Practice 4

Abraham Álvarez Cruz

Department of Computer Science
School of Engineering
University of Cádiz

abraham.alvarezcruz@alum.uca.es

27th December 2023

Summary

In this report, we will address the traveling salesman problem, a well-known optimization problem. We will introduce a solution based on the *Simulated Annealing* algorithm and implemented in C++. Finally, we will present and discuss the results obtained, placing particular emphasis on the parameter configuration used for the chosen algorithm.

Contents

Figures

1 Introduction

The traveling salesman problem (TSP), known by its English acronym, is a well-known problem within the scientific community and has a long history spanning several years. Specifically, this problem was first mentioned in the 19th century by the Irish mathematician *William Rowan Hamilton* and the British mathematician *Thomas Kirkman*.

It wasn't until later, in 1930, that mathematicians from the universities of Vienna and Harvard conducted the first formal studies on the problem. *Karl Menger* was the one who defined the problem and observed the non-optimality of using the nearest neighbor heuristic. Since then, personalities from various fields and domains have explored this problem.

Its formal definition is as follows:

> For a finite set of points, for which the distances between each pair are known, the objective is to find the shortest path that connects all the points.

From the perspective of Computational Complexity, solving an instance of this problem is computationally expensive because finding the shortest path involves two tasks. The first task is the more costly one, while the second is relatively less expensive: checking all possible routes and, among them, selecting the one with the lowest cost.

This problem shares similarities with the search for a Hamiltonian path in a graph because, essentially, the underlying idea is quite similar: *finding a path that connects all points without visiting the same point multiple times*. The challenge lies in the optimization version of the problem since finding the minimum-cost path involves much more than simply finding a path.

To solve the problem, various algorithms can be employed, each offering the best possible solution within a reasonable time frame. For instance, to provide a solution, one might use a greedy algorithm. This could yield a solution within a reasonable time, with the primary drawback being that the solution is very likely not to be the one of *minimum cost*. In our case, to attempt a solution, we will use an algorithm called *Simulated Annealing*. This algorithm is inspired by the treatment of metals in metallurgical industry. These metals, to be easily molded, are exposed to very high temperatures. Under these temperatures, they become easily moldable, and as they cool down, they lose this capacity. Our algorithm will follow a similar logic, exploring the search space when it is *hotter*, and as the temperature decreases, we will gradually "exploit" the solution we have at the moment to refine it.

2 Methods

To resolve this problem, the first step we took was to learn about the structure of the algorithm and its general scheme. The following C++ code snippet shows, in broad strokes, the logic of the algorithm:

```
while (repetitions < maxRepeats){

            // Obtenemos las dos ciudades a intercambiar
            auto cities2Swap = getCities2Swap();

            // Calculamos el coste adicional
            double loss = cost(cities2Swap.first, cities2Swap.second);

            // Cambiamos la ruta si la nueva tiene un coste mejor o, en caso de ser peor, bajo una probabil
            double uniform = distribution(generator);
            double prob = (loss <= 0 or temperature <= 0) ? 0 : cool(loss);
            if (loss <= 0 || uniform < prob){

                // Versión "intercambia cacho"
                // Invertimos el cacho que comprende [inicio+1, fin)
                //std::reverse(tempSolution.begin() + cities2Swap.first + 1, tempSolution.begin() + cities2S


                // Versión "intercambia ciudad individual"
                typename GrafoP<T>::vertice tempVertex = tempSolution[cities2Swap.first];
                tempSolution[cities2Swap.first] = tempSolution[cities2Swap.second];
                tempSolution[cities2Swap.second] = tempVertex;

            }

            temperature = temperature * coolingRate;

            // Comprobamos si la distancia de la ruta es menor que la menor hasta el momento
            double actualDist = evaluate();
            if (actualDist < lowestDist) {
                repetitions = 0;
                lowestDist = actualDist;
            }
            else {
                repetitions += 1;
            }
        }
```

In broad terms, the following elements of the algorithm can be identified:

- There is a main loop that iterates a finite number of times. The maximum number of iterations is determined by a formula (which will be presented later).
- The first step in the loop is to obtain the cities to be swapped. In this case, two random cities are chosen from the entire route.
- Based on the chosen cities for exchange, the algorithm calculates the difference in

the total cost between the new route and the current route.

