Problem 1. Approaching TSP using Evolutionary Computing

Victor Montiel Argaiz

January 30, 2012

Contents

1 Introduction **2**

2 Description of the GA **2**
2.1 Encoding . . . 2
2.2 Evaluation . . . 2
2.3 Mutation . . . 3
2.4 Recombination . . . 3
2.5 Selection . . . 3

3 Code Structure and implementation **6**
3.1 General structure . . . 6
3.2 Implementation details . . . 7
3.2.1 Mutation . . . 7
3.2.2 Recombination . . . 7
3.2.3 Selection . . . 8
3.2.4 Code flow . . . 8

4 Results **9**
4.1 Selection . . . 9
4.2 Mutation . . . 10
4.3 Recombination . . . 12
4.4 Numerical parameters . . . 16

5 Summary and improvements **20**

6 Appendix **23**

A Program Compilation **23**

B Program Execution **23**

1 Introduction

The Travelling Saleman Problem (TSP) is a well-studied NP-hard problem in combinatorial optimization and computer science. Given a set of points, and the distance amongst them, the problem consist on finding the shortest path that visits each city exactly once, finishing the tour at the departure point.

Since formulated first in 1930, it has been one of the most studied problem in optimization and theoretical computer sciences, so, even if the problem is NP-hard and computationally difficult, a large number of strategies and heuristics have been developped to give approximate solutions to large problems (hundred of thousands of cities, see [3]) in reasonable time, which have been proved to be close to the optimal solution.

In this exercise, we approach the TSP using genetic algorithms (GA) to tackle the problem, and study the solutions obtained by using different parameters and operators of the GA, such as mutators and recombination operators, mutation and recombination probabilities and survivor selection methods. The goal is to implement a GA which reads a list of cities and their geographical coordenates from a CSV file and return a minimum distance path satisfying the TSP conditions.

2 Description of the GA

2.1 Encoding

Although GA is a generic optimization tool that can be used to a wide range of problems, the genotype representation of the problem, as well as the mutation and crossover operators, have to be carefully chosen and adapted to the problem so that the optimization search performs efficiently. For the TSP, the natural genotype encoding is the permutation representation. This representation encodes directly the order in which the cities have to be visited. Given a problem of size N, with N the number of cities to be visited, the permutation representation for the genotype is a sequence of the numbers within the set $\{1, 2, \dots N\}$ where the elements can be arranged in any order, and each element appears once and only once. Using this enconding we assure that all genotypes are valid solutions to the problem, since they are chosen from the permutation space, thus, verifying the constraint of the problem that each city must be visited once and only once.

Once the permutation representation has been chosen, we must then select mutator and recombination operators adapted to this representation, which must preserve the permutation property that at each sequence (valid permutation) each element appears once and only once.

2.2 Evaluation

The natural evaluation function for the problem is the distance of the path. In our implementation, the distance is calculated as the sum of the geodesic distance between the cities according to the order given by the permutation in the genome of the best individual (solution). The use of the geodesic distance is a proxy for the true car-distance amongst the cities, and allows us to focus on the problem without the burden of calculating the true car-distances considering roads between cities. Since in the data file passed to the program the cities are located using their latitude and longitude coordenates, the geodesic distance between cities can be calculated using formula (1), where the coordenates are expressed in radians.

$$
\begin{aligned}
dlon &= lon_1 - lon_2 \\
dval &= sin(lat_1)sin(lat_2) + cos(lat_1)cos(lat_2)cos(dlon) \\
distance &= acos(dval) \cdot earthRadius
\end{aligned} \tag{1}
$$

2.3 Mutation

For the mutators operator, we have implemented *swap*, *insert*, *scramble* and *inversion* mutation to compare results, all described in [2] . These operators work by reordering the alleles in the genotype, differenciating from each other in the way they rearrange the alleles and the number of alleles they relocate. As we will see in the results, for adjacency-based problem, as the TSP is, the *scramble* mutation causes big changes when decoding the genotype to a problem solution, resulting in a lot of links broken after the mutation, thus, provoking a big evolutionary step of the local search being able to completely spoil the previous solution. That is the reason why we prefer the other three, specially *inversion* method, for the TSP problem.

2.4 Recombination

As for the recombination operators, we have implemented *partially mapped crossover* or *PMX*, *edge-3 crossover*, *order crossover* and *cycle crossover*, all of them described as well in [2]. As for mutation operators, the implementation of the four methods is for didactic purposes, since, as we will see in the results, some of the implemented recombination strategies are not suitable for adjacency-based problems as the TSP.

Concretely, the *PMX* is not a suitable operator for our problem since, by its construction, tends to create a lot of edges in the offspring that are not present in the parents, thus, desfiguring the previously achieved partial solution of the parents. *Order crossover* is an improvement of *PMX*, since, once the crossover region has been copied into the offspring, the rest of the genotype is obtained from the second parent conserving more edges than in the *PMX* method.

Cycle recombination tends to conserve as far as possible the absolute position of each allele in the offspring. Observe that this might not be an specially desired property for the TSP, since, as the problem consist on finding a cycle path (starting and finishing in the same point) the absolute position of each city in the path is not so relevant for this problem as the relative position of the cities with respect to the others (the actual edges indeed). Think that given a solution candidate $[a, b, c, d, e, f, g, h]$, any circular shift of this solution, for example $[b, c, d, e, f, g, h, a]$ will provide the same fitness value (distance of the tour), concluding that absolute positioning is not a so important property.

Finally, *edge-3 crossover* is one of best suited operators, amongst the four implemented, for this problem. The idea behind this operator is to create offsprings using, as far as possible, edges present in both parents. It is this tendency to preserve edges (pairs of cities visited consecutively) which makes it a better performer.

2.5 Selection

Selection methods is applied in two points during the execution of the GA. First, when selecting individuals for reproduction, we try to choose the best fitted individuals to create offsprings, this is *mating selection*. Once the population has reproduced, we have to select between parents and offsprings the best fitted individuals to go for the next generation. This last choice is *survivor selection*.

In our implementation, the population keeps the number of individuals at each generation constant, and the age of the individuals (number of generations that have survived) is not considered, looking only at fitness values as a criteria for selection. On the other hand, selection is done in a probabilistic way, assigning each individual a probability of being selected according to its fitness value.

We have implemented two methods to assign selection probabilities. The first method is *fitness proportional selection*, where we assign to each individual a probability which is proportional to its fitness with respect to the sum of the fitness of all individuals of the population. On the other hand, we have implemented *ranking selection*, where the probabilities are assigned according to the rank of the individual, and not to its relative fitness, performing a more selective method, specially when the fitness values of the population are pretty uniform.

The distribution of probabilities is made according to the *exponential ranking scheme* described in [2]. Our implementation has an additional parameter, *shape factor*, which allows to adjust how picky the selection algorithm will be. Given and individual i, and its rank in the population, $rank(i)$, the probability assigned to this individual is calculated as in formula (2), where c is the *shape factor* and a is a constant so that $\sum P_i = 1$.

$$P_i = \frac{1 - e^{-rank(i) \cdot c}}{a} \tag{2}$$

As we can see in figure (1), where we represent the accumulated probability distribution of being selected for a given rank, the *shape factor* allows us to define how selective we want to be. The larger c, the more selective we will be, selecting in most cases the first ranked individuals. For negative c, we achieve the opposite results, giving more probability of selection to the worst fitted individual. For values of c close to zero, observe that the accumulated probability function is almost a linear function, being able to reproduce then the *linear-ranking scheme* described in [2].

