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

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).

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).

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* 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, aa well as to a undoubtedly enhanced learning experience.
Details about the implementation are included in the following section.

Implementation Details

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 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 *_crossover* of *TSPSolver*)
- **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.

As a final remark, we want 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 GitHub.

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: FINSH THIS SECTION WHEN ALL EXPERIMENTS HAVE BEEN DONE

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.