- The route is updated if either of the following two conditions is met: the cost of the new route is lower than that of the current route, or if it is more expensive, based on a probability.
- Finally, if the route is not updated, or the cost of the new route is worse than the best one so far, the "repetitions" counter is incremented (this helps prevent the algorithm from indefinitely searching for worse or non-improving solutions).

For the generation of the initial solution, I currently use one of two techniques: a random route and a route returned by a greedy algorithm. In the next section, I will discuss the results of each version. The evaluation function used is simply the summation of the total cost of our route.

As for the cost function, it involves evaluating our current solution and the new solution, calculating their difference to determine whether there is an increase (worse) or a decrease (better) in the total cost.

Regarding the "successor" function, we currently select two random cities within our route and swap them. The probability distribution function used in the code is the *Boltzmann* function, a function commonly used for calculating probabilities in physical models. Finally, the cooling function used to update the temperature has been linear.

Once the entire code was developed, we chose a testbed to evaluate our solution. We used a set of *.tsp* files from the TSPLIB library. Each of these files contains pairs of coordinates indicating the position of each city. To work conveniently, our program first reads each *.tsp* file along with the solution proposed by the library (the *.opt.tour* files containing the sequence of cities to be visited in order).

We create a graph with as many nodes as there are cities on the map. For each pair of cities, using their Cartesian coordinates, we calculate the *Euclidean distance* to determine the cost of traveling from one city to the other. Once we have our weighted graph and an *optimal* solution, we proceed to run our algorithm for each of the input graphs a certain number of times (in our case, *25*), aiming to obtain a better estimate of the average cost of the solution found by the algorithm (remember that it is a metaheuristic and probabilistic algorithm, meaning that each execution could yield a different result from the previous one).

Finally, with the solution returned by our algorithm and a baseline solution, we calculate the difference between the two.

3 Results and discussion

In Figure 1, I have presented the average execution times of the algorithm. It can be observed that as the number of cities increases, the execution time grows polynomially. Although the time increases significantly as the number of cities grows, it is always upper-bounded by a polynomial, helping to keep the solution search within an acceptable range. If, on the other hand, we were using an inefficient algorithm (non-polynomial), such as a brute-force algorithm, for not very large input sizes, the execution time would get increased to unsustainable levels (e.g., with 450 cities).

In the next two figures, Figures 2 and 3, we can see the absolute errors committed in each of the executions. It can be observed that the absolute error committed by the two versions we have used (initialization through the greedy algorithm and random permutation) yields similar results, with the executions that used the greedy algorithm for the initial solution being slightly better (though insignificant).

In general terms, it can be observed that as the number of cities increases, there is no significant increase in the error committed by the solutions returned by the algorithm. This highlights the utility of these algorithms in finding solutions close to the global optimum.

Continuing with the analysis, looking at Figures 4 and 5, we can observe a linear trend in the relative terms. However, there appears to be an upper bound around 300

It is entirely understandable that as the number of cities increases, the error with respect to the reference solution also increases. This is because the search space scales very rapidly when adding more cities. Beyond a certain point, the number of possible combinations that need to be investigated increases exponentially, significantly complicating the task of finding the path with the lowest cost. This phenomenon is characteristic of combinatorial problems like the Traveling Salesman Problem.

Finally, I will comment on the configuration used by our algorithm implementation. [1].

