 0, 13, 5, 10, 15, 1, 9, 17, 18, 4, 7, 11, 6]

'Generation 10 | fitness: 11634
route: [6, 19, 12, 2, 3, 8, 16, 14, 0, 13, 5, 10, 15, 1, 9, 17, 18, 4, 7, 11, 6]

'Generation 11 | fitness: 11634
route: [6, 19, 12, 2, 3, 8, 16, 14, 0, 13, 5, 10, 15, 1, 9, 17, 18, 4, 7, 11, 6]

'Generation 12 | fitness: 11634
route: [6, 19, 12, 2, 3, 8, 16, 14, 0, 13, 5, 10, 15, 1, 9, 17, 18, 4, 7, 11, 6]

'Generation 13 | fitness: 11634
route: [6, 19, 12, 2, 3, 8, 16, 14, 0, 13, 5, 10, 15, 1, 9, 17, 18, 4, 7, 11, 6]

'Generation 14 | fitness: 11634
route: [6, 19, 12, 2, 3, 8, 16, 14, 0, 13, 5, 10, 15, 1, 9, 17, 18, 4, 7, 11, 6]

'Generation 15 | fitness: 11634
route: [6, 19, 12, 2, 3, 8, 16, 14, 0, 13, 5, 10, 15, 1, 9, 17, 18, 4, 7, 11, 6]

