Project II: Solving a Travel Salesman instance using a Genetic Algorithm

Degree in Data Science (Optimization)

Universitat Politécnica de Valencia

Álvaro Faubel Sanchis, Ignacio Cano Navarro, Daniel Goig Martínez, Angel Langdon Villamayor

***Abstract* - The traveling salesman problem is a very well-known problem, and it has a lot of applications in the real world. In this paper, we aim to solve an instance of the TSP with 52 spots placed on the city of Berlin. In order to solve this instance of the TSP, we have implemented a wide variety of genetic algorithms (GA), changing crossover operators, mutation operators and so on. After that, we tested every algorithm proposed in a fair environment giving each algorithm two hundred iterations and prematurely stopping them if in twenty iterations the solutions didn't get any better. The best algorithm used the sequential constructive crossover as crossover operator, SBM as mutation operator, population size of 200 and exponential ranking as the parent selection. The results achieved were almost optimal with a mean distance of 7663.56 and a mean time of 1.86 seconds after less than 200 iterations and tested with 25 random initial populations.**

I. Introduction

The traveling salesman problem (TSP) is a typical example of a very hard combinatorial optimization problem. The problem is to find the shortest tour that passes through each vertex in a given graph exactly once, what is known as a Hamiltonian cycle.

Apart from its theoretical approach, TSP is widely used as a model in many fields such as vehicle routing, scheduling problems, design of integrated circuits, physical mapping problems, constructing phylogenetic trees, machine flow shop scheduling and so on. Hence, solving TSP has significant practical implications.

In this case, our company wants to design a route around the city of Berlin that passes through 52 key spots exactly once, in order to reduce fuel and associated costs. As we can see, this problem is a TSP instance.

In this case we are assuming that we are on the symmetrical TSP (STSP). That is, the distances between vertex are symmetrical. In other words, supposing the distance between vertex a and b is x_{ab}, then, we have $x_{ab} = x_{ba}$.

Some of the algorithms used to solve the STSP are exact algorithms, while the others are approximate algorithms which will try but not necessarily will get the optimal solution. In this paper, we will be using genetic algorithms (GA). The GA is a population-based approach for heuristic search in optimization problems, and it has been applied to a wide variety of optimization problems.

To solve the proposed problem, we will be implementing a wide variety of crossover operators, mutation operators, offspring selection and so on (these will be all explained in the model section). In order to see which of these combinations gives us the best solutions in the least time possible, we will be doing some experiments with all the implementations.

This paper will be divided in the next sections: 'Related work', 'Data', 'Genetic Algorithm Proposed', 'Implementation', 'Experiments' and 'Conclusions'.

II. Related work

A. Multi-offspring genetic algorithm and its application to the traveling salesman problem [1]

In this paper, the authors compare the results of applying the multi-offspring genetic algorithm (MO-GA) to the symmetric travelers salesman problem (TSP) to the results obtained using the basic genetic algorithm (BGA).

The difference between MO-GA and BGA lies in the way the crossover and mutation operations are performed. In the existing literature on BGA, the crossover operation is performed with two parents generating two offspring, whereas two parents generate more than two offspring in MO-GA. To keep the population size unchanged, exceptional individuals are selected and maintained at the end of each generation.Thanks to this difference, MO-GA increases the number of excellent individuals generated, making the population more competitive.

In the results section, they compare the results of both algorithms in six well-known TSP instances. The results are outstanding for MO-GA as they beat BGA in both the value of the solution (optimal in most of the cases) and the running time in each of the six instances tested.

This paper, even though we won't be implementing MO-GA in our work, is useful as we get to see that maybe BGA is not the best genetic algorithm and that there are a lot more other choices in the world of genetic algorithms.

B. *A new genetic algorithm for the asymmetric traveling salesman problem* [2]

In this paper, the authors explain a new genetic algorithm (GA) for the asymmetric traveling salesman problem (ATSP). In short, the ATSP is different from the TSP in the fact that the distance between the node i and the node j is not the same as the distance between the node j and the node i.