Once we have defined the probability distribution, the actual selection is implemented using the *roulette-wheel* algorithm.

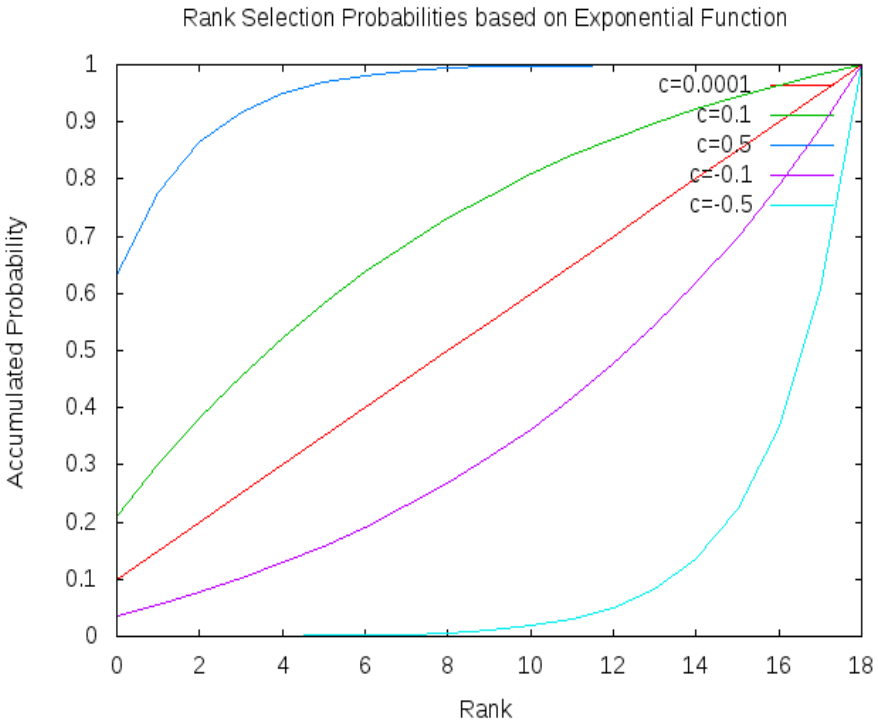


Figure 1: Accumulated probability distribution for a given rank.

3 Code Structure and implementation

3.1 General structure

For the implementation of the GA we have opted for an object oriented approach written in C++. Additionally, we have used Boost libraries [1] for random number generation, parsers, command line parameters as well as shared pointers. For result plotting we have used the gnuplot-iostream library, which allows a direct connection between our program and gnuplot.

Code organization is based in the following files:

- `City.hpp`: It defines the `City` object, representing a city, each vertex of the TSP. Each object contains the name of the city as well as the geographical coordenates.
- `CityDataBase.hpp`: Database storing all the available cities, read from a CSV external data file.
- `CSVTableParser.hpp`: Parser to read a CSV external data file with the information of the cities. It returns a `CityDataBase` object once the file has been parsed.
- `Destinations.hpp`: Object containing the cities which are part of a given instance problem. It calculates also the distance matrix amongst the cities.
- `Genome.hpp`: It models a genome or individual. The encoding of the genotype is made using the permutation representation, implemented as a vector of integer numbers, each entry i on the vector storing the i^{th} city visited.
- `Population.hpp`: Object containing all individuals of the population.
- `CrossOver.hpp`: Definition of crossover operator. We define an abstract class `Crossover`. Each crossover operator implemented has to derive from the abstract class and implement the `DoCrossOver` method. The method takes two genotypes from the parents as input and returns the two resulting genotypes of the offsprings.
- `Mutators.hpp`: Definition of mutator operator. We define an abstract class `Mutator`. Each mutator operator implemented has to derive from the abstract class and implement the `Mutate` method. The method takes as input parameter the genotype to be mutated.
- `Selector.hpp`: Definition of the selector operator. We define an abstract class `Selector` and each selector method implemented has to derive from that abstract class, implementing the `Select` method. The Select method takes a `vector` of `Genome` objects and a parameter *lambda*, representing the proportion of individuals from the population that must remains after the selection process.
- `Evaluators.hpp`: Definition of evaluator operation. We define an abstract class `Evaluator`, each evaluator implementation having to define the `Evaluate` method. Evaluation is done based on the geodesic distance between cities, just one implementation of the abstract class has been provided in consequence.
- `GA.hpp`: Orchestrator of the evolutionary process. It contains references to `Mutator`, `CrossOver`, `Evaluator`, `Selectors` and `Population` objects, as well as the parameters (mutation probability, recombination probability, maximun number of generations) required to run the simulation. It defines the method to run the GA in an intuitive way, via `Initialize`, `NextGeneration` and `TerminationCondition` methods, which, initializes the object, runs a one-generation step in the simulation and checks for the termination condition of the algorithm respectively.
- `ResultsAccumulator.hpp`: Object which stores partial results of the population for each generation. For each step in the simulation, it keeps tracks of the best and worst individuals, the average fitness value of the population, the user-cpu time consumed, the probability distribution (histogram) of the population, as well as a copy of the best fitted individual.

- `RandomGenerator.hpp`: Object defining three different types of random number generator, one for uniform integer numbers generators withing a range, one for uniform real number generator within a range and a Bernouilli random number generators. All have implemented based on the `random` classes in the Boost library.
- `optionparser.hpp`: File containing functions to parse the command-line arguments, using the `program options` classes within the Boost library.
- `main.cpp`: main file of the program (see the implementation details section).
- `test.hpp`: test batteries for the objects used through the simulation.
- `types.hpp`: Types used within the code.
- `utils.hpp`: Utilities used for debugging purposes.

3.2 Implementation details

Being Genetic Algorithms a highly repetitive process, the implementation of the operators, which will be called thousands and millions of times during the execution, might impact in a respectable way in the time-performance of the algorithm. On the other hand, the GA algorithm are easily parallelizable algorithms, with little or none inter-thread communication needed, thus, a parallel GPU implementation could speed up the performance for the most complex operators.

For this problem we have focused on studying the properties of the different operators, being the priority implementing several operators instead of just one highly optimized operator. Once we obtain the results, the next step could be the optimization of the implementation of the best suited operators and see how a good design could speed-up the algorithm with little effort. Finally, the implementation written for this problem has not taken paralelization into consideration .

3.2.1 Mutation

The mutation operators have been implemented using the ready-made functions in the C++ STD library. So, for the *insert operator* we have used the `insert` method for vectors, for the *scramble mutation* we have used the `random shuffle` method and for the *inversion mutation* we have used the `reverse` method. All these three operator have running time of $O(n)$. Finally, the *swap mutation* is implemented in the trivial way, just by swapping two randomly chosen positions, leading to a $O(1)$ runtime algorithm.

