HOW TO RUN

I am recommending testing the algorithms in jupyter notebook because of the structure and Comments that's in the notebook.

JUPYTER NOTEBOOK

- Use the terminal.
- Go to the placement of the jupyter notebook-file, and type 'jupyter notebook' to open the jupyter notebook. Here you can klick on the 'Assignment01.ipynb'-file to access the file.
- To run the code, simply click in the first code-box and press ctrl+enter.
- It is really important that the code gets EXCECUTED IN ORDER of the boxes. This means that if you want to check out the GA first you have to run ALL THE CODE BEFORE GA IN ORDER.

TERMINAL PYTHON FILE

- Use the terminal.
- Go to the placement of the 'Assignment01.py'-file
- In the terminal type 'python3 Assignment01.py' to run the file

EXHAUSTIVE SEARCH

The exhaustive search will find all the permutations possible with the given size of V, where V is the number of cities. It checks all the permutations and returns the permutation with the best result, which in this case is the permutation that gives the shortest distance.

Results:

```
V = 6: 5018.8099999999995
0.0019161701202392578 seconds


V = 7: 5487.889999999999
0.007659196853637695 seconds


V = 8: 6667.489999999999
0.0584108829498291 seconds


V = 9: 6678.549999999999
0.5253582000732422 seconds


V = 10: 7486.309999999999
5.957304000854492 seconds
Actual sequence:  (6, 8, 3, 7, 0, 1, 9, 4, 5, 2) and back to start (6)
```

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)? How long did your program take to find it? Calculate an approximation of how long it would take to perform exhaustive search on all 24 cities?

- The actual sequence of cities for 10 cities is (6, 8, 3, 7, 0, 1, 9, 4, 5, 2), where each number represent the index from 0 to 9 from left to right. It took aprox 5.96 seconds to find the shortest tour among the first 10 cities.

  My prediction for how long it would take to perform exhaustive serach on all 24 cities will be based on the pattern shown after running from V = 6 to V = 10. As of writing my numbers, from running the code from above, are as follows:

  V = 6: 0.0019161701202392578 seconds
  V = 7: 0.007659196853637695 seconds
  V = 8: 0.0584108829498291 seconds
  V = 9: 0.5253582000732422 seconds
  V = 10: 5.957304000854492 seconds
  V = 11: 66.67615795135498 seconds

  It would look like that the time of execution takes 10 times longer for each city added to the calculations. Therefore, my approximation would be something of: 66*10^13 seconds to find the shortest tour with an exhaustive search on all 24 cities. This has to do with the V! (factorial) amount of permutations.

  Shortest tour among first 10 cities in order:

  ['Copenhagen', 'Hamburg', 'Brussels', 'Dublin', 'Barcelona',
  'Belgrade', 'Istanbul', 'Bucharest', 'Budapest', 'Berlin']

HILL CLIMBING

The hill climbing algorithm starts at a randomly generated spot (permutation/solution) and tries to go ‘left’ and ‘right’ depending on swapping genes. If the gene-swapping is successful, it will ‘exploit’ the swap, and try a new route with the new permutation as base. This is done until there is no improvement in a given space of trails, (breaks when limit is reached). This is then tested repeated a given amount of times to replicate different starting points.

```
Comparison of exhaustive search and hill climbing for the first 10 citi
es:
V = 10
Starting points: 20
exhaustive: 7486.309999999999       |      5.9528210163116455 seconds
hill_climb: 7486.309999999999       |      2.339359998703003 seconds

There is a huge benefit to using hillclimbing at this point!

Stats for 10 cities using hill climbing:
Starting points: 20
Best result: 7486.309999999999
Worst result: 7486.310000000001
```

```
Mean result: 7486.3099999999995
Standard deviation: 5.974895296565468e-13

Stats for 24 cities using hill climbing:
Starting points: 40
Best result: 12517.89
Worst result: 13850.080000000004
Mean result: 13222.2685
Standard deviation: 366.99298746025215
```

The stats for 10 cities using hill climb shows that the best worst and mean is almost the same. This could be that we are running the code so many times inside that the given best always will be found in each run.