The idea is to test the Edge assembly crossover operator (EAX), that has been widely used on the symmetric TSP, on the ATSP with all the enhancements that were implemented to the operator in the following years since its creation (1997). In order to check the efficiency of the GA, it was tested on 153 ATSP instances.

The results are quite good as they even outperform other well-known algorithms for the ATSP. This algorithm then, can be used to solve real-world problems and also be used as comparison for future proposals.

This paper is quite useful as it presents a new crossover operator that achieves good results in both the TSP and the ATSP. Also, it gives us an idea of how to create the experiments for our own work.

C. *An efficient genetic algorithm for the traveling salesman problem with precedence constraints* [3]

In this paper, the authors will implement a genetic algorithm (GA) in order to improve the results of the well-known algorithms of the traveling salesman problem with precedence constraints (TSPPC). The main difference between the TSP and the TSPPC is that in the TSPPC the order of the nodes visited matters, making the problem even more difficult to solve in reasonable time. Modifications have to be made to the traditional implementations of GA of the TSP in order to handle the restriction of precedence. The way they do this is by doing a topological sort that won't let the GA generate infeasible solutions for the TSPCC. Also, the authors will implement a new crossover operator to efficiently search the solution's space.

The proposed GA approach gets superior performance compared to other algorithms. For small and medium size problems, obtain optimal solutions. On a larger size problem, the proposed GA approach generated best solutions.

This paper has been useful to further understand the difficulties added to the original TSP and see which strategies can be implemented. However, the paper doesn't help to reproduce the results and the tables and figures are in general not very useful, so the results have to be analyzed carefully.

D. *Deep Reinforcement Learning for Traveling Salesman Problem with Time Windows and Rejections* [4]

This paper applies deep reinforcement learning (DRL) to a variant of the TSP, the Traveling Salesman Problem with Time Windows and Rejections (TSPTWR), a more realistic approach to real-world problems.

In TSPTWR, xi is the 2-D coordinate of node i and the time window/deadline of visiting node i. The goals are twofold, which are minimizing the rejection rate R = (rejected nodes)/(total nodes) and meanwhile minimizing the total length of the tour L.

The DRL implemented follows the self-attention based encoder-decoder perspective. However, it adds an explainable heuristic helper function that postprocesses the output sequence from the decoder. The helper function allows the framework to assess the solution quality (reward) correctly according to the setting of TSPTWR.

Based on the results, the running time for different problem sizes is compared to one of the most used algorithms for this problem, the Tabu Search. While the running time of the tabu search considerably increases with the size of the problem, DRL running time increases significantly slower. The overall results of the DRL are way better than the tabu search in less time.

In this paper, we have seen another approach to the TSP (in this case the TSPTWR) and using another method other than Genetic Algorithms.

III. Data acquisition

The data comes from the TSPLib, a well-known library specialized in providing instances of the symmetric TSP. As we have previously said, the data consists of 52 nodes placed in the city of Berlin, and for each node, we have available the distance between the current node and the rest.

IV. Genetic Algorithm proposed

We will represent the feasible solution as a vector of length N, where N is the total number of nodes. This vector must contain all the nodes (locations) and therefore there can be no repeated nodes. So we represent the solution as a permutation of size N (N=52 in the Berlin-52 instance).

As we are looking for a solution that returns to the initial node (cycle), the final path will consist of going through the permutation from the first element to the last and from the last to the first, thus completing the requested path.

Initialization

Regarding the genetic algorithm we have proposed, we will first discuss the initialization of the problem.

First, since the initial problem presents a fully connected graph, all possible permutations of nodes are valid as a solution and could fit perfectly into the initial population.

That is why we create the first instance of the problem, randomly mixing the order of the list elements with the nodes/cities of the problem. This is done with the *create_random_path()* function.

To create the initial populations, we use the *create_informed_random_path()* function to generate random but informed individuals. This function follows the nearest neighbor heuristic to generate an instance. It consists in constructing a low-cost Hamiltonian cycle through a greedy solution based on the vertex closest to a given one.