3.2.2 Recombination

The first recombination algorithm is *Partially Mapped Crossover*. Basically it works by copying the mid region (region between crossover points) of the parent genotype in the offspring, and filling up the other regions with the alleles in the second parents, avoiding repeating genes already within the crossover mid region. The implementation of the algorithm leads to a worst-case runtime of $O(n^2)$, since we might end up doing linear search on the genotypes for each position outside the crossover region.

Order crossover has a more simple implementation, leading to a $O(n)$ algorithm which copies the crossover region of one parent into the offspring and then copies the remaining part of the genotype from the other parent in the same order, skipping the already assigned genes in the crossover region.

The implementation of *cycle crossover* is based on linear search to find the cycles between the two parents. Starting from the first position, we scan for cycles between the two parents, copying the elements of the cycle in the two offsprigns. Once a cycle has finished, we continue with the next non-asigned allele. This lead to an implementation with worst-case runtime of $O(n^2)$. A more intelligent data representation, where we use data structures avoiding the linear search (hash-tables) could lead to a better performing

algorithm.

Finally, for the implementation of *edge-3 crossover* we have to build the adjacency list, which can be done in $O(n)$ just by walking through the genome and saving in a set the previous and the next element in the genome for a given allele. Once an entry in the table has been chosen, according to the algorithm described in [2], we have to remove all references in the table to that element. Observe that we dont need to go through all the adjacency list in order to look for references for a given entry (which would lead to a final $O(n^2)$ algorithm). Instead, think that for a given entry, at most there are 4 other entries which contains this city as an edge. Thus, to do an effective removal of used entries, we have just to visit the original adjacency table (a table, copy of the initial adjacency list, where we have not removed any of the elements), visit the entry for the element that we want to remove, and recover the up to four potential entries which might contain references to that element. Using this trick we have speeded up the inital $O(n^2)$ implementation to $O(n)$ with a considerable saving of time when the size of the problem is large. This last optimized version in the code is reffered as `OptimizedEdgeCrossover` in the code, and it is the recombination operator that we have used to test the performance of *edge-3 crossover*.

3.2.3 Selection

Selection operator has been implemented in the `Selection` class, providing two implementatios, one for *Fitness Proportional Selection* and another for *Ranking Selection*. The probability assigment for the last one has been previously explained in equation (2). The roulette-wheel algorithm has been used to pick the individuals according to the underlying probability distribution. Observe that the algorithm described in the book has a worst-case running time of $O(n^2)$. With some effort, we could have improved the algoritm doing binary search on the accumulated probability distribution once the random variable has been drawn, leading to a $O(nlog(n))$ version.

3.2.4 Code flow

The execution flow of the program can be summarized according to the code in the main method as follows:

- Step 1: Parsing of the command line arguments passed to the program to define the parameters necessary for the Genetic Algorithm.
- Step 2: Creation of the operators to be used by the Genetic Algorithm according to the parameters specified on the program invocation. This includes the creation of the mutation, recombination, evaluation and selection opperators. Creation of the population object, which will gather all individuals alive at a given time (generation).
- Step 3: Creation of the GA object instance, passing as argument the operators to be used in the evolutionary process. Creation of the ResultAccumulator object, which, in coordination with the GA object, will keep track of the partial results and the evolution of the algorithm.
- Step 4: Initialization of the evolutionary process, invoking the `Initialize` method on the GA object.
- Step 5: Repetition of the evolutionary process until the termination condition is met. The instructions repeated in the loop, are, by this order: Selection of the parents that will reproduce (`SelectParents` method), creation of the offsprings using recombination (`Recombinate` method), mutation of the entire population, parents and offsprings (undertook by `Mutate` method), evaluation of the individuals alive in the population (`Evaluate`) and selection of the best fitted individuals (performed by the `SelectSurvivors` method). At the end of each generation, we update the `ResultAccumulator` objects with the statistics of the current generation by calling its `UpdateResults` method. Finally, we signal the GA object to move to the next generation using the

`NextGeneration` method. During the looping, at fixed periods of time, we print out the progress of the execution as well as we save a snap-shot of the current partial solution generated by the algorithm.

- Step 6: Check for the termination condition. In our implementation, the termination condition is met once the algorithm has reached the number of generations specified in the command line arguments. Other termination conditions might be equally valid, such as an specified number of generations without material improvement in the quality the solution. Observe that in the case of the TSP, we cannot set the termination condition to be the achievement of the optimal solution since, being TSP an NP-Hard problem, the calculation of the optimal solution is not, in the general case, possible in a reasonable amount of time.

- Step 7: Plotting of results. The partial results stored in the `ResultAccumulator` object are plotted in images files so that it allows further interpretation and evaluation of the results. In our implementation, we generate the following plots: Evolution of the best solution for each generation, evolution of a simplified statistical distribution for the population, plotting best, worst and average fitness for the population, evolution of the best solution with respect to the User-CPU time consumed in the computation. Finally intermediary graph showing the evolution of the solution are generated during the evolutionary process. Being the TSP a problem with an intuitive graphical representation, the plots of the partial solution allow a better understanding on how the local evolutionary search is performed at each edge. Equally, we write a file with the numerical partial results for each experiments, so that we can graphically compare two differents parametrizations (mutators, recombinator, mutation probability, recombination probability...) of the GA.

4 Results

Once the GA has been implemented, we have to test the different funtionalities over differents data sets in order to figure out the best configuration of the algorithm to get the solution with minimum distance. The guideline for this section will be, first, identify the best suited selection, mutation and recombination operators for the problem, and then, fine tune the remaining parameters, such as probabilities of recombination and mutation, shape factor, population size to get the best quality solution.

4.1 Selection

As discussed in [2], the preferred selection method is usually the *ranking selection*, since the *fitness proportinal* is not selective enough when the fitness evaluations of the individuals of the population are very similar. To compare both methods, we have run two different simulations on a 50 cities problem, each with a different selection operator. In figure (2) we have plotted the best-fitted individual at each generation for the two selection methods. As we can see, the *ranking* method is by far much more performant that *fitness proportional*. Indeed, for this case, *fitness proportional*, does not appear to be of great utility, since the selection preasure it forces is so small that the evolutionary process does not seem to improve the quality of the results. As discussed previously, when the population is made of individuals with uniform values of fitness, this method does not work very well. Some tricks can be done to the *fitness proportinal* method in order to improve the results, however, given that the *ranking selection* performs well with no need for further revision or improvement in the algorithm, we consider we should apply this last method for the TSP.

