Assignment 1
TRAVELLING SALESMAN

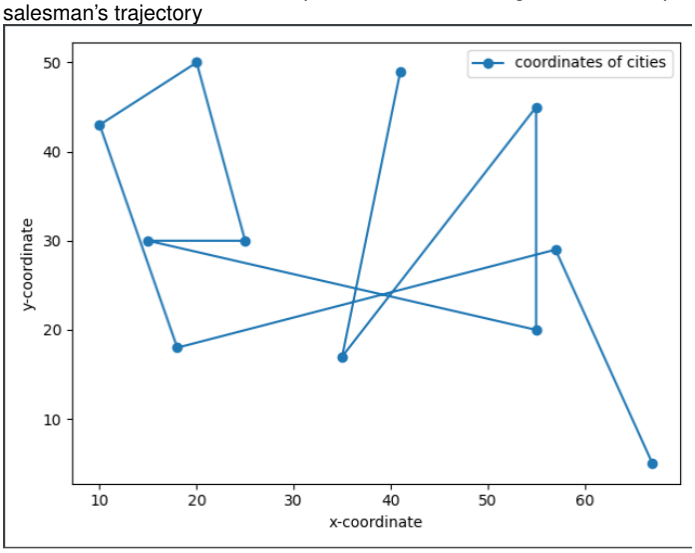
Jennifer Za

Abstract—In this assignment I have realised several experiments tackling the travelling salesman problem exemplified by problems in TSPLIB. I have explored various algorithms ranging from constructive heuristic ones, genetic algorithms and memetic algorithms and attempted to fine-tune their parameters to attain improved solutions to problems at hand. I have attempted to make my code as interactive as possible, whereby through changing macro parameters one can attain a new version of solution, fine-tuned to fit specific instances of the travelling salesman problem. This report explains how to proceed in tweaking the code along with explaining my thought processes behind each component.

1 Introduction

THE travelling salesman problem (hereinafter referred to as TSP) is a discrete, combinatorial, NP-hard problem. As it is the case with other combinatorial problems, its solution is a combination of feasible components which satisfies pertinent criteria set by the particular problem variation at hand. In its most general form, TSP is formulated as a task faced by a travelling salesman of visiting cities scattered across a plane of arbitrary dimension in a way which would minimise distance traversed by the salesman whilst visiting each town and returning back to origin, thereby traversing a Hamiltonian cycle in cases such as ours, where every two cities are equidistant (path from city a to city b is the same length irrespective of where one starts) and triangle inequality holds, due to which there is no need for a traveller to return to already visited non-starting positions. Given the fact that TSP is a NP-hard problem, it cannot be solved in polynomial time. For instance, a brute force algorithm approach takes a factorial of the number of cities to solve, which becomes strenuous even with lower tens of cities. This report discusses various methods and configurations I employed in trying to tackle the TSP problem. Through algorithms such as the genetic algorithm, simulated annealing or use of local search methods we may not reach the optimal solution of the problem at hand, as the brute-force approach strives to get. On the other time it can strike a balance between getting a good-enough solution in a fraction of time.

Figure 1. Graph illustrating the travelling salesman trajectory. Points are cities scattered across the XY plane, lines connecting them make up salesman's trajectory



2 Experiments

2.1 Genetic Algorithm

A general genetic algorithm consists of creating a first generation, calculating its members' fitness as per certain fitness function, selecting top performing solutions, "crossing-over", which consists of making new, still valid, solutions carrying traits of their "parent" solutions and their subsequent mutation, since just like in the case of natural selection and genetics, there is always a nonzero probability of change in genetic codes. At each stage of the process, there are several things to consider. The subsequent sections discuss which choices I made whilst building my genetic algorithm and reasoning behind them.

2.1.1 Initial population

The initial population of solutions was created by the function called "randomsolution()", which generated as many random orderings of cities a salesman could visit, as was possible given a "cap" represented by the POPULATION-SIZE macro. This macro can and has been tweaked several times and has oscilated in the interval <100, 1000>, which was in my eyes suficcient for statistical inferrence.

2.1.2 Fitness evaluation

There are at least two types of problems in TSPLIB, library of TSP probmlems we drew from. One kind gives out a matrix of distances, another for instance does not conform to triangle inequality. For the sake of clarity, I have focused only on those tests which present data as points in Cartesian plane. This meant that "fitness" of solutions became the Euclidean

distance between all cities traversed. This calculation is done in the function called eucdist().