An arbitrary initial node is chosen (not all solutions will start with the same vertex). From this, the nearest node is searched for and inserted into the cycle. When a node is inserted into the path, it is marked as a visited node. The nearest node to the last one inserted in the path will continue to be searched until all the nodes in the set have been visited. At this point the algorithm closes and returns to the initial node.

So to generate a solution of size *n*, it will be enough to call the previous function *n* times. This is done by the *create_random_population()* function.

It should be noted that with this guided initialization of each individual, we seek to create a first initial population as close as possible to the optimal solution, in order to achieve better results with the execution of the GA in fewer iterations.

Another point to take into account in the population is the presence of duplicates. According to the way we have designed the initialization, it could happen that there are repeated individuals, because they are equal to others or because they follow the same path. However, we have not considered this to be a problem, and have preferred the presence of duplicates in our population to the temporary cost of removing them. It would be interesting to see what effect this has in future studies.

Fitness

Fitness is the evaluation of the evaluation function and indicates how good the individual (i.e. the solution to the problem) is with respect to the others.

We first calculated the distance of a path with the function *get_path_distance()*. Since our problem is a minimization problem, the shorter the distance the better the solution is. Therefore, we have created a function called *get_fitness()* proportional to the total distance of a path, which will take a larger value the smaller the total distance of a solution. In this way we use fitness for the algorithm (to identify the best solutions such as those with the highest fitness), even though some operators use distance.

Mutation Operators

A mutation operator is responsible for increasing or decreasing the search space in a genetic algorithm and for providing some genetic variability of individuals. These are used as a perturbation mechanism, in order to better explore the solution space and escape from local optima.

In our genetic algorithm we have used 6 mutation operators: *Simple swap*, *Reverse sequence mutation* ('Inversion'), *Random gene inserted next to nearest neighbor mutation* ('RGIBNNM'), *Worst left and right gene with random gene mutation* ('WLRGWRGM'), *Worst gene with random gene mutation* ('WGWRGM') and *Inverse random gene inserted next to nearest neighbor mutation* ('IRGIBNNM').

In *simple swapping*, an offspring is generated by swapping two random genes on a chromosome.

In *reverse sequence mutation or inversion*, two genes are randomly selected and the order of content between the two positions is reversed.

In the *random gene mutation inserted next to the nearest neighbor mutation* [5] the worst city is selected randomly, i.e. the concept of worst city here is not defined, it is just a random city, and is not based on its negative contribution to the fitness of the chromosome. Since the worst city is randomly selected, we increase the diversity in the search space.

On the other hand, in the *left and right worst gene method with a random genetic mutation* [5], the worst gene is the one with the maximum total distance between that gene and its two neighbors, the one on the left and the one on the right. Considering both distances (left and right) may be more informative than considering only a left or right distance.

To perform the *worst gene with random gene mutation* [5], we have to search for the 'worst' gene on the chromosome from index 0 to L-1, where L is the length of the chromosome. The worst gene varies depending on the definition of the worst for each problem. The worst gene is the point on a given chromosome that contributes the most to increasing the cost of that chromosome (solution).

Finally, we used a hybrid mutation called *Inverse random gene inserted beside nearest neighbor mutation* (IRGIBNNM) [6]. In this mutation we combined two mutation operators, inversion mutation and RGIBNNNM.

IRGIBNNNM initially applies the inversion mutation to an individual, and then RGIBNNM (*Random gene inserted beside nearest neighbor mutation*, explained above) is applied to the resulting individual. Thus, the new offspring benefit from the features of both mutations, trying to improve the performance of both mutations, increasing the diversity in the search space, and therefore, providing better results.

Finally, we applied the *select the best mutation* (SBM) [5] strategy. In this strategy we apply several mutation operators simultaneously to the same individual, and from each mutation we keep the 'best' offspring.