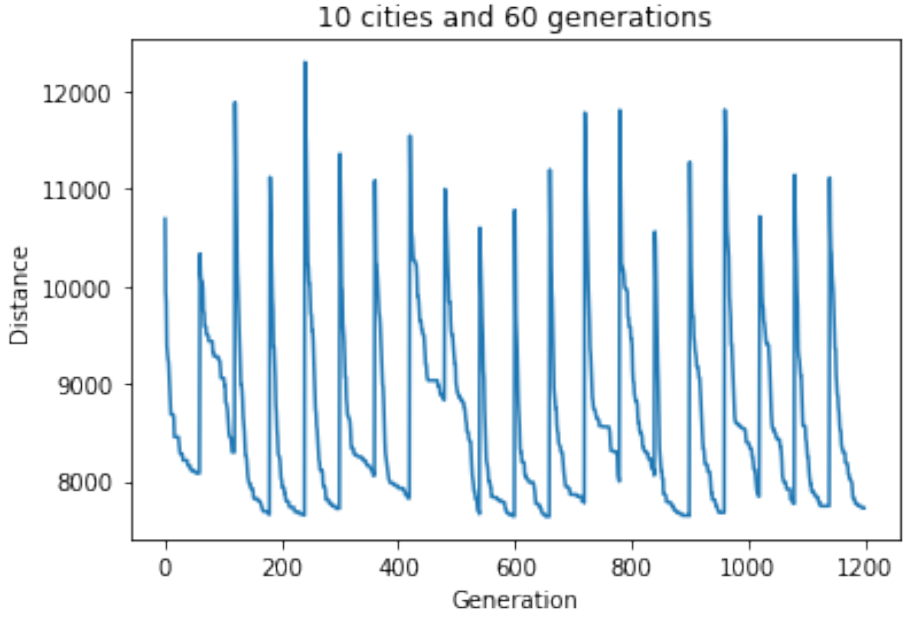
GENETIC ALGORITHM

My genetic algorithm is using order crossover and swapping genes as mutation. At first, it creates a random starting population. From this it creates two children from using order crossover with two parents. Then it mutates the children by swapping two genes at random. Then it picks the best offspring to become the new generation, but only an amount equal to the population size. This process will then be repeated at the number of generations given the function. I have been testing with 60, 300, and 1000 generations.

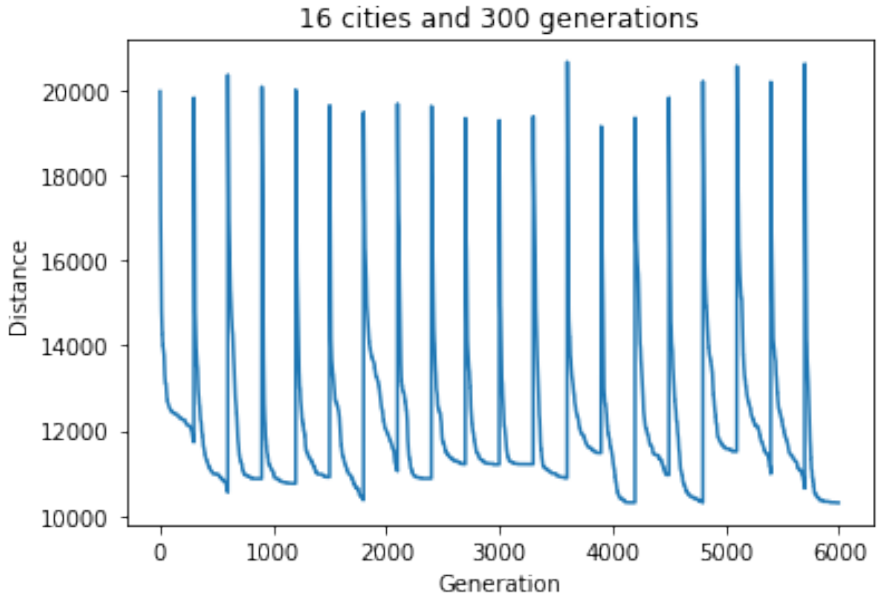
I am not quite sure if the task wants us to run our code to run for 20 generations, or if we are allowed to pick the amount of generations ourselves but run the code 20 times. When I run my GA with only 20 generations, I get a result of 7486.31 with V=10, but when I run the code for V=16 or V=24 the results are getting worse and worse. So for the sake of good results, I have used the version which gives me the best results overall, but I have included a plot which contains average fitness of the best fitted individuals of a whole run through generations with 20 runs of the code (which gives 20 averages in total), and a generation count of 60, 300, and 800.

The stats for 24 cities take the longest time to execute, aprox 30-40 seconds.