2.1.3 Selection

After evaluation of fitness of all members of our population. Selection of the fittest ensued. There are several ways of going about selecting members. Some call for conducting a tournament of sorts among members of the population to determine their order and then create "offspring" solutions from the best couple of solutions, gradually working its way through the ranks of solutions, making offsprings along the way. This way, however, in my perspective fails to mimic natural selection genetic algorithms are based off. In nature, the fittest individuals do not breed merely with fittest counterparts, but their edge consists in a greater likelihood of mating at all. Furthermore, it would be a rather unstable evolutionarily "stable" strategy for one to mate only with those "similar" to them in fitness, which may cause loss of variation leaving offsprings less capable of adaptation. That is why in my genetic algorithm, I have opted for a combination of Elitism selection along with a quasi-"roulette-wheel" type of selection. First, a fraction of the fittest members of the old population determined by ELITE SELECTION FRACTION macro is incorporated into the next one. One could formulate this as the "chosen ones" who get a ticket to the next round. The rest of solutions is then chosen from the original population through a method which in some respects resembles Simulated Annnealing. A temperature is chosen at first, equal to the size of the population, and gradually as we loop through each member of the original population, it is decreased and compared to a number generated randomly with each instance of the loop. If said number (in the range of 0 to the population size) happens to be smaller than temperature at that moment, the solution corresponding to the particular iteration of our loop is added alongside the "elite" into the new population. Since temperature decreases gradually and is higher when we first start off looking at the "fitter" of solutions, it is clear that the objective of fitter individuals having a greater chance of "survival" holds. At the same time, however, to make sure that even less fortunate of strategies makes it through, temperature descends at an exponential pace to ensure nonzero probability of selection even for the less fit solutions.

2.1.4 Crossing-over

After we have selected the "winners" of each generation fit to mate, there are several ways of creating "offspring" solutions from them. Whilst crossing over of chromosomes in genetics at first glance does conform to few rules and is seemingly random, in our case, we have to make sure several conditions hold. We cannot go about selecting random parts of solutions and "gluing" them together, since we may run into the problem of not visiting all cities and visiting cities multiple times, which destroys the purpose of finding the Hamiltonian cycle. Similarly, we have to make sure created solution starts and ends in the same city. There are several possible ways of going about this. Some more obvious and others less so. In my experiment, I have decided to implement the cycle crossover method first proposed by Oliver et. al.. Explanation of the algorithm is beyond the scope of this report, however, its explanation taken from is observable in fig. 2. The second crossover algorithm selected is the Non-Wrapping Order Crossover algorithm, which is a variation of the widely-used Order Crossover algorithm taken from (Cicirello, 2006). Its intuition is demonstrated in fig. 3. In order to test how the two crossover methods stand against each other, a test in the form of pub01.in has been derived from the elil101.tsp problem from the TSPLIB library. The two methods were tested head to head with a genetic algorithm on 5 runs, with elite selection fraction of 0.05 and swap perturbation (further explained in the subsequent section).

	1	2	3	4	5
CO	2762	2766	2725	2715	2677
NWOX	2462	2874	2911	2752	2286

As per our results, we can observe that there was not a very significant change from one crossover method to another. This could be explained either because solutions provided by the two methods were too similar for paths of our particualar size, or maybe the two methods in the context of TSP do not prove to generate differing solutions. In future, it would be interesting to see how the two methods fare with very different types of TSP problems and different configurations of global parameters.

2.1.5 Mutation

Having created a population of offspring solutions from a population of parents representing mostly the fittest of members of our preceding generation, the next step along the genetic algorithm pipeline was to mutate said offsprings, since even in nature we see random variations of phenotypes from parents to children. I have tackled mutation in the context of TSP from two angles: first, I have created the swap perturbation, which swaps two cities in the salesman's path. In order to make the mutation substantial, the number of swaps performed on each offspring is specified by the macro SWAPS. This way, we have an insight into to what extent does the variation influence outcomes we may get. Whilst too few swaps could prevent us from leaving a local optimum, too many could disable us from attenuating our results and reach any optimum at all. It is a matter of fine-tuning the said parameter and perchance rethinking it with every input. In order to inspect to what extent my algorithm in the instance of pub01.in is sensitive to rate of mutation, I have conducted a mini-experiment which is summarised in the table below.