Cross Operators

Crossover operators act on pairs of individuals and usually produce a pair of individuals that combine characteristics of the parents. In our genetic algorithm we have used 3 crossover operators: *Cut and crossfill*, *Sequential constructive crossover* and *Edge recombination crossover*. All our crossover operators create a new pair of individuals by combining parts of the two parent chromosomes. This is a prerequisite for then being able to apply a diversity maintenance method, such as Determinis Crowding.

In the *Cut and Crossfill* crossover operator, a position is randomly selected by splitting the chromosomes into two parts. The first segment of the offspring is a copy of the genes from the parent to the corresponding offspring while the second segment of each offspring is filled with genes from the other parent skipping those already contained.

In the *Sequential constructive crossover* [7], a point, called the crossover site, is randomly selected and the information is exchanged after the crossover site of the two parent strings, thus creating two new children.

The sequential constructive crossover (SCX) operator constructs an offspring using the best edges based on their values present in the parent structure. This method does not depend only on the structure of the parents; sometimes it introduces into the offspring new, but good, edges that are not even present in the current population.

In scientific article [7], node 1 is chosen as the crossover point. Since this is a deterministic and guided method, two equal children are formed. As we want to generate two children that are different, we choose the first elements of each parent. In this way we achieve that the two generated children are different from each other.

The last crossover operator we have used, *Edge recombination crossover*, tries to introduce as few paths as possible. In problems like ours (traveling salesman) the introduction of a missing edge between two nodes is usually very negative for the fitness of a chromosome. The idea is to use as many edges, or node connections, as possible to generate children. Edge recombination is usually more efficient than PMX (partially mapped crossover) and ordered crossover, but it usually takes more computational time.

This method has been adapted from the notes of lesson 5 of the course made by Victor Sanchez Anguix [8]. In the notes a single child is created from two parents. As the initial element of this child, one of the two initial elements of the parents is chosen randomly. As we want to generate two new chromosomes, we adapt the method and choose the first two elements of the parents as initial genes of the offspring.

Based on the scientific article [5] where I explained the selection of the best mutation operator we have considered doing the same for the crossover operators using the *SBC()* (Select best cross) function.

Parent selection

Once the mutation and crossover operations have been carried out, the next step will be to select the fittest individuals, which will be the parents of the next generation. To carry out this task we have implemented 3 methods: selection proportional to fitness, selection by linear ranking and by exponential ranking.

In the first one, *selection proportional to fitness*, each individual has, as its name indicates, a probability to be selected proportional to fitness. Individuals with a very good fitness with respect to the rest of the population tend to take over the entire population very quickly, leaving little room for the algorithm to search in other areas of the solution space where better solutions may exist. This problem is known as premature convergence.

When all fitness values are very close, there is almost no selection pressure and the method is similar to selecting uniformly. Once the worst individuals are discarded, the mean fitness of the population converges very slowly.

On the other hand, in *linear ranking selection*, we use the ranking (position according to fitness) of an individual in the population instead of using fitness directly as a guide to select the individual. From one individual to the next, the probability of selection always increases by the same amount (hence linear).
This method depends directly on the value of the hyperparameter s. We have chosen $s = 2$ so that the individual with better fitness accumulates more probability.

The last of the 3 methods used, *selection by exponential ranking* is similar to selection by linear ranking, but the best individuals are allowed to accumulate more probability and the worst less. The difference in probability of selection between two contiguous individuals grows exponentially.

This method also depends on the value of a hyperparameter, c. We have chosen $c = 0.2$, a value close to 0, to have a more exponential behavior and thus select individuals with better fitness.

We have also consulted these three methods from the notes of topic 5 of the course by Víctor Sánchez Anguix [8].

Diversity maintenance

To avoid premature convergence and ensure that the algorithm converges quickly to local minima without exploring areas where the global minimum may be, we have implemented diversity maintenance, seeking a balance between exploitation and exploration.

For diversity maintenance we have used Deterministic Crowding. This technique matches 2 parents and 2 offspring by minimizing some measure of distance and uses the deterministic acceptance rule of always choosing the best individual in each parent-offspring pair through a tournament. This tournament is conducted independently of the parents chosen and is done between the best parent-child pairs. To conduct this tournament we set a probability of choice at 0.9. In this way, almost always (in 90% of the cases) the best parents/children are chosen leaving a small probability to explore other areas. For this reason we have not implemented population replacement. It is implicit in the algorithm.