The other parameter we can play with is the *shape factor* which adjusts how selective we are on the selection process. We ran several simulations with different values to see how the *shape factor* parameter influences in the solution. Results are plotted in figure (3). As we discused previously, large values of *shape factor* means a more selective GA, while values close to zero applies an almost linear accumulated probability distribution assigment based on the rank, so, the selection preassure applies equally on best

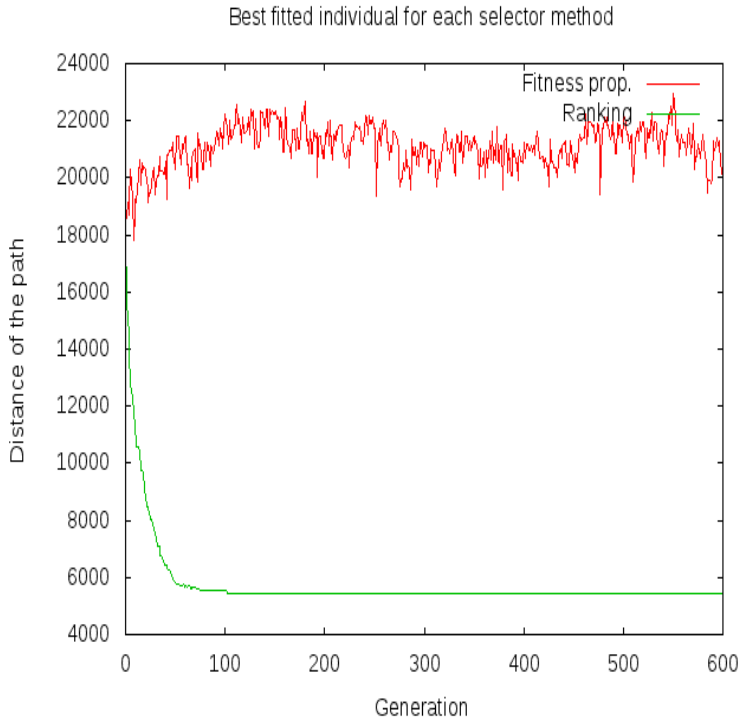


Figure 2: Comparation of Fitness Proportional and Ranking selection methods for a small TSP.

and ordinary individuals, an the evolutionary process fails to improve the solution at each generation. As we can observe, for the first two values of the *shape factor* ($s = 0.0001, s = 0.001$) evolutionary process does not really evolve, and the best individual at each generation does not converge to a better solution. The last two values ($s = 0.1, s = 0.01$) of *shape factor* make the algorithm more picky, doing a most aggresive selection of the best fitted, leading to a real evolutionary process where the best individual improves rapidly at each generation. Observe however that for these two last values, the algorithm reach a solution that is not able to improve after the 100^{th} generation. Notice also that for the case of $s = 0.01$ we get an slightly better result than for $s = 0.1$, $5537.32km$ vs $5643.65km$, which is an example of premature convergence, and indicates that being very selective, specially at the initial stage of the algorithm, can lead to sub-optimal solutions rapidly.

4.2 Mutation

At first sight, *scramble* mutation does not seem to be a good choice for this problem, since, most of the time, the application of this operator completely change the initial solution, being the two individuals, before and after applying the mutation, very distant one from another. In general, we have seen better results when the evolution is made of small changes to the individuals, rather than big random changes that can lead to spoil previous evolution steps by completely desfigurating the individuals. The other three remaining operators share the property that the change they make on the genotype is minimal, changing just a few edges with respecto to the initial solution.

In the case of *swap* mutation, only two alleles are swapped, which means that the number of edges changed in the path represented by the individual is four. For the *insertion* operator, again, only four edges are changed after the application, remaining the rest of the itinerary equal to the initial version. Finally, for *inversion* operator, just two edges are changed within the itinerary.

In order to compare the four operators, we have run simulations on a 1000 cities TSP, plotting the fitness of the best solution after 400 generations and a population size of 200. The results are plotted

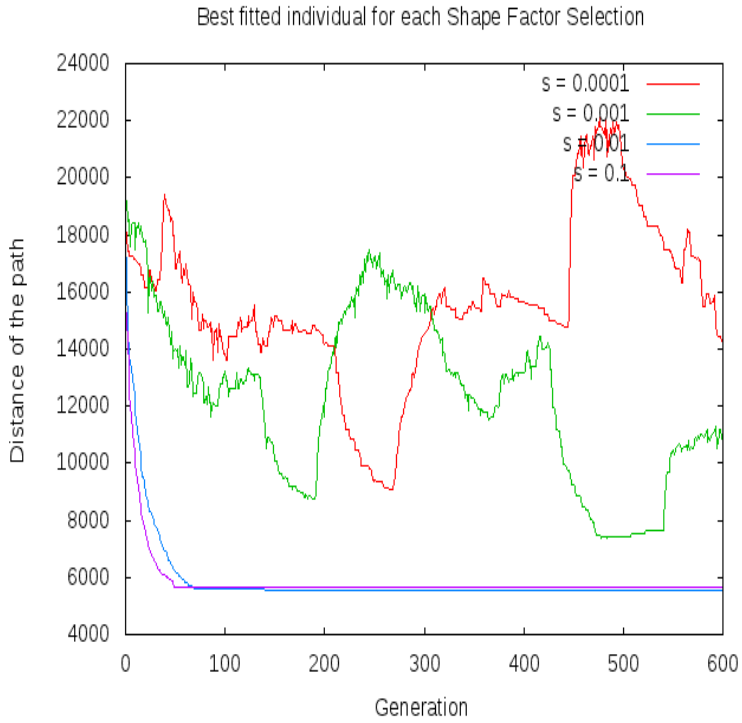


Figure 3: Comparation of Shape Factor on the Selection Process.

in figure (4). As expected, *scramble* operator is the worst performer, and the other three behave in a similar way. In order to break ties, we can also plot the minimum distance obtained for each operator against the user-CPU time consumed. Results are shown in figure (5), and we cannot see any material difference in the efficiency of any of the three, which means, either that the three operators are equally efficient, or that the real bottle-neck of the algorithm is somewhere else, probably at the recombination operator.

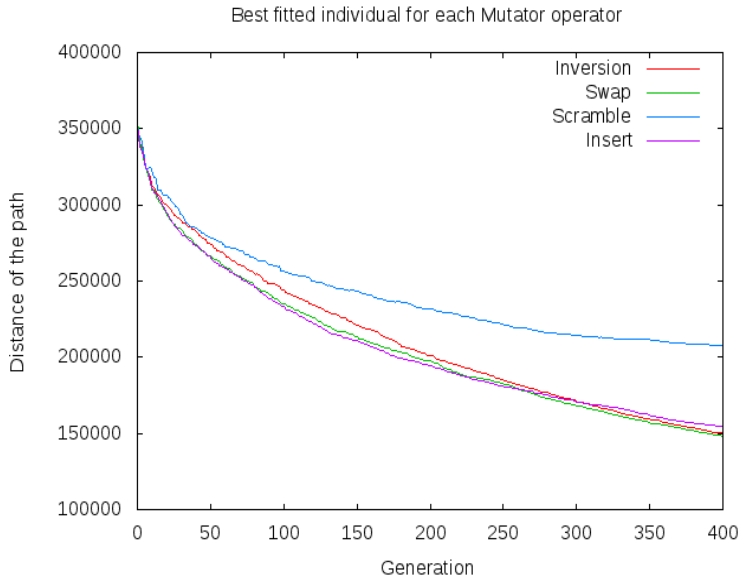


Figure 4: Comparation of Mutator Operators on a 1000 cities TSP

Studying the operators over a smaller set of cities might help to better understand the behaviour of