0	1	2	4	10
2875	1855	2462	2762	2911
2973	1776	2271	2766	2955
2871	1852	2180	2725	2965
2862	1840	2286	2715	2951
2911	1918	2264	2677	3027

The table shows results of genetic algorithm on a population of 100 paths with 100 generations, using the cycle crossover method. The only parameter changed was the macro determining the number of swaps made. Each column represents results achieved in 5 runs with respect to a certain number of swaps specified by numbers on the first line. As can be observed, results vary substantially even

Figure 2. Cycle crossover algorithm

2.1.3. Cycle Crossover Operator. The cycle crossover (CX) operator was first proposed by Oliver et al. [27]. Using this technique to create offspring in such a way that each bit with its position comes from one of the parents. For example, consider the tours of two parents:

$$P_1 = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8),$$
$$P_2 = (8\ 5\ 2\ 1\ 3\ 6\ 4\ 7). \quad (10)$$

Now it is up to us how we choose the first bit for the offspring to be either from the first or from the second parent. In our example, the first bit of the offspring has to be 1 or 8. Let us choose it be 1:

$$O_1 = (1\ \times\ \times\ \times\ \times\ \times\ \times\ \times). \quad (11)$$

Now every bit in the offspring should be taken from one of its parents with the same position, it means that further we do not have any choice, so the next bit to be considered must be bit 8, as the bit from the second parent is just below the selected bit 1. In first parent this bit is at 8th position; thus

$$O_1 = (1\ \times\ \times\ \times\ \times\ \times\ \times\ 8). \quad (12)$$

This turnout implies bit 7, which is the bit of second parent just below the selected bit at 7th position in first parent. Thus

$$O_1 = (1\ \times\ \times\ \times\ \times\ \times\ 7\ 8). \quad (13)$$

Next, this forced us to put 4 at 4th position as

$$O_1 = (1\ \times\ \times\ 4\ \times\ \times\ 7\ 8). \quad (14)$$

After this, 1 comes which is already in the list; thus we have completed a cycle and filling the remaining blank positions with the bits of those positions which are in second parent:

$$O_1 = (1\ 5\ 2\ 4\ 3\ 6\ 7\ 8). \quad (15)$$

Similarly the second offspring is

$$O_2 = (8\ 2\ 3\ 1\ 5\ 6\ 4\ 7). \quad (16)$$

But there is a drawback that sometimes this technique produces same offspring, for example, the following two parents:

$$P_1 = (3\ 4\ 8\ 2\ 7\ 1\ 6\ 5),$$
$$P_2 = (4\ 2\ 5\ 1\ 6\ 8\ 3\ 7). \quad (17)$$

After applying CX technique, the resultant offspring are as follows:

$$O_1 = (3\ 4\ 8\ 2\ 7\ 1\ 6\ 5),$$
$$O_2 = (4\ 2\ 5\ 1\ 6\ 8\ 3\ 7), \quad (18)$$

which are the exactly the same as their parents.

Figure 3. non-wrapping cycle crossover algorithm

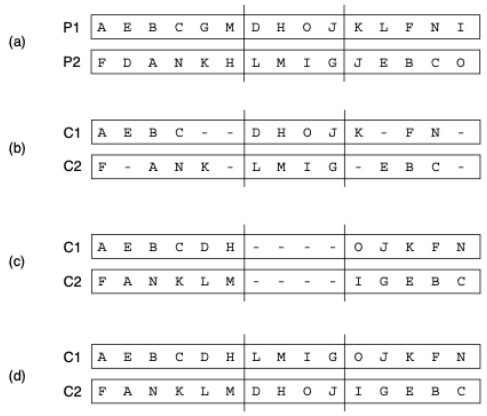


Figure 2: Example of the steps taken by NWOX

with slightest of changes. At first, I deemed 4 swaps to be appropriate solely based on my inner biased heuristic. After changing the swap amount to 2, however, results substantially improved and bringing swaps to a mere one has improved results even more dramatically. This seemed to show a trend that the less interference with solutions the better. However, after changing the number of swaps to zero, it is observable that this leads nowhere, as solutions dramatically worsened beyond values with 4 permutations. Similarly, when 10 permutations were done, we achieved the worst results, which proves our supposition that the more perturbations are made, the harder it is to converge towards any optimum.

The second type of mutations chosen was the flip mutation, which - unexpectedly - consisted in reversing a portion of a path. The start and end of the path reversed were generated randomly, which means the change made to the path was likely greater than that done with our preceeding perturbation. I believe this is the reason why it was outperformed by the swap perturbation on pub01.in case, which, as explained, was rather sensitive to perturbation rate. The flip perturbation can be experimented with by changing the macro PERTURBATION from 1 to 2. In the case of pub01 and by changing the FLIP SEGMENT SIZE macro, which specifies what random fraction of the path at hand should be flipped.

