Combinatorial Optimization Final Report

Nicoló Ruggeri

1 Introduction and First Notations

In this report we analyze the performances of two different methods for solving a particular instance of the TSP problem. Specifically, we solve the problem presented below with a linear programming network flow formulation, as requested in homework 1, and with a genetic algorithm presented in section 3.
In our case we formulate the problem on a complete graph $G = (V, E)$ where the adjacency matrix $A \in \mathbb{R}^{|V|}$ of G is symmetric and dense. In our case we consider a weighted graph, so that $a_{i,j} \in \mathbb{R}_{\geq 0}$ is the cost of moving from point j to i. We will explain how these distances are generated in section 2.
The particular properties of the matrix reflect two assumptions that we make on our model, namely:

- A symmetric. This means that the cost of going from j to i is the same as going the other way around. This makes sense for the underlying practical problem at hand, i.e. the one of a drill moving from position i to j
- A dense (and in the particular case of our experiments, with almost all elements strictly positive, see sections 2 and 4). With a slight abuse of terminology, here we mean that all entries of A are available (or, equivalently for the problem, lesser than infinite). This reflects the fact that $ij \in E \ \ \forall i, j$, i.e. the drill can move between any two arbitrary points

In the following section we will refer to points equivalently as nodes, and to the connections between them as edges (from the graph formulation).

2 Data Generation

2.1 Distance Matrix

We frame the problem as that of moving between points in the unitary square $[0, 1]^2$. Notice that this doesn't represent a restriction, since it can be seen as a rescaling of a problem of any size. Given that we have a finite number of points, we can number them from 1 to n, so that the vertex set of the graph is $V = \{1, \dots, n\}$. Now we consider these two dimensional sampled vectors

$x_i \in [0,1]^2$, that represent the position of every point in the drilling board. If we want to generate the cost matrix A from these points, a natural interpretation could be to assume a movement time proportional to the distance between the points. In fact in the experiments section 4 we consider a distance matrix given by

$$a_{ij} := ||x_i - x_j||_2 \tag{1}$$

The choice of the 2-norm is arbitrary, and could be replaced by any euclidean norm, which could in principle affect the final optimal solution.
Notice that this approach is easily generalized in many ways. For example to exclude some edges from the final solution we just need to impose $a_{ij} = +\infty$. To differenciate the cost between points we can substitute the formulation above with $a_{ij} = f(x_i, x_j)$ for some function f (that doesn't even need to be symmetric in the arguments, as the genetic algorithm from section 3 and the OPL solver accept totally general distances).

2.2 Experimental Data

Following the process above, we just need to generate any number n of points $x_i \in [0,1]^2$, and then build the cost matrix that we will input to the solvers. In our case we decided to generate these points at random in the following way:

- a number n_1 of points is generated according to a mixture of equiprobable two-dimensional gaussians with given variance σ^2 and means $\mu_1, \dots, \mu_k$. Since the sampled points could end up being outside the allowed space, we clip all the points' coordinates in $[0,1]$.
- the remaining $n - n_1$ points are radomly distributed on a fixed granularity grid on $[0,1]^2$. This allows to have some clusters, given by the gaussians, and some other points scattered on the square.

Notice that this procedure allows, with small probability, overlapping points. This is not a problem for our model, as the solvers need to be able to recognize this peculiarity and find a solution where such points are visited sequentially, since this movement has cost 0 according to equation (1).
This particular choice of data generation has been made to resemble the actual real life problem at hand: the drill is moving over an electric board making holes in preparation for its use. By looking at an actual finished electric board (figure 1) we can see that some points are naturally clustered together for certain electric components, and some other are less close one another and more distributed.

3 The Genetic Algorithm

After the use of the OPL model for homework 1, we decided to implement a heuristic for the solution of the problem described above in homework 2. Specifically, we designed a genetic algorithm, represented as a class *TSPSolver*

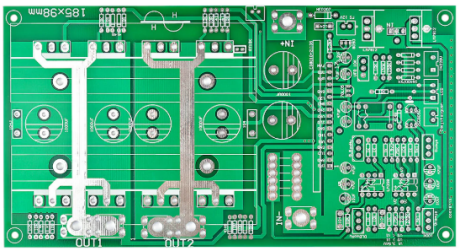

Figure 1: An electrical component. Notice the presence of clustered components, as well as scattered points

in the code attached to this report. Moreover, we decided to implement this class from scratch. This choice is not due to the lack of Python libraries for genetic algorithms (see for example DEAP or pyeasyga), but rather to a will to better understand the challenges of a complete implementation of a model, as well as to a undoubtedly enhanced learning experience.
Details about the implementation are included in the following section.