'Generation 16 | fitness: 11634
route: [6, 19, 12, 2, 3, 8, 16, 14, 0, 13, 5, 10, 15, 1, 9, 17, 18, 4, 7, 11, 6]
```

Code 35: GA Generations (test-data)

In order to see the best evolution, we have the fitness evolution per generation to the test data and had a clear and visible route.

We can see the decrease of fitness, until the point where the fitness establishes and doesn't decrease anymore.

As said before we can also notice a very subtle difference between routes of different generations. For example, between the fourth and the fifth generation, only two cities change places. This change is more noticeable when we have more data, but we in this we can have a good idea of the process between



Code 36: Test-data Fitness Representation GA

generations

In the plot of fitness, the stagnation of fitness is even more clear, and after 50 generations the fitness doesn't change anymore, until 100 generations.

3. Algorithms Comparative Analysis

For the comparison, we used mostly excel to have a very clear idea of each iteration and then in the end, the mean and standard deviation of each iteration, for each algorithm. Both algorithms perform 30 iterations of 100 generations/epochs, respectively for the GA and ILS.

Statistics:	Mean	Standar Deviation
Epoch 0	2214853	24214
Epoch 1	2183016	26450
Epoch 2	2167094	25952
Epoch 3	2156015	24182
Epoch 4	2147582	25713
Epoch 5	2133913	25892
Epoch 6	2124894	24097
Epoch 7	2113427	23838
Epoch 8	2107195	22266
Epoch 9	2097692	22364
Epoch 10	2088350	22529
Epoch 11	2081643	22486
Epoch 12	2076179	22211
Epoch 13	2071439	21472
Epoch 14	2065265	20616
Epoch 15	2061160	21365
Epoch 16	2056748	18784
Epoch 17	2053772	19069
Epoch 18	2049609	17599
Epoch 19	2046032	18242
Epoch 20	2040407	16690
Epoch 21	2035717	17181
Epoch 22	2031255	17497
Epoch 23	2028649	17177
Epoch 24	2024704	17321
Epoch 25	2020494	16650
Epoch 26	2016827	16300
Epoch 27	2014469	15879
Epoch 28	2009486	15672
Epoch 29	2006978	16350
Epoch 30	2003263	16341
Epoch 31	2000161	16008
Epoch 32	1997407	15402
Epoch 33	1993787	16116
Epoch 34	1990657	15531
Epoch 35	1987967	14937
Epoch 36	1983866	14674
Epoch 37	1980364	13296
Epoch 38	1977582	12332
Epoch 39	1974248	12012
Epoch 40	1970827	11490
Epoch 41	1968192	10859
Epoch 42	1965885	11007
Epoch 43	1963209	10452
Epoch 44	1959783	10167
Epoch 45	1957074	10035
Epoch 46	1954226	10841
Epoch 47	1951604	10431
Epoch 48	1948283	9915
Epoch 49	1945122	10604
Epoch 50	1941490	11281
Epoch 51	1938594	11625
Epoch 52	1936092	11527
Epoch 53	1934063	11621
Epoch 54	1930937	10963
Epoch 55	1928567	12094
Epoch 56	1925735	11840
Epoch 57	1923118	11905
Epoch 58	1920463	12195
Epoch 59	1918352	12109
Epoch 60	1915876	12287
Epoch 61	1912901	13161
Epoch 62	1909795	12906
Epoch 63	1907576	13166
Epoch 64	1904248	13252
Epoch 65	1902114	12911
Epoch 66	1899398	12922
Epoch 67	1897505	13238
Epoch 68	1895505	13236
Epoch 69	1892736	13508
Epoch 70	1888532	12900
Epoch 71	1885771	13044
Epoch 72	1882602	13706
Epoch 73	1878935	13790
Epoch 74	1877237	13674
Epoch 75	1874904	13849
Epoch 76	1872267	13679
Epoch 77	1869545	13405
Epoch 78	1866803	13487
Epoch 79	1863686	12638
Epoch 80	1860613	12431
Epoch 81	1857429	12884
Epoch 82	1854900	12363
Epoch 83	1851123	13566
Epoch 84	1847872	13760
Epoch 85	1845015	12984
Epoch 86	1841484	12958
Epoch 87	1838528	13495
Epoch 88	1834710	13283
Epoch 89	1829835	13557
Epoch 90	1825241	14215
Epoch 91	1820440	13697
Epoch 92	1815739	13299
Epoch 93	1809750	15780
Epoch 94	1804438	15570
Epoch 95	1797965	16750
Epoch 96	1790264	17186
Epoch 97	1783424	14892
Epoch 98	1774863	16567
Epoch 99	1758864	18565

Code 37: ILS Results

Statistics	Mean	Standar Deviation
Gen. 0	2124483	10048
Gen. 1	2109371	10179
Gen. 2	2096627	11969
Gen. 3	2078632	16849
Gen. 4	2066798	16378
Gen. 5	2050359	15450
Gen. 6	2044369	17053
Gen. 7	2031435	17650
Gen. 8	2022589	17564
Gen. 9	2010969	16368
Gen. 10	2000600	16302
Gen. 11	1990358	17015
Gen. 12	1981982	18561
Gen. 13	1973250	20812
Gen. 14	1961157	22947
Gen. 15	1951912	24948
Gen. 16	1946563	24365
Gen. 17	1939989	24468
Gen. 18	1933216	26498
Gen. 19	1926995	26537
Gen. 20	1920512	26831
Gen. 21	1915537	27838
Gen. 22	1910987	28876
Gen. 23	1905203	29556
Gen. 24	1901142	30045
Gen. 25	1895903	30870
Gen. 26	1892747	31602
Gen. 27	1889051	32879
Gen. 28	1885599	33489
Gen. 29	1882268	33649
Gen. 30	1879719	34322
Gen. 31	1876788	34486
Gen. 32	1874470	35007
Gen. 33	1871932	35400
Gen. 34	1869609	35382
Gen. 35	1867071	36265
Gen. 36	1864798	36778
Gen. 37	1862479	36531
Gen. 38	1860816	36496
Gen. 39	1858339	37007
Gen. 40	1856537	37069
Gen. 41	1854245	37048
Gen. 42	1852357	37299
Gen. 43	1850282	36829
Gen. 44	1848665	36806
Gen. 45	1846442	36838
Gen. 46	1844061	36524
Gen. 47	1842183	36415
Gen. 48	1840072	36492
Gen. 49	1838424	36485
Gen. 50	1836408	36192
Gen. 51	1834902	35905