Since offspring tend to compete for survival with their most similar parents, a member of the population is replaced by a member similar to it, thus conserving the area of the search space spanned.

Since our problem is an adjacency problem and our solutions are permutations we have used the Hamming distance metric to compare children to parents. This distance calculates the number of positions in which the characters differ from one chromosome to another. It should be noted that we consider as distance 0, those instances that present the same path although the distribution of the nodes in the vector is different.

Final algorithm

To gather all the "ingredients" explained above in a single algorithm, we have created the function *genetic_algorithm()* that structures the GA according to the chosen operators and performs its execution.

Thus, the function receives as parameters the population size, the parent selection method and the desired crossover and mutation operators, as well as a seed in case you want to replicate the run. It should also be noted that the crossover probability, as well as the mutation probability has been set at 0.9 and 0.8 respectively, with the intention of not always carrying out such operations. Furthermore, the number of parents selected per iteration is equal to half the chosen population size. Nevertheless, it would have been interesting to test different values of these hyperparameters.

The algorithm consists first of initializing the first population with the desired size and then starting the main loop.

So we start iterating by selecting the n parents with one of the explained techniques. Once the parents are selected, they are crossed (with probability *p_cross*) and generate two new children that can be mutated (with probability *p_mut*). At this point, the parents are paired with their closest offspring and the tournament is held (with probability *p_tour*) that will decide which of the members of each pair survives in the next population.

With the population already updated, the best solution is evaluated and the stopping criteria are checked. If the maximum 200 iterations have been reached or the current best solution has not been improved in 20 consecutive iterations the algorithm terminates and returns the best solution so far, otherwise the parents are re-selected and iteration continues.

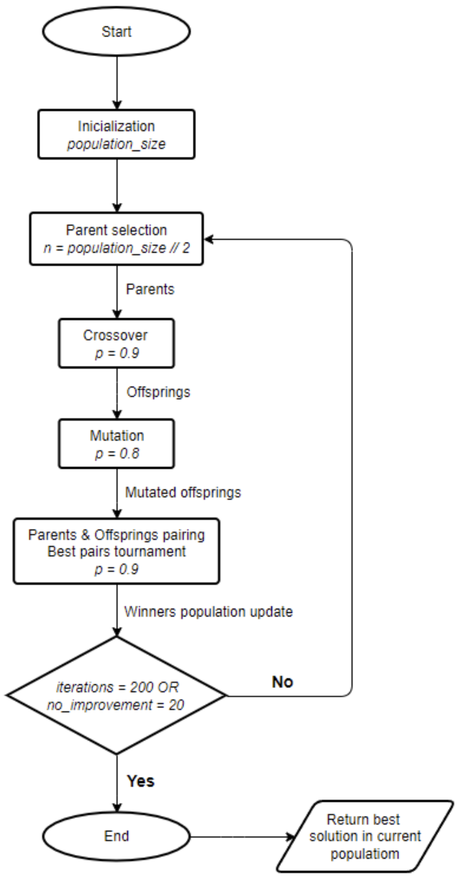

Fig. 1. Proposed Genetic Algorithm diagram workflow

V. IMPLEMENTATION

Given the complexity of the problem (52 nodes/cities) it has been solved using a computerized environment. For that, we have used *Python 3* for implementing the genetic algorithm. as well as, *numpy* and *pandas* (matrix/tables libraries) for

Thus, the *requests* library has been used to download the data, as well as *numpy*, *math* and *random* for the calculations and implementation of the operators.