3.1 Implementation Details

3.1.1 Funcionalities

The genetic algorithm represents solutions to the TSP problem using the path representation. This means that a solution is a permutation of $\{0, \dots, n-1\}$ where the nodes are listed in order of visit. The only neighbourhood considered is the standard 2-opt neighbourhood.
The following options have been made available in the implementation (see code attached for further details, it has been made as clear and documented as possible):

- **initialization**: individuals (i.e. solutions) are initialized at random. Every generation is made up of a customized size of individuals. At initialization time it is possible to choose a fraction of individuals to substitute with their best 2-opt neighbour
- **selection of individuals**: all selection types seen in class have been made available, namely montecarlo, linear ranking and n-tournament. This selection type affects which individuals are chosen for mating.
- **mating**: two types of mating have been made available. The first is the selection of a given number of pairs of individuals. Every pair gives rise, according to the type of crossover, to one of two individuals as offspring. The second is the selection of a "mating pool": a number of individuals is chosen, and every possible couple undergoes crossover.
- **crossover**: many types of crossover are available in literature, two possible reviews are [1] and [2]. Since we found more clarity in the explanations

from [1], we implemented all the crossover types presented there. For clarity, we stick to the notation in reference in the code (which is also pretty intuitive, see the private method of the solver *TSPSolver._crossover*)

- **mutation**: with a customized probability, every new individual undergoes mutation. If mutated, every individual is substitued with a random 2-opt neighbour
- **generation replacement**: it is possible to keep the best m individuals among the old and new generation, or to remove the worst m
- **stopping criterion**: there are some stopping criteria available: the maximum number of non improving evolution steps, the time limit, the target score (i.e. stop as soon as the best individual passes a certain score threshold) and the maximum number of generations. These can also be used in combination.
- **fitness function**: here we allowed no option but the total cost of the path (including the return to the initial node 0). This point is just to highlight that for efficiency recursive evaluation of the fitness score has been used when possible (i.e for all points from the second generation onward)

As a final remark, we would like to to underline that the code has been factorized as much as possible, building appropriate private (or public, even though the difference is almost non-existent in Python) methods for *TSPSolver*. This allows modification/integration of everyone of these customizations with relative ease. The code is also available on my GitHub account.

3.1.2 Technical Details

In this section we just include some technical details about the implementations itself.
As we said, the solution representation of choice is the path representation. Since the natural translation is to use array objects to encode such solutions (or individuals), in Python one could use, for example, a simple list. We decided to go for numpy arrays. These allow for optimized vectorial operations, which come in handy at the crossover step, and are usually regarded as an efficient array tool for general use.
The actual solution flow executes as follows.

Initialization First, a *TSPSolver* instance needs to be created. At initialization moment the solver object is endowed with some attributes regarding the options included in the section above. At this stage also some other placeholder attributes are attached to the it. These include some private attributes, used for the solution loop (such as the starting time in case the time is among the sopping criteria, the generations count, etc.) as well as variables that are inferred when an actual cost matrix is given as input for the problem.
The private attibutes (which are indicated by a leading underscore in Python)

are used by the solver and are not intended for the final user. These inlcude the current generations and relative fitness values, and are continuously modified during the "evolution" step. Second, there are variables intended for posterior reference, such as the best individual (i.e. the solution) and the relative fitness. Also the cost matrix and problem size are public attributes that one can access later.