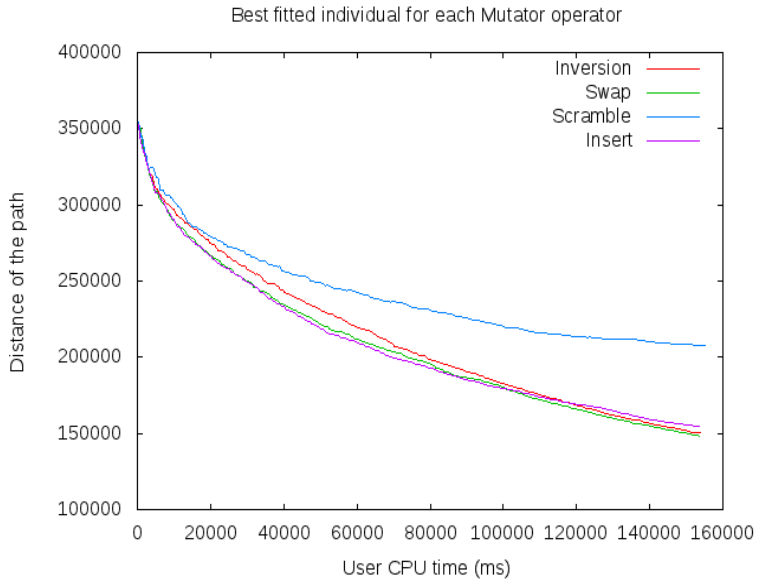


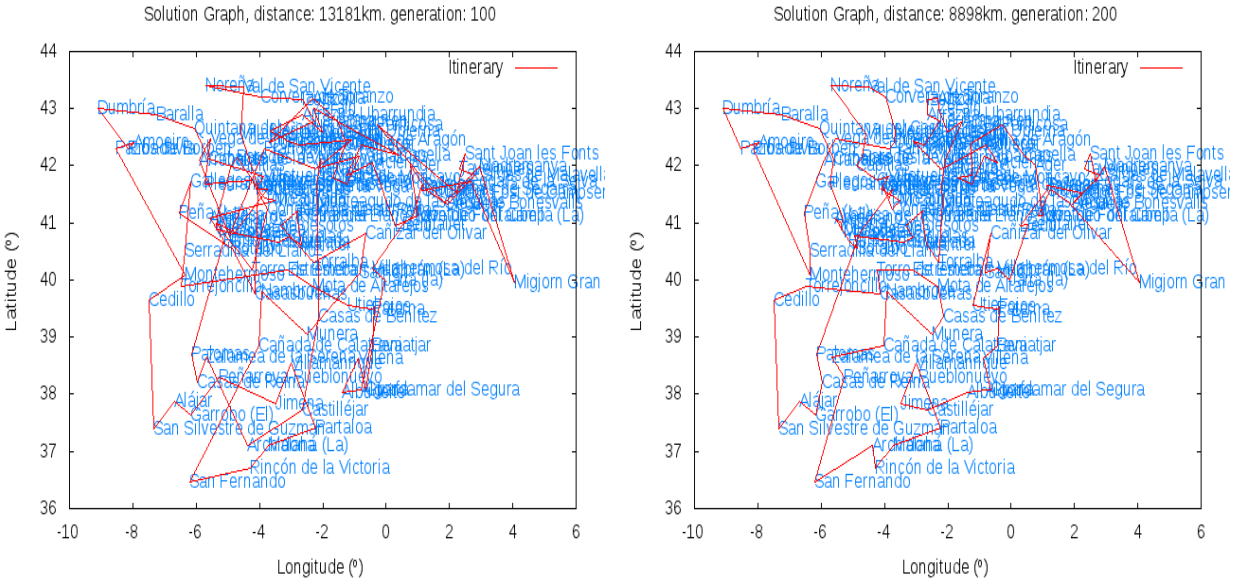
Figure 5: Comparation of Mutator Operators on a 1000 cities TSP, x axe is user-cpu time

each of them. In figure (6) we can see the evolution of the partial solution for a problem of 125 cities. Observe on the figures that the evolution of the algorithm go through unwinding the tours that cross over themselves. Put it on another way, imagine a tour of $k - edges$ between two cities, if the tour pass twice or more times over a point (this point is anypoint in the plane between cities), that tour is not probably optimal. We will be able to find another tour between those two cities (typically reversing one ore more of them), which does not cross over itself, and has shorter length. This is exactly the idea behind the *Lin-Kernighan*, or λ-opt algorithm described in [3]. Of course, the idea developped on [3] is much more complex than this, but the working method of the algorithm is rather similar to the *inversion mutation* operator. Being the *Lin-Kernighan* heuristic one of the best approximate solution to the TSP, that would explain why the underlying idea of the *inversion mutation* works so well on this particular problem.

4.3 Recombination

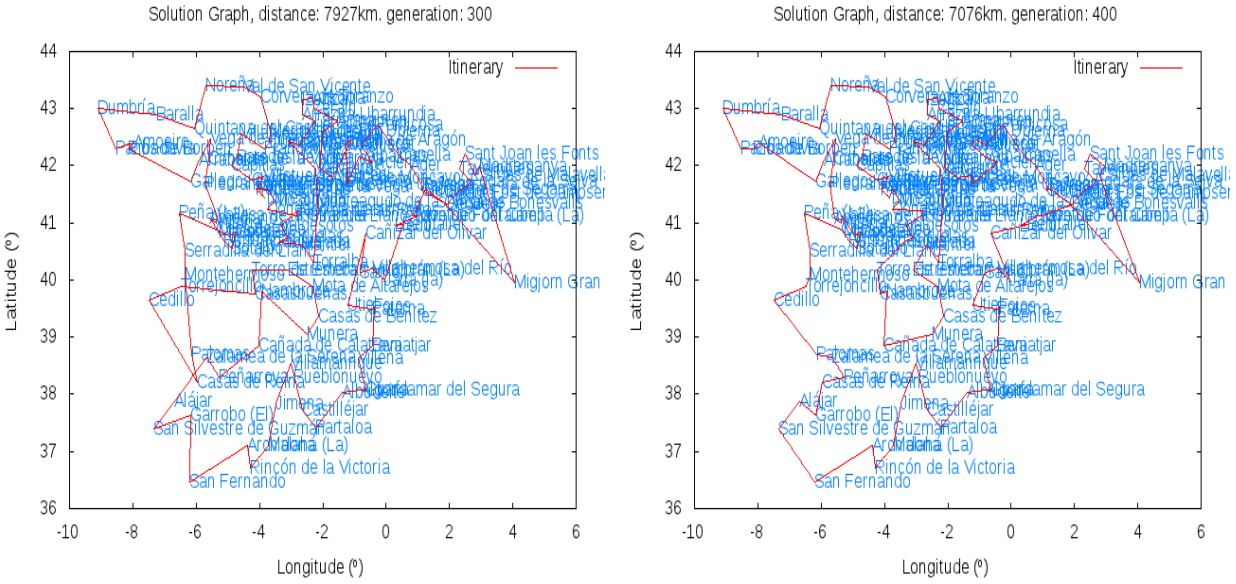
Applying the same ideas as in the mutation section, the recombination operators that breaks a lot of links in the parent genome tends to perform worse than the operator that introduce small changes at each step, trying, as far as possible, to respect the edge presents in both parents. For this reason, we would expect *3-edge* operator to perform much better than the others, specially than *PMX* and *order*. Finally, as discussed previously, the property of *cycle* operator of conserving as far as possible the absolute position of the gens after the recombination operator is not specially interesting for the TSP, so we should not expect an special good performance for this operator. To check empirically the different types of recombination, we have run simulations on a 1000 cities problem using the four operators, results are plotted in figure (7). As for the mutation operator, it is interesting to study, not only the absolute performance, but also the relative performance against the user-CPU time consumed. Results for this comparation are plotted in figure (8).