In addition, in the experimentation, *itertools* is used to test different genetic algorithm configurations, *matplotlib* to graph the results obtained, as well as *scipy* and *statsmodels* for the statistical verification of the results.
All the source code has been developed using a *Jupyter Notebook* (.ipynb)[1]. It is structured in the following section and subsections:

TABLE I. Source code structure

Section	***Subsection***	***Description***
Preprocessing	Data loading	Data downloading and precomputation of all node distances
Modeling	Initialization	Functions to create an individual (path) and the population
	Fitness function	Fitness function based on path distance
	Mutation operators	Implementation of multiple complex mutation operators
	Cross operators	Implementation of three complex cross operators
	Parent selection	Implementation of three parent selection ranking algorithms
	Diversity maintenance	Implementation of hamming distance and paring tournament
	Final algorithm	Complete genetic algorithm function with multiple configuration parameters
	Solution	Function to pretty print a solution (path)
	Example of use	Practical example of a complete execution of the genetic algorithm
Experimentation	Experimentation design	Function to create a genetic algorithm experiment (uses Grid Search)
	Best results	Display of the best obtained results (includes optimal solution)
	Results by factor	Display of the results based on different factors
	Factor influence	Different statistical tests to measure influence of factors

VI. Experiments

In the experiments section, we will be covering all the tests done to the proposed GA, as well as presenting the results provided and selecting the best ones.

Because of the heavy time consuming computation, it was decided to solve for each possible combination of parent selection, crossover operator, mutation operator and population size only 25 times the proposed STSP (as the operators used are not deterministic, each run will be different due to randomization). The initial population of each of the runs that the problem is solved is the same in every test thanks to using the same seed, that is, we used the same 25 seeds for each experiment done. Also, we have restricted the number of iterations of the GA when solving the STSP to 200 and in addition, the algorithm will stop if the solution didn't improve after 20 iterations.

In total, there were 252 possible combinations. Because of the size, we are only going to provide the best ones by population size, that is, the best results for population size equal to 50, 100 and 200. If the reader would like to have a look at the full results they are available in a *.csv* file in a Google Drive's folder[2]. In the results we have considered both the combination that provided the best solution (in terms of distance) and also the solution that had the best balance between the mean value of the solution and the mean time after 25 runs.

TABLE II. Best solutions obtained

N Pop	Crossover	Mutation	Parent selection	Avg Time	Avg Distance
50	cut and crossfill	SBM	select_exp _ranking	0.35	7818.36
50	SBC	SBM	select_exp _ranking	1.84	7802.36
100	cut and crossfill	SBM	select_exp _ranking	0.66	7715.08
100	SBC	SBM	select_exp _ranking	3.84	7709.96
200	sequential constructive	SBM	select_exp _ranking	1.86	7663.56

In order to be able to measure the quality of the distance obtained, we will compare it with the known optimal solution of the STSP Berlin 52 instance, which is 7542. We can observe that the average solution of the best model *(Population size* equal to 200, with *sequential constructive crossover*, *SBM* and *exponential ranking*) achieves 7663.56 with a maximum of 200 iterations (probably less taking into account that after 20 iterations if the solution has not improved, the algorithm will stop). The average time was 1.86 seconds per execution, which is not the fastest of the best solutions, but surprisingly it is not the slowest (3.84) . We can also highlight that for this configuration, 7 of the 25 repetitions have reached the optimal solution, which is equivalent to almost 30%.

For the other population sizes (50, 100), we struggled to decide which option was the best, because while it is true that the algorithms with SBC crossover operator had the lowest mean distance, they also had the highest mean time running.

That's why we decided to include the solutions that we think are more balanced, in which by giving up a bit of distance, we get much better running time (up to 6 times lower). This can be achieved by changing the SBC crossover operator for the cut and crossfill operator and keeping the rest of parameters the same.

[1] https://drive.google.com/file/d/1GBl1YxZUcDqe14VlSgsMHWDFMbyuQn3S/view?usp=sharing

[2] https://drive.google.com/file/d/1d8_BpWo7Ri0kb0aULN01BO6PS1z-Fq3Q/view?usp=sharing

In relation to the mentioned before, it must be pointed out that in every solution in Table II, the mutation operator and parent selection were always the same, *SBM* and *exponential ranking* respectively.