[1] **NOTE: Relative errors are calculated based on reference solutions, with these being 100%. In other words, a solution with a relative cost of 150% differs from the reference solution by 50% of its cost.**

3.1 About de implementation

After conducting an exhaustive search regarding the best configuration for this algorithm on the internet, I came across a series of articles discussing the topic. In the report [1], the author conducted a study on the advantages of using a fixed temperature over time. The issue with this approach is that the exploration-exploitation dilemma remains constant, and a potential solution might be adversely affected by giving too much probability to exploration in the later stages of the algorithm execution. That's why I chose to modify the temperature over time.

Additionally, in another technical report I read, they used different formulas for both the calculation of the initial temperature and its update throughout the execution. Ultimately, I opted to use the formula:

$$initialTemperature = 1 - 100^{-\log_{10}(n)+1} \tag{1}$$

I extracted this formula from the technical report [2], where the author mentions that it provided the best results in their executions. As the cooling function, I used linear cooling with a value of α = "average distance between cities." Again, I chose these values based on the results from the aforementioned report.

The probability distribution function I chose to select a worse solution than the *i-th* solution was the *Boltzmann* function. In general, this function is commonly used in standard implementations of the algorithm and works well in most cases. The *Boltzmann* function is defined as:

$$p = e^{-\frac{E}{kT}} \tag{2}$$

Where E is the energy of the current state (in our case, the energy of the current state is the cost difference between the new potential solution and the one we have so far), k is a constant value (also called the *Boltzmann constant*. In our case, $k = 1$), and finally, T refers to the thermodynamic temperature (in our case, it will be the value of the temperature).

Finally, for the generation of a new solution, I initially chose two random cities and inverted the segment between these two cities. I also tried randomly swapping two cities without any inversion. After executing both options, there was no noticeable difference between them, so I opted to stick with randomly swapping two cities. However, in the C++ code, the other version is commented (see the *cost()* function in the *annealing.cpp* file, around line 170).

All these decisions were made based on the articles and technical reports found in Section 4.

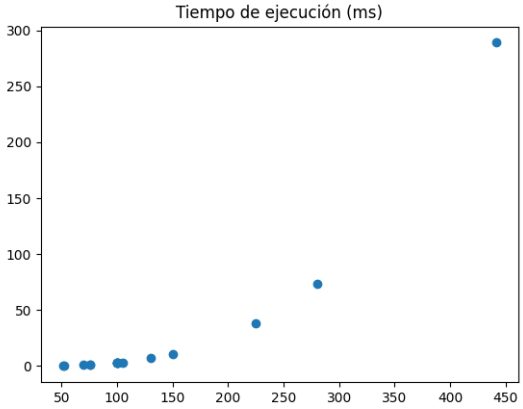

Figure 1: Execution time vs Nº if cities.

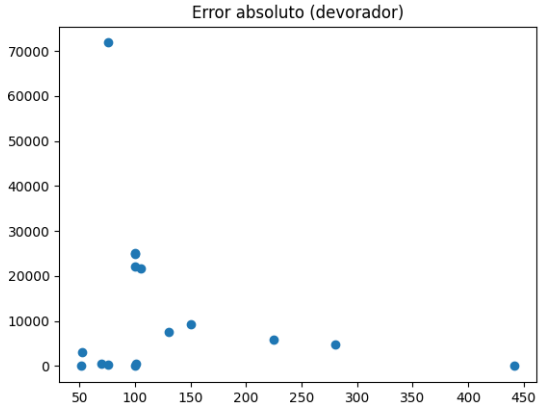

Figure 2: Absolute error (greedy initialization).

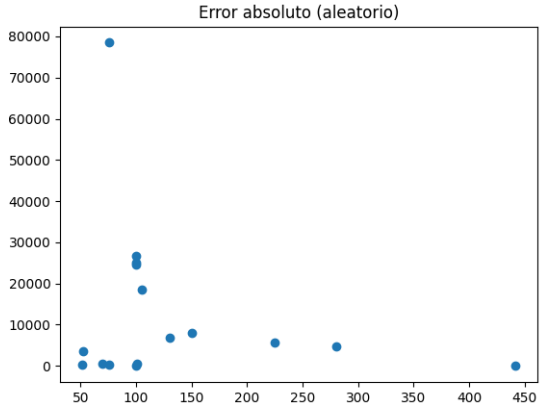

Figure 3: Absolute error (random inicialization).