As expected, the *edge-3* crossover is by far much more effective than the other ones. When compared against the second one, the *order* crossover, we obtain a solution 20% shorter for the same number of generations. *Order* and *PMX* operator perform rather similar, being *Order* slightly better. Observe that, as previously discussed, the *order* operator was conceived in order to improve the *PMX* solution, with the property of transmitting as much as possible the information about the relative order from the



(a) Partial solution after 100 generations

(b) Partial solution after 200 generations



(c) Partial solution after 300 generations

(d) Partial solution after 400 generations

Figure 6: Evolution of the partial solution each 100 generations.

second parent, breaking fewer links present in the initial solution of the parents. Thus, it is not surprising that *order* crossover is slightly better. Finally, the *cycle* operator is the worst performer, having being the reasons previously explained.

In terms of efficiency, fitness function against CPU time consumed, our *edge-3* operator implementation is not very performant, needing a time five times greater than the *order* operator, which is the most efficient. It is thus interesing to compare the solution obtained by *order* operator employing same CPU time than the *edge-3*. In order to run this comparation, we experiment with a GA using five times more generations in the case of the *order* operator. Results can be visualized on figure (9) and surprisingly, even if the *logic* behind the *edge-3* operator is more suited for the TSP, our implementation makes that the *order* operator performs better when measuring against CPU time, being able to reach or even improve the solution provided by *edge-3*. Observe than in the figure, we can see that for the same

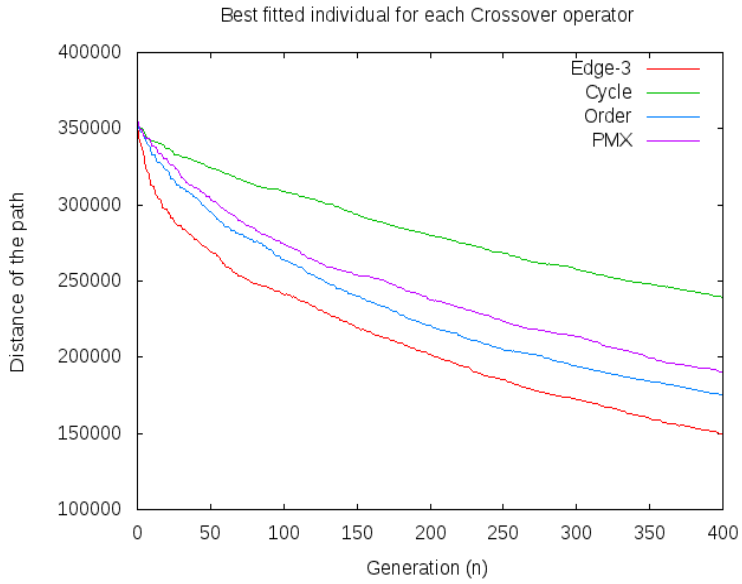


Figure 7: Comparation of Crossover Operators on a 1000 cities TSP

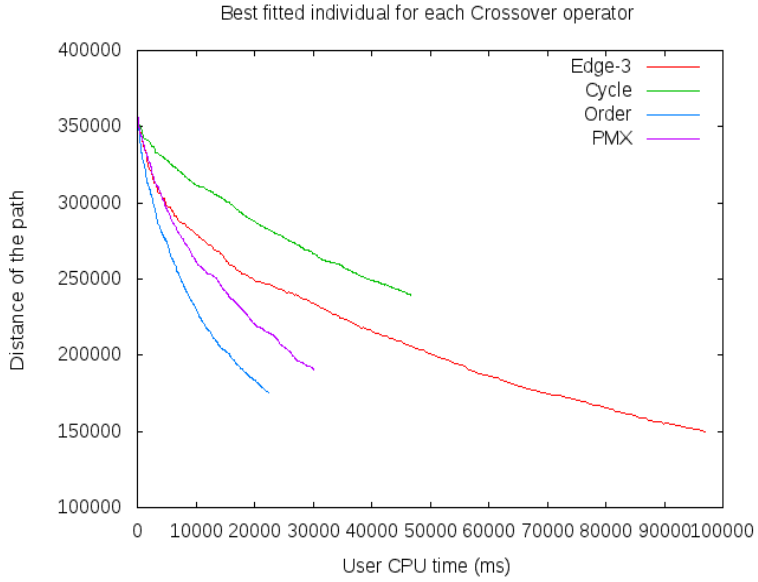


Figure 8: Comparation of Crossover Operators on a 1000 cities TSP, x axe is user-cpu time

CPU time, we get a solution approximately 50% shorter. In a future revision, we could probably improve our *edge-3* implementation to obtain even better results, reducing the current CPU time consumed per generation.