In the following part of the experimentation part, we tested the different parameters in order to know if there really is any statistical difference between the values of the parameters for both the distance and times.

The first step is to check if the data distribution for the different parameters is normal, in order to decide which type of significance analysis to perform.

Given the fact that the number of samples is more than 50, the Kolgomorov-Smirnov (KS) test was chosen to check whether or not the data followed a normal distribution.

Due to the huge number of KS tests (18 +18 for both times and distance), we are going to summarize and state that every single KS test confirmed that the data didn't follow a normal distribution.

Even though we are forced to use a non-parametric test such as Kruskal-Wallis test, we conducted a Levene's test to check homoscedasticity:

TABLE III. Levene's test results for time

Parameter	***p value***	***Homoscedasticity***
Crossover operator	5e-13 < 0.05	No
Mutation operator	0.00051 < 0.05	No
Parent selection	0.18 > 0.05	Yes
Population size	4e-21 < 0.05	No

TABLE IV. Levene's test results for distance

Parameter	***p value***	***Homoscedasticity***
Crossover operator	0.61 > 0.05	Yes
Mutation operator	0.036 < 0.05	No
Parent selection	0.046 < 0.05	No
Population size	0.00095 < 0.05	No

In the Levene's test we see that some parameters actually have equal variance between groups. However, we are still forced to conduct a Kruskal-Wallis test due to the lack of normality in both distance and time data.

The Kruskal-Wallis test is a rank-based nonparametric test that can be used to determine if there are statistically significant differences between two or more groups of an independent variable and as it is considered the non-parametric alternative of the One-way ANOVA [9], we used it to check whether there were difference between the groups of our four parameters or not (each one separately).

It is important to note that the Kruskal-Wallis test is an omnibus test and won't tell us which groups of our independent variable are significantly different from each other.

TABLE V. Kruskal-Wallis results for time

Parameter	***p value***	***Equal medians***
Crossover operator	1.4e-31 < 0.05	No
Mutation operator	0.036 < 0.05	No
Parent selection	0.62 > 0.05	Yes
Population size	9.4e-14 < 0.05	No

TABLE VI. Kruskal-Wallis results for distance

Parameter	***p value***	***Equal medians***
Crossover operator	0.96 > 0.05	Yes
Mutation operator	1.4e-06 < 0.05	No
Parent selection	3.4e-34 < 0.05	No
Population size	5.3e-08 < 0.05	No

In the Kruskal-Wallis results for distance, we observe that it doesn't matter which crossover operator we choose, the difference between distances won't be significantly different from each other, so we should choose the crossover operator based on the running time in which there are significant differences. That means that using SBC isn't the best option as it is the slowest one.

Another surprising finding is that the chosen parent selection operator should be the one that gets better distance as it won't affect the running time because there are not any significant differences between the running time of the 3 operators tested. This helps a lot, because in the best results the *exponential ranking* always appears and as the running time is statistically the same for every selection operator, we should always choose the *exponential ranking operator*.

For the population size, a post-hoc analysis (Conover's test) will be conducted (we won't show the results of mutation, crossover and parent selection because it would take up too much space and also they are available at the *jupyter notebook*).

TABLE VII. Conover's test results for time

Population size	***50***	***100***	***200***
50	1	2.27e-17	5.19e-16
100	2.27e-17	1	0.00023
200	5.19e-16	0.00023	1

TABLE VI. Conover's test results for distance

Population size	***50***	***100***	***200***
50	1	0.000235	1.125e-08
100	0.000235	1	0.028
200	1.125e-08	0.028	1

Pairwise comparisons show that we may reject the null hypothesis ($p < 0.05$) for each pair of population sizes and conclude that all groups differ in both times and distances. That is, population size equal to 200 means better distance, while also making the time slower (statement based on experimentation and confirmed thanks to Conover's test). So, depending on what we prefer, running time or better results we should choose one or another, as in the case of the mutation operator (not shown in this paper due to the lack of space but available in the jupyter notebook).