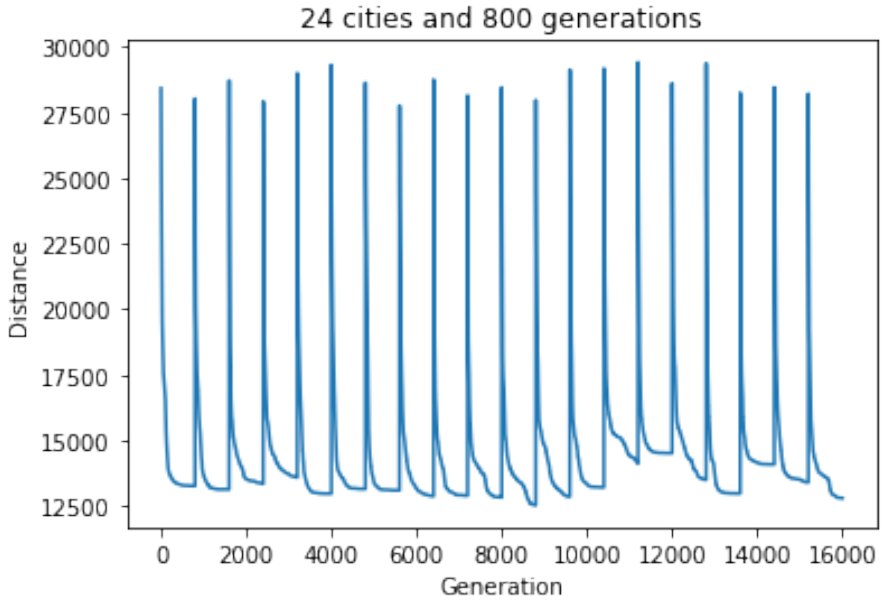
```
Stats for 10 cities using a genetic algorithm:
Best result: 7486.309999999999
Worst result: 11135.19
Mean result: 8048.292083333304
Standard deviation: 677.1181723111962
time usage: 0.26178908348083496 seconds
```




```
Stats for 16 cities using a genetic algorithm:
Best result: 10049.61
Worst result: 18817.92
Mean result: 11516.81751499992
Standard deviation: 1377.6291318347332
time usage: 3.9874069690704346 seconds
```

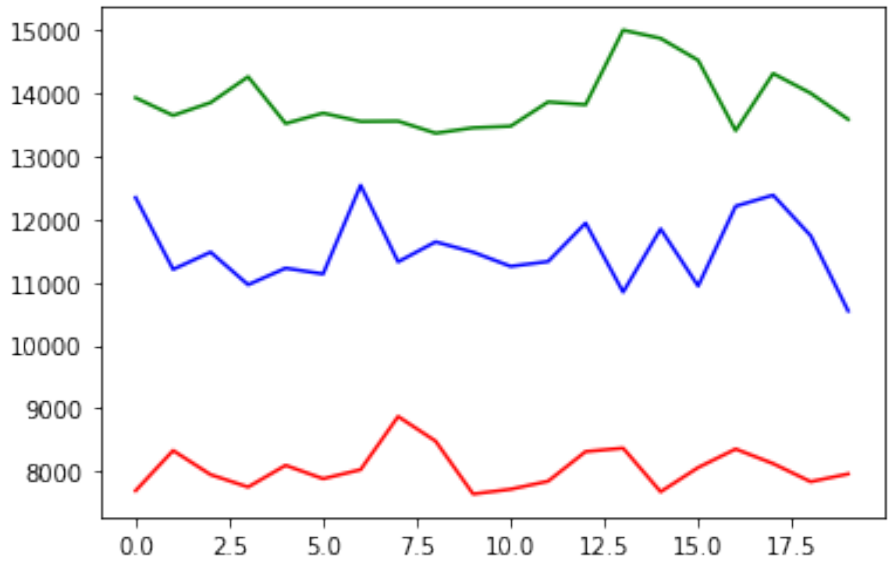


```
Stats for 24 cities using a genetic algorithm:
Best result: 12287.069999999998
Worst result: 27904.79
Mean result: 13881.832269374214
Standard deviation: 1706.2639297340625
time usage: 28.105993032455444 seconds
```



Average fitness of the best fitted individuals of a whole run through generations with 20 runs of the code (which gives 20 averages in total), and a generation count of 60, 300, and 800:

Red: V=10
Green: V=16
Blue: V=24



The best result I got with using elitism with V=10 and a generation size of 60 offspring is 7486.309999999999.