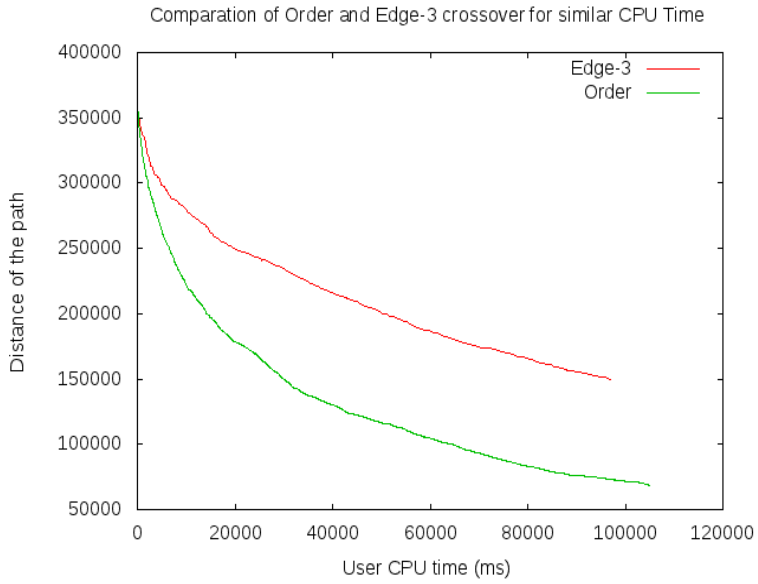


Figure 9: Comparation of Order and Edge-3 Operators on a 1000 cities TSP

4.4 Numerical parameters

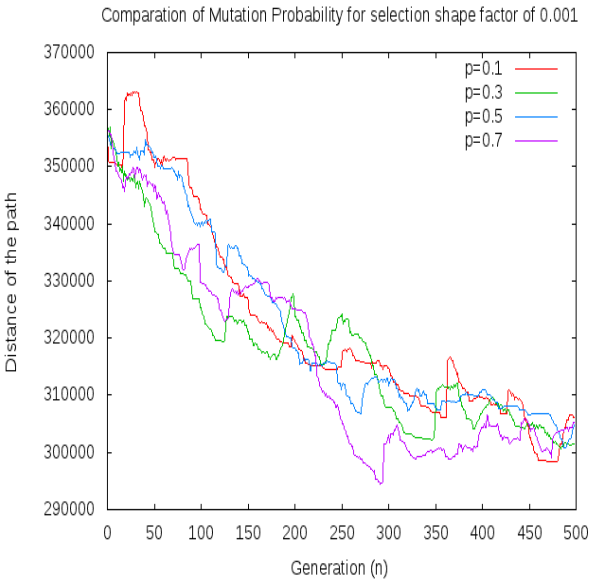
Once we have compared the relative performance between different mutator, recombinators and selection operators, we should study how the parameters which controls those operators, i.e., *mutation probability*, *recombination probability*, and *shape factor*, impact on the final performance of the algorithm. For the study, we have used a 1000 cities TSP.

Concerning the *mutation probability*, we have explored the impact of this parameters against different values of *shape factor*. Remember that the *shape factor* makes the GA be more or less selective. Then it is reasonable to think that for different selection preassures, the *mutation probability* parameter might impact in a different way. In figure (10) we can observe the results. Observe that for low selection preassure, the results obtained are, in any case, rather poor. As we increase the selection preassure, we observe that the greater the *mutation probabilty* the better the results, although the difference is not huge. Thus, the combination of big selection preassure along with high *mutation probability* gives the population more variety to find new solutions, and at the same time, big probabilities of choosing the best individuals, leading, overall, to better final results.

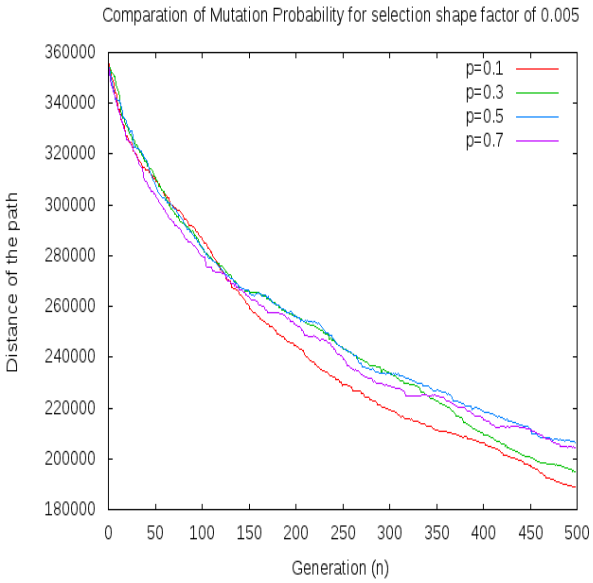
Concerning the *recombination probabily*, the results are not so clear. For low selection preassures, i.e., low values for *shape factor*, the behaviour of the GA with different *recombination probability* is also rather erratic, as it was for *mutation probability*. However, when selection preassure increases, the behaviour tends to be rather uniform for a wide range of *recombination probabilities*, showing no material differences. Results for these experiments are shown in figure (11).

To conclude, being more aggresive on the selection process seems to be a better configuration of the GA simulation. On the other hand, when we are very selective, the impact on the probablities of mutation and recombination tends to be minimized, showing a slightly preference for higher *mutation probabilities*. This could be because, being very seletive, only the very best individuals will remain, so, even if the probability of changes in the population are high or slow, just a few variety of individuals, the best, will persist after some generations. If this is the case, being selective means having a population more uniform, with little differences between the best and the worst individual. Observe that, althogh this could lead to poor results of the GA in the general case, for the TSP might not be the case. As we observed in figure (6), the evolution of the GA leads to more clearer solutions on the pictures, where the edges of the graph do not crossover, drawing a cleaner itinerary. So, the local search of the problem focuses on unwinding step by step these crosses or overlappings of the paths, and once the algorithm have been able to correctly cluster the itinerary in different group of cities situated in the same region on the graph, the evolution steps focus on unwinding this overlappings, which represents, each of them, a small difference in the final evaluation of the fitness function. For this reason, uniform populations at advanced number of generations are not specially negative for the results in the case of TSP.

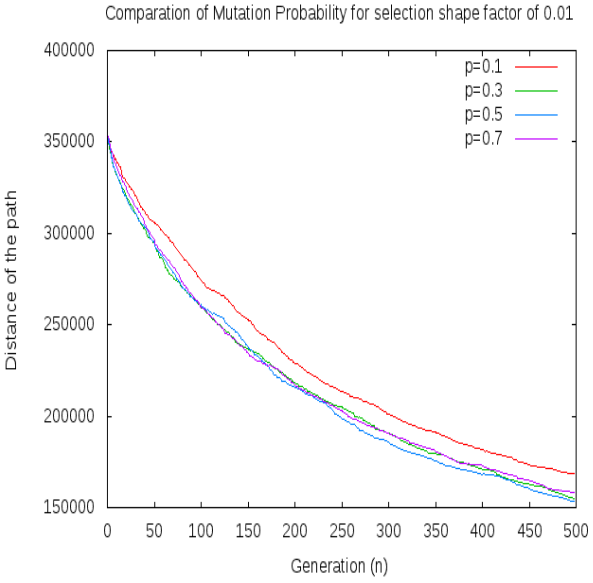
Finally, it is intersting to study the performance of the GA with different configuration of *population sizes* and *maximum number of generations*. For this comparation, we have performed simulations over a 1000 cities problem, maintaining the total number of individual constant, 400.000, being the total number of individual the *population size* times the *maximum number of generations*. Results are observed in figure (12). As we can see, the GA have preference for small population sizes which go through a big number of generations, being the performance, measured as the best solution found against the user-CPU time consumed, the best.