To conclude this section, we would like to select both the fastest (without giving up too much on the quality of the solution) and the best (in terms of the solution's value) GA combinations. The fastest would be a GA with population equal to 50, crossover operator equal to cut and crossfill, mutation operator equal to simple swap and the selection operator would be the exponential ranking. This GA achieves a 0.35 seconds time on average and a distance of 7818.
The best would be a GA with population equal to 200, crossover operator equal to cut and crossfill, mutation operator equal to SBM and the selection operator would be the exponential ranking. This GA achieves a 1.86 seconds time on average and a distance of 7663.

VII. Conclusions

In this paper, we have shown the design and experimentation of a Genetic Algorithm focused on solving a symmetric salesman traveling problem (STSP).

We have implemented several mutation operators, crossover operators, population sizes, selection operators and tested them in order to see if there were any significant differences between them in both the solution obtained and the running time. In addition, we have presented the two options that we think are the best, one in terms of time efficiency and the other one in terms of the better solution (which surprisingly is not the slowest solution overall).

We conclude that we have obtained very good results in some of the tested configurations, even reaching the optimal solution.Thus we found that the use of genetic algorithms in the TSP task is a good method of resolution even in terms of computational cost, at least for instances of similar size to Berlin-52.

As future proposals, it would be great to study the possible interaction between all the factors tested (something similar to a 4-way ANOVA but for non-parametric data). It is true that the fact of using non-parametric tests makes it difficult to analyze the magnitude of the significant effect and therefore the interactions between factors are difficult to study. Nevertheless, it would be interesting to use some technique such as the permutation test, or to try to normalize the experimental results with some transformation.

It would also be useful to test different configurations that take into account different values of the hyperparameters of both the general algorithm (number of children and crossover and mutation probability) and the operators used (linear ranking, exponential ranking and tournament probability). As we have read in some studies [10], these hyperparameters can affect the value of the obtained solutions, being interesting to find the optimal ones.

Also, creating and implementing custom operators for the 52 node TSP instance may increase the overall performance of the algorithm.

References

[1] Ujjin, S., & Bentley, P. J. (2003, April). Particle swarm optimization recommender system. In *Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No. 03EX706)* (pp. 124-131). IEEE.

[2] Rodriguez, M., Posse, C., & Zhang, E. (2012, September). Multiple objective optimization in recommender systems. In *Proceedings of the sixth ACM conference on Recommender systems* (pp. 11-18).

[3] Rad, H. S., & Lucas, C. (2007, September). A recommender system based on invasive weed optimization algorithm. In *2007 IEEE Congress on Evolutionary Computation* (pp. 4297-4304). IEEE.

[4] Pérez-Marcos, J., & Batista, V. L. (2017, June). Recommender system based on collaborative filtering for Spotify's users. In *International Conference on Practical Applications of Agents and Multi-Agent Systems* (pp. 214-220). Springer, Cham.

[5] Hassanat, A., Alkafaween, E., Al-Nawaiseh, N. A., Abbadi, M. A., Alkasassbeh, M., & Alhasanat, M. B. (2016). Enhancing genetic algorithms using multi mutations. *arXiv preprint arXiv:1602.08313*.

[6] Esra'a Alkafaween, Ahmad B. A. Hassanat. Improving TSP Solutions Using GA with a New Hybrid Mutation Based on Knowledge and Randomness.

[7] Zakir H. Ahmed. Genetic Algorithm for the Traveling Salesman Problem using Sequential Constructive Crossover Operator.

[8] Sánchez, V. Unidad 5: Metaheurísticos - Algoritmos Genéticos, Optimización.

[9] Terrádez, M., & Juan, A. A. (2003). Análisis de la varianza (ANOVA). línea]. Disponible en: http://www. uoc. edu/in3/emath/docs/ANOVA. pdf.

[10] Rexhepi, A., Maxhuni, A., & Dika, A. (2013). Analysis of the impact of parameters values on the Genetic Algorithm for TSP. *International Journal of Computer Science Issues (IJCSI)*, *10*(1), 158.