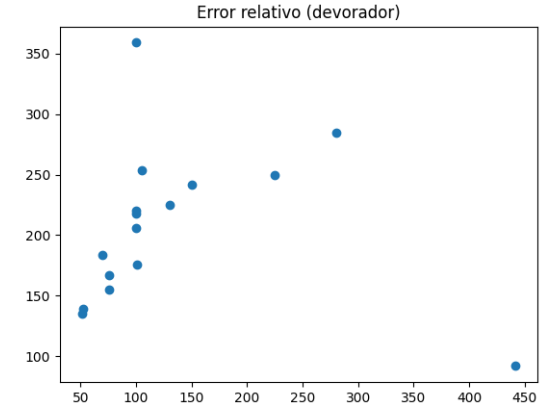

Figure 4: Relative error (greedy inicialization).

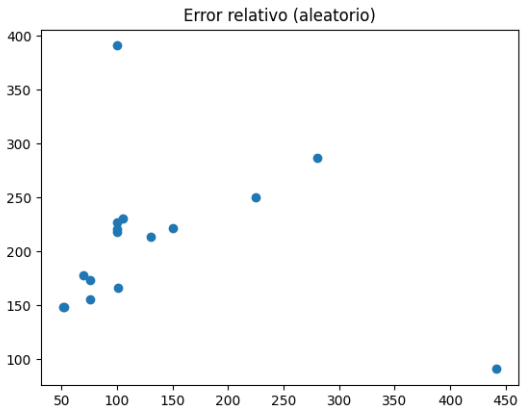

Figure 5: Relative error (random inicialization).

4 Conclusions

Metaheuristic algorithms are widely used today for their ability to provide pseudo-optimal solutions to problems that are too costly to be processed by a heuristic algorithm that returns an optimal solution. There are many such algorithms, each approaching problems from a different perspective. Nature-inspired algorithms, for example, aim to find solutions by mimicking the physical behavior of real-world elements, such as ant colonies. Figure 6 provides a glimpse of the variety of existing metaheuristic algorithms, each employing a unique approach.

Personally, metaheuristic algorithms are among my favorites, and I have spent a considerable amount of time reading about them and coding some. The ones that fascinate me the most for their remarkable problem-solving capabilities are those inspired by nature with colony behaviors, such as HHO (Harry's Hawk's Optimization), Grey Wolf, Ant Colony, Bees Colony, Polar Bear, and many others.

Regarding their use in the traveling salesman problem, I believe they are a very suitable and efficient response for finding a pseudo-optimal solution due to their ability to achieve competent results in acceptable time frames. It's important to consider that without using algorithms like these, finding a solution that is not very costly could take a considerable amount of time, along with the associated economic cost.

In summary, metaheuristic algorithms are a powerful tool that we can always leverage and should keep in mind for various applications.