2.1.6 Evolution

Having mutated all members of the newly-formed population of solutions to the TSP at hand, the portion of the genetic algorithm is to repeat, thereby emulating an evolutionary process. The number of iterations is specified by the macro GENERATIONS and can be easily altered. Intuitively, we may assume that whilst fewer generations may prove to be computationally less exhaustive, we are more likely to achieve superior optima with larger numbers of iterations. The default number of generations in my code is 100. This, in my view at least, proved to be a sufficient, as (local) optima found stopped changing after a couple of

tens of generations. However, it should be noted that this is not a robust indication of a sufficient number of iterations, as a great portion of results rely on values of variables described so far, ranging from perturbation rate, crossover method, even depending on the size and fitness of the initial population.

The genetic algorithm in its pure form can be run by changing the macro ALGORITHM to 1 and tweak by changing macros explained up to this point. The subsequent tables demonstrate results attained with cycle crossover method, swap perturbation of 1, population of 100, 100 generations and elite selection fraction of 0.05 run on different test cases extracted from TSPLIB. Some cases were adjusted to resemble in size elil101.tsp problem, to make comparisons between tests and across methods easier.

d15112.tsp	eil101.tsp	a280.tsp	berlin52.tsp
545480	1855	5594	15716
556491	1776	5857	15893
491652	1852	5359	17103
582404	1840	4988	17046
566242	1918	5583	16071

bier127.tsp	brd14051.tsp	d15112.tsp	d18512.tsp
275565	29454	549705	30915
298031	28286	504827	31843
295226	25012	562058	31911
278994	33202	536193	32155
297303	25300	588342	31892

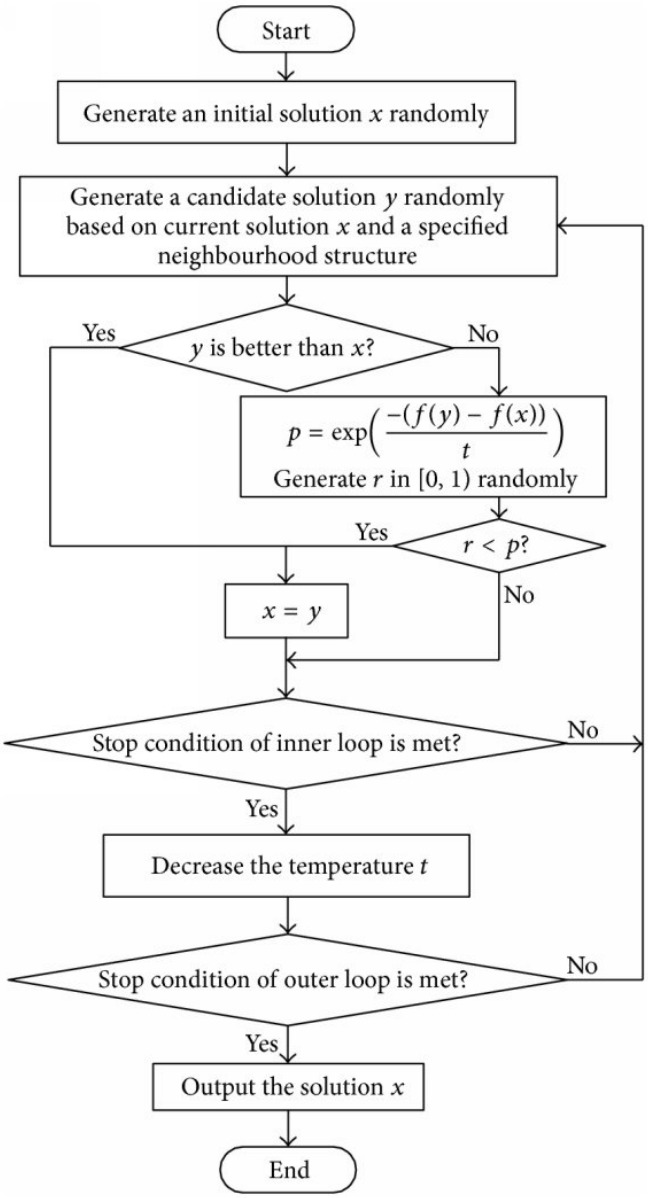
dsj1000.tsp	kroA200.tsp
33730416	96284
34873698	91168
33901647	96977
34934706	99073
33167247	92839