The best I got with the Genetic algorithm with V=24 is 12287.069999999998 after many tries. (800 gen, 20 runs). 800 is probably a lot more than needed, but I wanted to be sure that there is no room for improvement later down the line with some random mutation. Also, I tried to look at the graph of one run to see where it started to flatten. I also wanted the answer to be consistent.

By looking at the plot over average fitness by each generation, it looks like the genetic need more time (more than 20 generations) to find the global optimum when the number of cities rises. The graph has not flattened and therefore it may be a sign of a better potential global optimum. Sometimes my GA performs better than my hill climb, and sometimes it does not. This could be because the mutations of my GA in one run is really good, and other times really poor.

To improve my GA I could've probably used another crossover or other forms of mutation that's more likely to give a more consistent answer for 24 cities, since this is the one where the answer changes from each run, unlike for 10 (at 7486.3099999999995) and 16 (at 10049.61) cities which are quite stable (under testing that is).

When generation size is of 60, V=10 and the algorithm runs for 20 times, the number of visited solutions would be aprox gencap * (parents+offspring) * runtimes = 60*20*20 = 24 000 solutions (not counting the solutions that are alike). For the exhaustive search that would be 10! (factorial) = 3 628 800. That's a really good reduction in time usage! And this number is even better when calculated with V=24 even though we are increasing the gencap and popsize.

The time usage for the GA is 0.32283687591552734 seconds for V=10, popsize=10, gencap=60, and 20 runs. While the time usage for the exhaustive search is 6.0101377964019775 seconds.

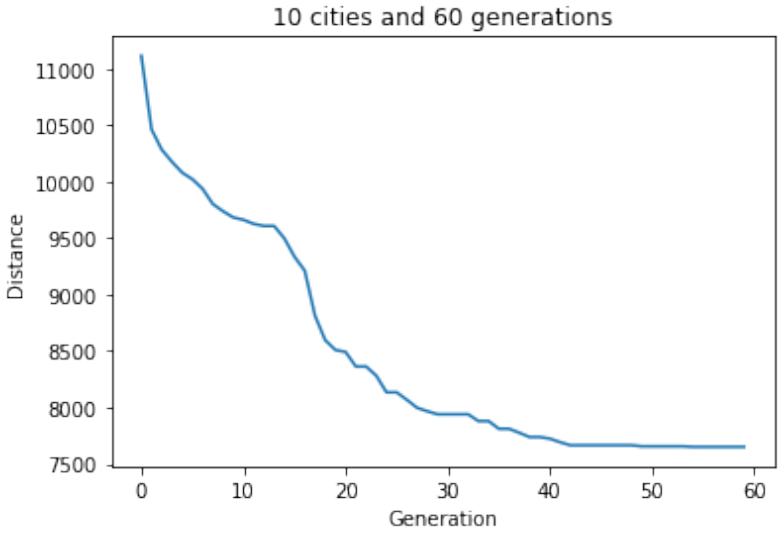
The time usage for the GA is 33.02506899833679 seconds for V=24, popsize=24, gencap=1000, and 20 runs. While the time usage for the exhaustive search is my calculated 66*10^13 seconds. Quite a difference!

OVERALL COMMENT

It looks like the hill climb and GA does a good job at finding the global optimum, but in some cases it gets stuck in some local optimum. For the hill climber this is understandable when the number of cities is rising because of the algorithm's simplicity. When it comes to the GA I presume the number of randomness is too low in my implementation to consistently return the lowest recorded distance, while testing, every single time. To find the best solution every run is not too easy, since both hill climb and GA are based on randomness. But nonetheless it gives a result close to the lowest recorded every time. The exhaustive search is really outpaced by both hill climb and GA in all cases.

Overall, I'm pretty satisfied with the results that the algorithms are giving. They show a gradual decrease closer to the global optimum for every iteration, which is what the

algorithms are made to do. When both the hillclimber and GA reach a local optimum, we see that the random swaps and mutations makes the algorithms go even deeper. Here's a closeup of one run of the GA with 10 cities and 60 generations:



Average fitness over generations

As the algorithm approaches aprox. 9600 it starts to flatten a bit, but then the randomness of both crossover and mutation kicks in and changes the whole graph. Before it goes in a bit of a staircase drop until the final flattening.