Solution After a solver object has been created, one can call the *solve* method, providing a cost matrix. Notice again, that the cost matrix has to be a square, two-dimensional, numpy array. This *solve* method is actually just a coordinator that, after saving all the variables from above (cost matrix, problem size, etc.) sequentially calls private methods for the single steps of the evolution. These are *_initialize_generation* and, while no stopping criterion is satisfied, *_evolve_generation*. The *_evolve_generation* method, in turn, calls in order the *_select_individuals* and *_mating_and_mutation* and then updates the actual generation according to the generation replacement criterion provided by the user. In the end, the code for running the solution loop is pretty simple, and looks something like this:

```
solver = TSPSolver()      # in case modify default parameters

with open('cost_matrix.pkl', 'rb') as file:
        cost_matrix = pkl.load(file)

solver.solve(cost_matrix)

print(
'The best solution found is {}, with score {}'.format(
        solver.best_individual, solver.best_fitness
        )
)
```

Notice that this way of handling problems, i.e. by using a class that stores all the information needed for and after the solution is also typical of the Python approach to machine learning, for example in scikit-learn objects or keras models. The advantage is the possibility of clear and concise sintax, automation and possibility of simple saving procedures, since the class instance incapsulates all the parameters and procedures.

3.2 Implementation Challenges

One of the main reasons for the usage of genetic algorithms is their relative speed and ease of implementation. We found that this is actually true, with just some caveats. The implementation of the single parts of the algorithm is not very challenging, especially due to the fact that the procedure is well organized into distinct sections (selection, mating, crossover, etc.) and due to the (in this case) very clear and explicative metaphor with population evolution. One downside,

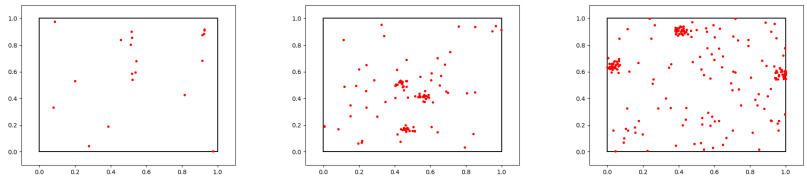

Figure 2: synthetic datasets used for the experiments. The sizes are respectively of 20, 100 and 200 nodes. Gaussian clusters points are visible among all other points.

though, is that in the moment in which one wants to allow more options, things could get messy. For this reason we report, as main difficulty encountered, the need to keep every functionality properly organized, mainly by a great deal of code factorization. As we said above, this need has in the end made the code more tidy and organized, with the possibility to add new features with relative ease.

4 Experiments

As requested for the homework, here we present a comparison of the performances of the exact OPL solver, with the network flow formulation, and the genetic algorithm. The datasets are built as follows: following the procedure from section 2.2, we produce three datasets of sizes 20, 100 and 200 nodes. In these, half of the points come from the gaussian mixture, the other half are scattered on a grid. Plots of the data are presented in figure 2.

For evaluation, we run the exact solver and the genetic algorithm 5 times each on each of the first two datasets. We just run once on the third one, allowing a longer time limit. This procedure is justified by the fact that the variability in the results from the first two datasets is low enough to show that results are consistent throughout experiments. Consequently we show confidence intervals only in figure 3, and only for the genetic algorithm, since the exact solver has almost zero variance.
Moreover, even if only with 200 points, the OPL solver requires more or less two hours to only find a feasible solution (see figure 4), making repeated experiments very time consuming. We comment the results separately in the following sections.

20 nodes dataset The genetic algorithm solver has been trained using the following parameters:

TSPSolver(init_size=100, init_type='best2opt', init_best2opt_frac=0.2, fitness='total_cost', selection='montecarlo', mating='tuples', mating_n=100, crossover='OX', mutation_prob=0.2, gen_replacement='keep_best', gen_replacement_par=100, stop-