It was a relief to see that on all tests the algorithm seemed to produce sensible results. It must be noted though that some test cases' sizes were altered. Some of the test cases had optimal solutions published in TSPLIB. Whilst my algorithm did not attain solutions identical to those that did, it did optimise and alter its solutions as generations progressed, which I interpret as a sign that after tweaking parameters of the algorithm, my solutions could approach optimal ones more closely. In hindsight, had I chosen a higher-level language such as Python, I could probably get away with comparing test instances in a greater detail, as I for example would not have to alter nearly all data types to accommodate different formats of tests in TSPLIB. Furthermore, some tests in the format of elil101.tsp in the library were missing (inaccessible due to some error probably). This, along with differing formats of test instances made it harder to get 10 or more similar tests. All tests used are included in the dataset file attached with this report.

2.2 Simulated Annealing

To be able to compare results attained through Genetic Algorithm, I have decided to implement Simulated Annealing algorithm. A succinct explanation of my algorithm's workings is conveyed in fig.4 in a flow-chart (done by Zhan, Shi-hua, et al. 2016).

Figure 4. Simulated annealing algorithm



Results obtained with simulated annealing are discussed further down along with other methods used. In order to tweak the algorithm to one's liking, one can change the macro TEMPERATURE, which determines from which temperature descend begins. By default, it starts at a 100. As conveyed by the table below, which shows results attained at temperature 100 vs at 1000, there does not seem to be a statistically significant improvement in the algorithm's findings with respect to the eil101.tsp test.

100	1000
1720	1760
1620	1556
1686	1771
1658	1781
1606	1624

Figure 5. Greedy algorithm

```
procedure GreedyAlgorithm()
1.  S ← ∅;
2.  Evaluate the incremental cost of each element e ∈ E;
3.  while S is not a complete feasible solution do
4.      Select the element s ∈ E with the smallest incremental cost;
5.      S ← S ∪ {s};
6.      Update the incremental cost of each element e ∈ E \ S;
7.  end_while;
8.  return S;
end.
```

2.3 Nearest Neighbour algorithm

This algorithm, also known as the greedy algorithm, consists of generating a random starting position for the traveller to start from and then proceeds in trying to find the optimal path by visiting the immediately closest, yet unvisited, neighbouring town next. The algorithm is briefly explained in fig. 5., by a pseudocode by Martins, et.al. (2006).

In order to make the code run the greedy algorithm, it suffices to change ALGORITHM macro to 4. When I ran this algorithm on my eil101.tsp test, I was rather excited and disappointed at the same time. On one hand, the algorithm performed extremely well, which can be seen in the last section of this report. The TSPLIB library states that the best known solution to this instance was 629, which is pretty close to my best solution achieved with this seemingly short-sighted algorithm, which was 734. On the other hand, the great extent to which it outperformed my genetic algorithm and other local searches, memetic algorithm and heuristic searches seemed to bbe a testament to what extent I have yet to optimise my genetic algorithm to make it better.

2.4 Memetic algorithm

Having done the genetic algorithm and being able to see how it fares in comparison to other ones, a thought of improving the standing configuration of the genetic algorithm arises. I have created six versions of the genetic algorithm (which could be extended to 9) differing in the way they use local search to improve current code. To run memetic algorithms, ALGORITHM macro has to be changed to 3 and then the MEMETIC ALGORITHM macro can be also switched from 1 - 4, where 1 is the original genetic algorithm with an improved initial population (depending on the PERTURBATION macro, this could be done in differing ways), the second memetic algorithm, for which MEMETIC ALGORITHM must be 2, alters the "elite" population of the fittest top-performing solutions which are added into our breeding pool. The alteration is done through the function perturbate(), which depending on current perturbation "mode" selected in respective macros, alters our top performers and makes them even better. Third version of the memetic algorithm consists in improving already mutated children, which then become members of the new generation along with mutation of the elite exemplified by the second algorithm. This way, we achieved slower, yet even more precise version of the preceeding algorithm.

The degree of improvement in all memetic algorithms is regulated through the macro IMPROVEMENTS, which is currently at a 100. The greater its value, the better the results we shall see, but at the same time the longer it is going to take the script to yield them.

1	2	3
2479	755	688
2370	757	679
2322	698	701
2396	738	663
2425	756	679

As can be viewed in the table above, each memetic genetic algorithm yielded different results. It should be noted first, that all of these were performed with the flip perturbation explained earlier, with a population of 100, over 100 generations, with an elite selection fraction of 0.05, 1/8 translocation segment fraction determining the size of the flipped part, performed on the eil101.tsp problem.

