INF3490 2018: Biologically inspired computing
Mandatory assignment 1:
Traveling Salesman Problem

Francisco Jose Nava Lujan
User name: francijn

21st September 2018

Abstract

In this assignment, I implemented functions that helped me solve and understand the traveling salesman problem (TSP). Every program contained alongside this document was programmed completely from scratch.

1 Exhaustive search

For this excercise, we were asked to implement an algorithm that searched over all the possible solutions for the problem. The source code of this implementation is attached along this document with the name ***exh.py***. At the bottom of the file (line 30), you'll see a call to the function tsp, written like so:

$$cl, md = tsp(6, data)$$

Just change the 6 to the number of cities you want to analyze and run.
To measure the amount of time for my program to run, I used the *time* module, saved the starting time on a variable and at the end of the execution, I simply subtracted the end time minus the starting time. The execution times were then recorded as shown in Table 1. After 11 cities, the execution would run out of memory, so I was just able to record up to 11 cities.

Cities	***Executiontime(s)***
6	0.045873403549194336
7	0.02097034454345703
8	0.2154233455657959
9	1.8490538597106934
10	21.289076566696167
11	254.3648784160614

Table 1: Time of execution of the algorithm for n cities

After doing this, I used the Excel regression tool to obtain a 6th order polynomial trendline, and forecast the amount of time it would take to run given a number of cities. The polynomial obtained was:

$$time = 1.3693x^5 - 19.908x^4 + 110.28x^3 - 287.11x^2 + 345.51x - 150.1$$

Q: *What is the shortest tour (i.e., the actual sequence of cities, and its length) among the first 10 cities (that is, the cities starting with B,C,D,H and I)?*
A: The actual sequence of cities and its length is:

('Copenhagen', 'Hamburg', 'Brussels', 'Dublin', 'Barcelona', 'Belgrade', 'Istanbul', 'Bucharest', 'Budapest', 'Berlin', 'Copenhagen')
Tour length: 7486.31

Q: *How long did your program take to find it?*
A: As shown in Table 1, the time of execution for 10 cities is: 21.289 seconds.

Q: *How long would you expect it to take with all 24 cities?*
A: I assume the *permutations* function in *itertools* takes a factorial time to execute itself, since it has to find all the permutations for a given number of cities. In this case, I just used the equation given the regression tool in Excel to forecast that, and the result was: 5665501.9352 seconds which is aproximately 66 days, but given the nature of this particular problem, I'd say is way more than that: to analyze the problem as a factorial one, making a simple function that multiplies the amount of cities (n) times the amount of time taken to execute the program with n-1 cities, the result is:

$$3.95 * 10^{18} s$$

or roughly 125 billion years.

2 Hill Climbing

The source code of this implementation is attached along this document with the name ***hca.py***. At the bottom of the file (line 68), you'll see a call to the function final_props, written like so:

$$final\_props(10, 20)$$

The arguments that this function takes are the number of cities and the times it repeats the algorithm, and it prints all the information that the assignment asks for.

Q: *How well does the hill climber perform, compared to the result from the exhaustive search for the first 10 cities?*
A: Timewise, the hillclimbing algorithm runs many times faster than the exhaustive search algorithm. The problem is that it doesn't always find the global maxima, although sometimes it does. My implementation uses a Greedy approach, meaning it will find the first better solution on the neighbourhood and then analize this new solution's neighbours and so on.

The report of the information as asked for in the assignment is described in the next two tables (Table 2, Table 3):

Property	***Value***
Best tour length	7944.36
Worst tour length	12351.99
Mean	10660.8605
Standard deviation	853.5875973946978

Table 2: Properties of 20 runs for the first 10 cities, with random starting tours

Property	***Value***
Best tour length	21021.75
Worst tour length	28904.49
Mean	25564.463
Standard deviation	2042.3241970056415

Table 3: Properties of 20 runs for the 24 cities, with random starting tours

3 Genetic Algorithm

Finally, I created the genetic algorithm. The source code of this implementation is attached along this document with the name ***gen_algorithm.py***. To run it, simply execute the script. To select number of cities, change the value of the variable *cities* (line 132). The generations number and the lowest population value are calculated

using the number of cities.

The crossover operator I chose was PMX (Partially-mapped crossover), as explained in chapter 3 of the Eiben and Smith textbook. Basically, what I did is, after choosing a lowest population value, I selected the best half of that population of solutions as parents. With this better half, I did the PMX algorithm and obtained the offspring. Then, I took half of the selected parents (1/4 of the new population) and half of the obtained offspring (1/4 of the new population) and passed it over to the next generation. I used random generation of solutions to populate the missing half for the next generation, in order to maintain variation. This way, I made sure the algorithm doesn't get stuck on a local optima.

The two variations are essentially the same, but with the population value multiplied by 2 and by 4. The results of 20 runs for 10 and 24 cities is detailed in the next two tables (Table 4, Table 5).