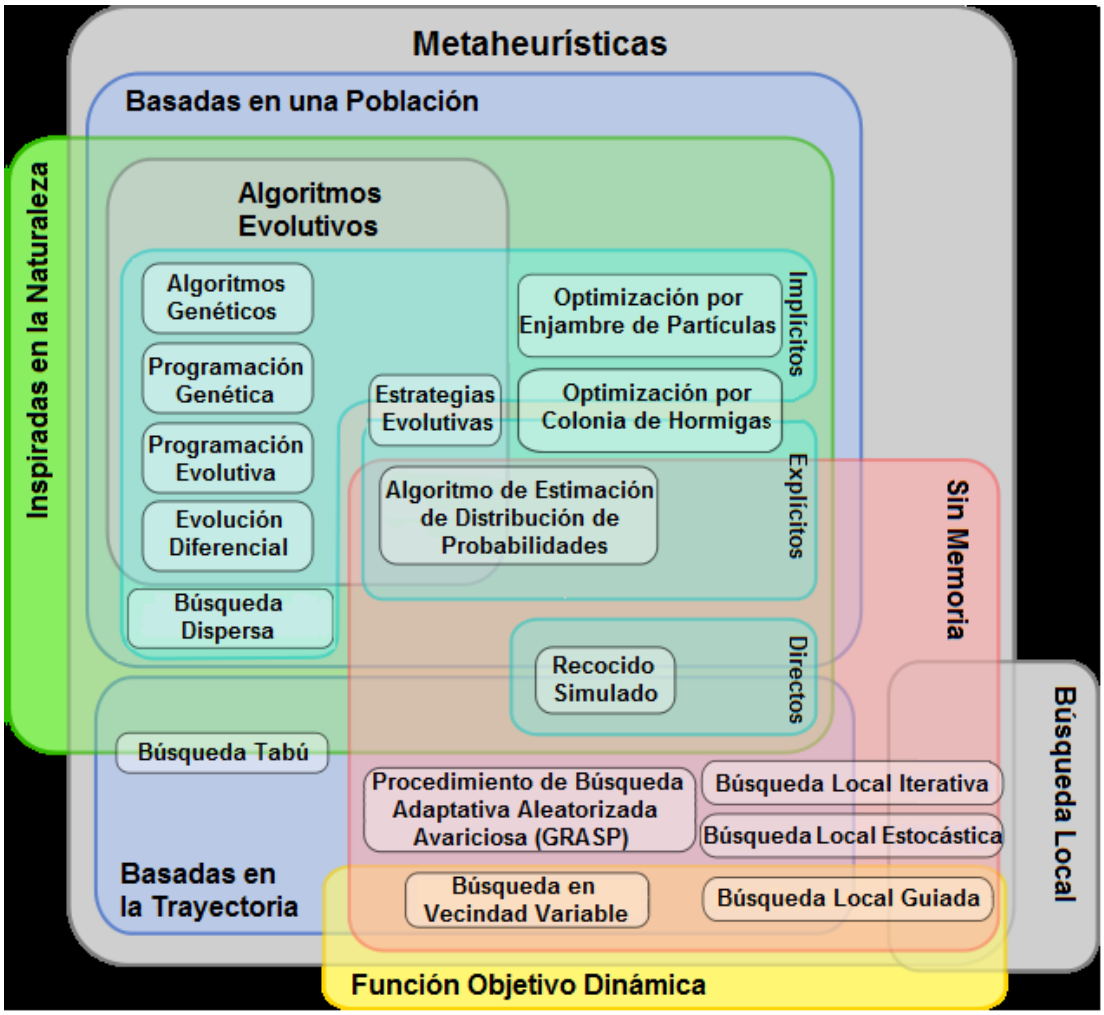

Figure 6: Metaheuristic algorithms.

Bibliography

[1] Vahid Majazi Dalfard. 'Adjustment of the primitive parameters of the simulated annealing heuristic'. In: *Indian Journal of Science and Technology* (2011).

[2] Mark James Fielding. *Simulated annealing with an optimal fixed temperature*. Tech. rep. University of Wollongong, 200.

[3] Oded Goldreich. *P, NP, and NP-completeness. The basics of computational complexity*. Cambridge University Press, 2010.

[4] Gary Sun. *Parameterising Simulated Annealing for the Travelling Salesman Problem*. Tech. rep. University of New South Wales, 2021.

[5] Dennis Weyland. 'Simulated annealing, its parameter settings and the longest common subsequence problem'. In: July 2008, pp. 803–810. DOI: 10.1145/1389095.1389253.