As can be observed, improving only the initial population as done in the first memetic agorithm represented by the first column of the table above, does not yield results significantly different from those achieved with the classic genetic algorithm. On the other hand, introduction of improvement to the best scoring "elite" population right after selection proces, showed terrific improvements to our score ranging from 700-750. Mind you that the best score recorded on this particular problem set was 629.

Furthermore, after additionally changing our second memetic experiment by introduction of local search applied to members of the new population after mutation, as anticipated, we can see even greater improvements, where results range from 660 - 700, which is even closer to the best score ever recorded. Having seen my genetic algorithm being outperformed by a greedy, short-sighted Nearest Neighbour algorithm does make one question whether there is not a fundamental flaw in the algorithm. However, after seeing what difference relatively small adjustments can make, as exemplified by the second and third memetic algorithm, it seems that with further fine-tuning, an optimal-path finding algorithm could be reached.

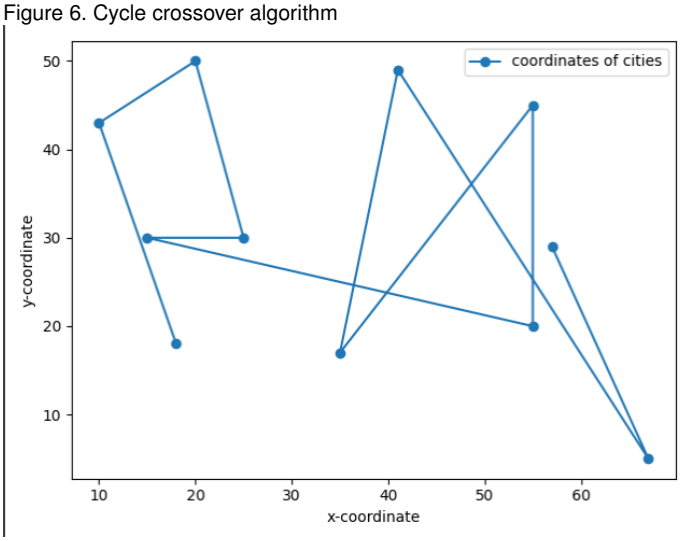
3 Comparison

Lastly, let me demonstrate how each of my algorithms faired when tackling eil101.tsp experiment, as conveyed by results showing runs from best configurations of each of the algorithms in tables below.

GA	SA	Greedy	Memetic algorithm no. 3
1855	1720	817	688
1776	1620	856	679
1852	1686	835	701
1840	1658	782	663
1918	1606	801	679

At first, results of the genetic algorithm seemed rather discouraging. Even though there was a clear improvement from initial values across generations, after running the simulated annealing local search algorithm, it could be seen that results reached through it were comparable, if not better than those of the genetic algorithm. Furthermore, after running the nearest neighbourhood algorithm, which rather primitive heuristic is only to minimise distance to

Figure 6. Cycle crossover algorithm



the next city, and seeing it outperform by a great margin both simulated annealing and genetic algorithm, seemed to suggest either a flaw in my code, or or perhaps a flaw on a more abstract algorithm level. However, having altered genetic algorithm with local searches at different stages, as exemplified in section 2.4, makes the genetic algorithm outperform all other algorithms and their permutations so far.

A certain dose of resilience and patience is necessary to improve current memetic algorithm further to become even better and to fit other TSP instances at least as well as it does eil101.tsp. It would be very interesting to see, how introduction of noise into perturbations at every level change results, or to examine closely to what extend does the result depend on crossingover technique employed.

Lastly, I would like to mention that whilst the code implementing the solution to the problems does not generate a graph, I have written a plotting script in the file visualisation.py, which I would gladly demonstrate in person. For reference, I am enclosing an example of the kind of visualisation it generates.

REFERENCES

[1] Cicirello, Vincent A. *Non-wrapping order crossover: An order preserving crossover operator that respects absolute position*. Proceedings of the 8th annual conference on Genetic and evolutionary computation. 2006.

[2] I. M. Oliver, D. J. d. Smith, and R. C. J. Holland. *Study of permutation crossover operators on the traveling salesman problem.* https://dl.acm.org/doi/10.5555/42512.42542

[3] Martins, Simone L., and Celso C. Ribeiro. *Metaheuristics and applications to optimization problems in telecommunications.* Handbook of optimization in telecommunications. Springer, Boston, MA, 2006. 103-128.

[4] Zhan, Shi-hua, et al. *List-based simulated annealing algorithm for traveling salesman problem.*. Computational intelligence and neuroscience 2016.