Population	***Best***	***Worst***	***Average***	***Deviation***
80	7663.70	15763.78	10129.78	2664.85
160	7486.31	15367.68	10156.75	2843.86
320	7503.10	16627.01	10146.15	2851.44

Table 4: Properties of 20 runs for the first 10 cities, using genetic algorithm (200 generations)

Population	***Best***	***Worst***	***Average***	***Deviation***
192	19732.48	37632.05	25543.22	6130.70
384	18759.55	36872.26	25109.34	6560.25
768	20571.83	37410.50	26007.91	5670.17

Table 5: Properties of 20 runs for the 24 cities, using genetic algorithm (240 generations)

Now, in Figure 1, we can see a plot of the average fitness of the best fit individual in each generation for the three variants, for 10 cities. As we can see in the plot, the solution is very much alike for the second and third variation. Although the number of generations on my approach is fixed, I'd say the genetic algorithm behaves better than the hillclimbing algorithm. Exhaustive search is totally out of the question for tour lengths of 12 and higher, at least in my personal computer. So I'd say, the best algorithm is the genetic algorithm.

Q: *Among the first 10 cities, did your GA find the shortest tour (as found by the exhaustive search)?*
A: Yes. As we can see, with a population size of 160 individuals, and a maximum of 200 generations, my algorithm found the best tour distance as found by the exhaustive search, which is 7486.31 (see Table 4)

Q: *For both 10 and 24 cities: How did the running time of your GA compare to that of the exhaustive search?*
A: The GA was many times faster, as the execution time is exponential, and the exhaustive search is factorial. It still took a while to run the GA for the 24 cities, but overall did a much better job timewise.

Q: *How many tours were inspected by your GA as compared to by the exhaustive search?*
A: For the GA, the number of tours inspected were: Number of cities * population size * generations. Since population size and generations depend on the number of cities in my code, for 10 cities, a total of 80,000, and for 24 cities, a total of 1,105,920.
For the Exhaustive search, for 10 cities, a total of 10! = 3,628,800 and for 24 cities, a total of 24! = 620,448,401,733,239,439,360,000 tours.

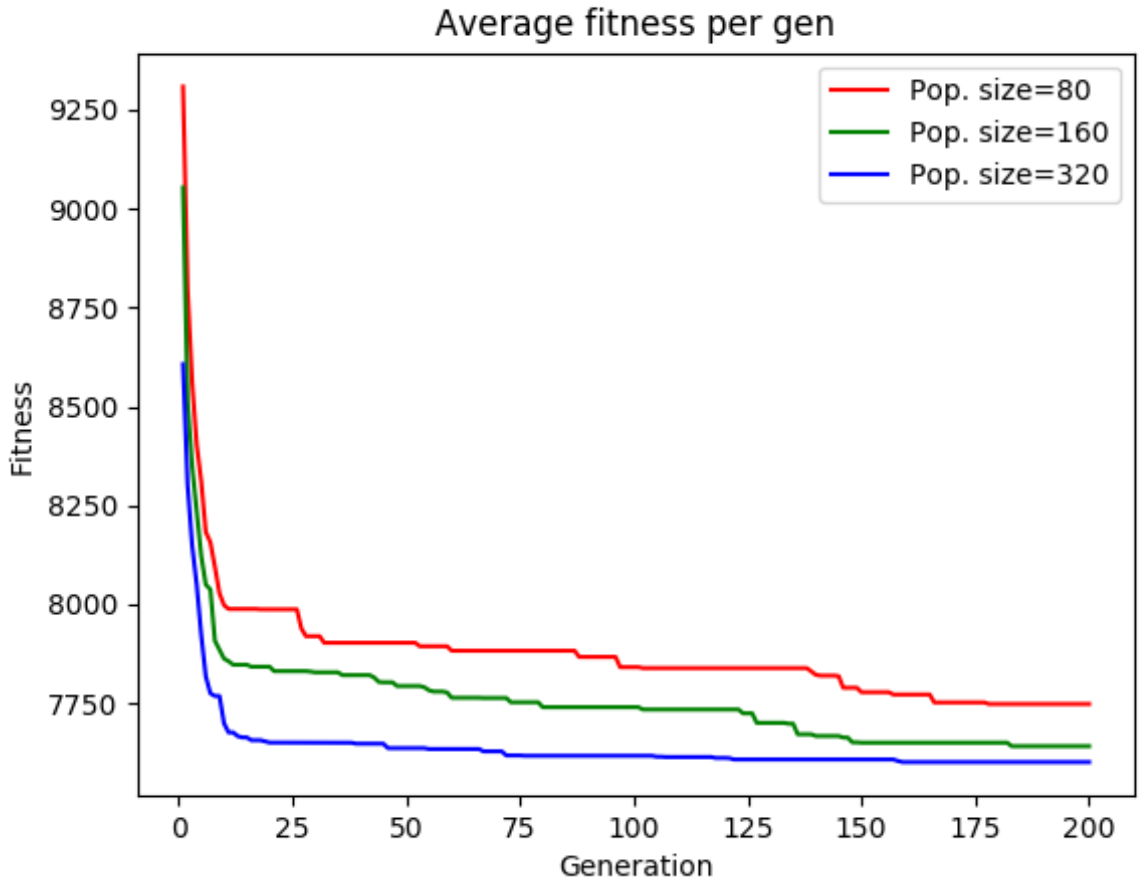

Figure 1: Curve of average best solutions in each generation.