Gen. 52	1832630	36581
Gen. 53	1830936	36447
Gen. 54	1829210	36124
Gen. 55	1827455	36243
Gen. 56	1825785	36308
Gen. 57	1823728	36500
Gen. 58	1822153	36267
Gen. 59	1820112	36437
Gen. 60	1818144	36057
Gen. 61	1815855	36326
Gen. 62	1813907	35811
Gen. 63	1812019	35964
Gen. 64	1810220	35494
Gen. 65	1808664	35750
Gen. 66	1806759	35628
Gen. 67	1804942	35934
Gen. 68	1802665	36108
Gen. 69	1801030	36140
Gen. 70	1799232	35588
Gen. 71	1797670	35895
Gen. 72	1795989	35260
Gen. 73	1794163	35322
Gen. 74	1792377	35408
Gen. 75	1790639	35497
Gen. 76	1788991	35448
Gen. 77	1787321	35241
Gen. 78	1785881	35239
Gen. 79	1783929	34938
Gen. 80	1782169	34963
Gen. 81	1780406	34677
Gen. 82	1778385	34974
Gen. 83	1776930	34943
Gen. 84	1774571	34692
Gen. 85	1772942	34971
Gen. 86	1771391	35044
Gen. 87	1769676	34627
Gen. 88	1767977	34612
Gen. 89	1766230	34619
Gen. 90	1764516	34364
Gen. 91	1762849	34466
Gen. 92	1761464	34344
Gen. 93	1759856	34263
Gen. 94	1758060	34102
Gen. 95	1756398	33996
Gen. 96	1754872	33764
Gen. 97	1753426	33918
Gen. 98	1751558	33989
Gen. 99	1750148	33654

Code 38: GA Results

In a first look, we can observe two things. According to the mean fitness at the first generation/epoch (index 0) it is a bit smaller in the ILS algorithm than in the GA and the final mean of fitness in the last generation is higher in the GA than in the ILS. By now we can observe that GA offers better results, but before jumping to conclusions we decided to analyse the Standard Deviation of each generation along 30 iterations.

As we can see in the previous figures, the ILS shows higher values of standard deviation in the first generations (around 24214), and these values decrease slowly to a fitness of 9915 until they increase a bit in the last generations (around 18565). In the case of the GA, it is completely the opposite, in the first generations we have a standard deviation of fitness of 10048 and the last round 33654. We can also see that the ILS shows values of the standard deviation of fitness slightly lower than the ones in the GA in the last generations and the opposite in the first ones.

We can say that the GA is more precise in the fitness values in the beginning but less precise in the final generations, and the ILS is more precise in the final generations. So, overall, the ILS shows smaller values for the standard deviation of the fitness, but the difference is too small to point as a big factor of comparison.

With this in mind, we did another study to take into consideration for the comparison of the algorithms, which was to use both the mean and the standard deviation, to calculate the confidence interval of where the mean fitness of each algorithm belongs. For this comparison we used the best fitness obtained in the 100 generations. The best being the last one, we consider a significance level of 5% (z value), n as the number of iterations made to get the mean and the standard deviation and we used the following $\mathbf{CI_{5\%}} = \boldsymbol{mean\ fitness} \pm \boldsymbol{(z * std) / sqrt(n)}$ formula:

ILS:

$CI_{5\%} = 1758864 \pm 1.96*18565) / \sqrt{30}$

$=] 1752220{,}6; 1765507{,}4 [$

GA:

$CI_{5\%} = 1750148 \pm 1.96*33654) / \sqrt{30}$

$=] 1738105{,}07; 1762190{,}93 [$

Using a confidence interval of 5%, we get that although more precise the ILS, it also can show higher values for the fitness. Comparing with the fitness of the GA in the last generation, although being less precise (having a bigger confidence interval range) it can present smaller values for the fitness. Overall, no statical difference can be inferred (due to the overlap of the confidence intervals and the maximum value of the confidence interval of the GA being higher than the mean value of the last epoch of the ILS).

After some tests, and considering all the runs done previously, we notice that although for 100 generations and 30 iterations the difference between both is very small, the GA shows better results for the fitness and therefore, the best route for the salesperson. Besides, the difference between both algorithms grows tremendously when we implement more generations since with the same 1000 generations the GA achieves a fitness of around 109938, and the ILS stays around 1706326, only a bit smaller than the value in 100 epochs.

We can conclude that the GA Algorithm is slightly more efficient than ILS when we have a lot of generations. GA shows better results for the fitness and therefore, is the best route for the salesperson. It is important to acknowledge, that the difference between both algorithms grows tremendously when we implement more generations since with the same 1000 generations the GA achieves a fitness of around 109938, and the ILS stays around 1706326, only a bit smaller than the value in 100 epochs.