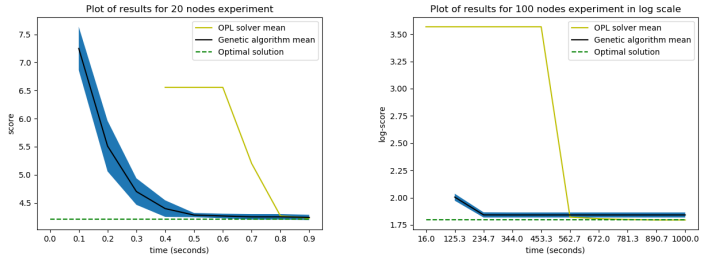

Figure 3: results of the experiments of the two smaller datasets of 20 and 100 points

ping={'time': 1000, 'not_improving_gen': 1000})

where the important points are: the use of a random 0.2 probability best-2opt improvement at initialization, initial population 100 individuals, 100 matings at every generation, 0.2 probability of mutation and fixed population size keeping the 100 best elements (see documentation or help from *TSPSolver.__init__*).
Results are presented in figure 3 (left). For this small instance the exact solution is reached in a short time from the OPL solver. It is interesting to see that also the genetic algorithm succedes in finding the optimal solution, and at every moment in time it holds a better score. This trend is not really significant though, as the small instance of the problem doesn't allow to see the expected behaviour, which is instead highlighted in the following results.

100 nodes dataset The genetic algorithm solver has been trained using the following parameters:
TSPSolver(init_size=5000, init_type='best2opt', init_best2opt_frac=0.2, fitness='total_cost', selection='montecarlo', mating='tuples', mating_n=5000, crossover='OX', mutation_prob=0.2, gen_replacement='keep_best', gen_replacement_par=5000, stopping='time': 1000, 'not_improving_gen': $+\infty$)

To allow for a fair comparison between the two methods, the same time constraint of 1000 seconds has been applied. As we can see, the size of the population, both at initialization and during evolution, has been increased to comply with the bigger problem size.
Results are presented in figure 3 (right, notice the log-scale). Even though the OPL solver seems to find a solution on the spot, it then doesn't improve for a very long time. The genetic algorithm instead finds a very good solution in a very short time, which is overcome only after the exact solver approaches (and eventually reaches) the optimal solution.

200 nodes dataset The genetic algorithm solver has been trained using the following parameters:

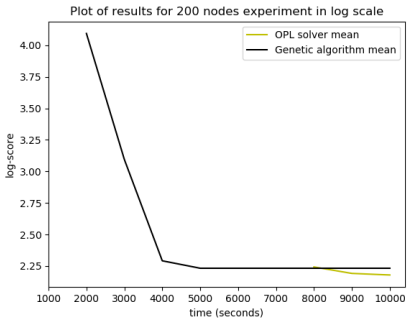

Figure 4: result of the experiment (single run) on the 200 points dataset

TSPSolver(init_size=2000, init_type='best2opt', init_best2opt_frac=0.2, fitness='total_cost', selection='montecarlo', mating='tuples', mating_n=2000, crossover='OX', mutation_prob=0.2, gen_replacement='keep_best', gen_replacement_par=2000, stopping='time': 10000, 'not_improving_gen': $+\infty$)

Again, both methods have been restricted with the same stopping criterion, in this case 10000 seconds of solution time. Here the optimal solution hasn't been found by the exact solver within the time constraints, and is therefore not available.
Results are presented in figure 4. We confirm the trend from the previous experiment, where the exact solver finds a better solution of the genetic algorithm only after a long time. Even more, it is able to find a feasible solution only after about 7500 seconds, while TSPSolver is able to find one straight away (indeed, it is already provided with a population of feasible solutions at initialization time). On the other hand, with the increase of the search space magnitude we can see that the solution found by the genetic algorithm seems to be more distant from the optimal one, that is instead asymptotically approached by the OPL solver. In fact, the reported estimated gap between the solution at time 10000 and the optimal results is 1.89%

As a final remark, we maybe need to highlight the fact that the genetic algorithm has not undergone extensive parameter optimization, both for reasons of time and because results were satisfying enough using "reasonable" choices. However, bayesian optimization techniques could be applied as the problem is suited for extensive hyperparameter tuning (for a review about bayesian optimization, see for example [3]).

References

[1] I. Gupta and A. Parashar, "Study of crossover operators in genetic algorithm for travelling salesman problem.," *International Journal of Advanced Research in Computer Science*, vol. 2, no. 4, 2011.

[2] O. Abdoun and J. Abouchabaka, "A comparative study of adaptive crossover operators for genetic algorithms to resolve the traveling salesman problem," *arXiv preprint arXiv:1203.3097*, 2012.

[3] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2015.