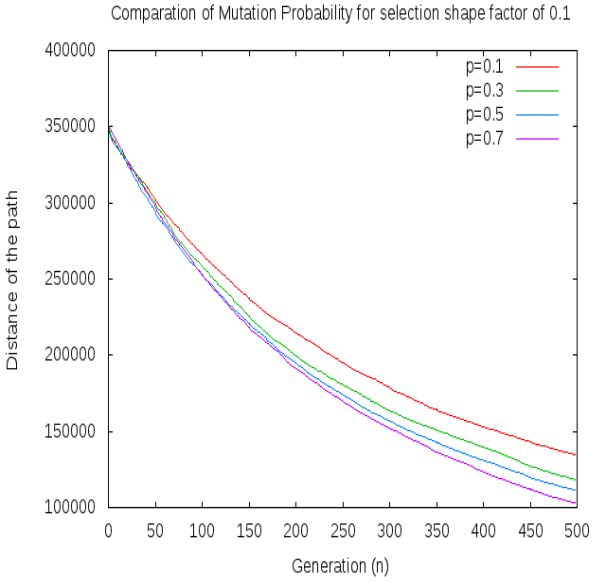
(a) Analysis of mutation prob., shape factor 0.001



(b) Analysis of mutation prob., shape factor 0.005

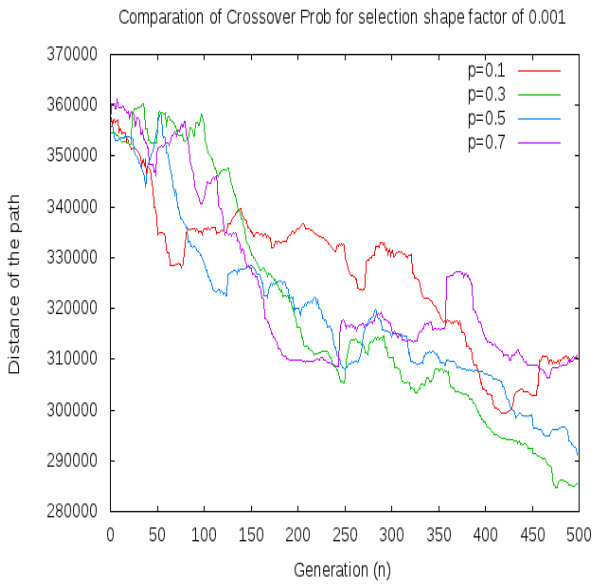


(c) Analysis of mutation prob., shape factor 0.01

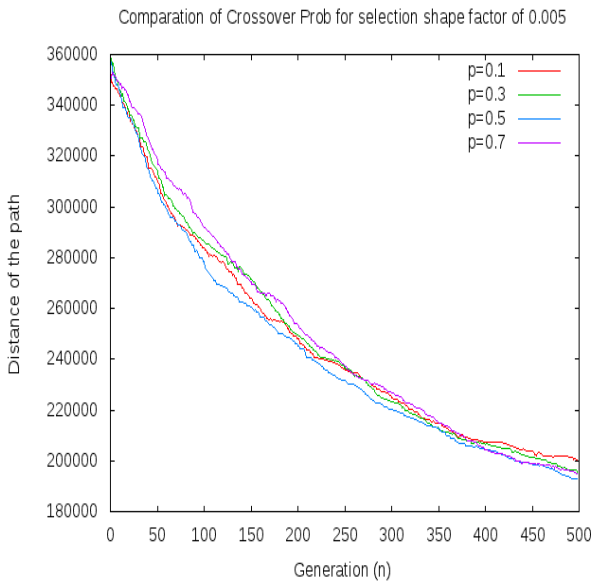


(d) Analysis of mutation prob., shape factor 0.1

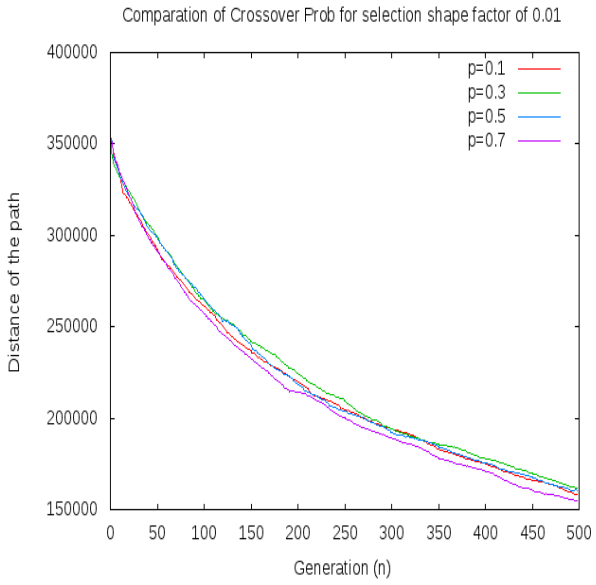
Figure 10: Behaviour with different mutation probabilities for different values of the shape factor in the selection function.



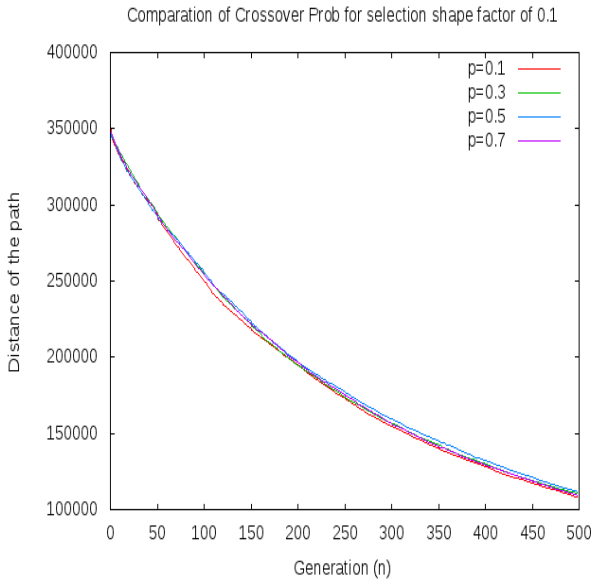
(a) Analysis of xover prob., shape factor 0.001



(b) Analysis of xover prob., shape factor 0.005



(c) Analysis of xover prob., shape factor 0.01



(d) Analysis of xover prob., shape factor 0.1

Figure 11: Behaviour with different crossover probability for different values of the shape factor in the selection function.

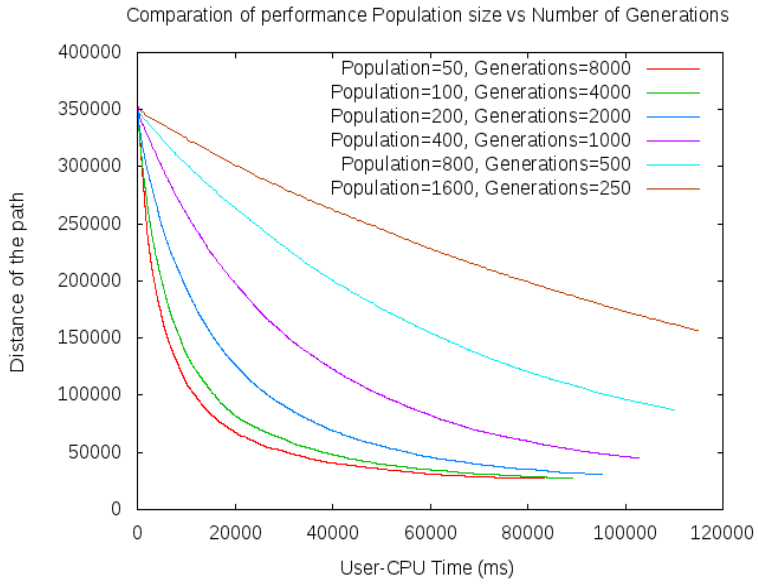


Figure 12: Comparation of performance for different configuration of population size and max. number of generations. In all instance problems, the number of individuals evaluated are 400.000 for a TSP size of 1000 cities.

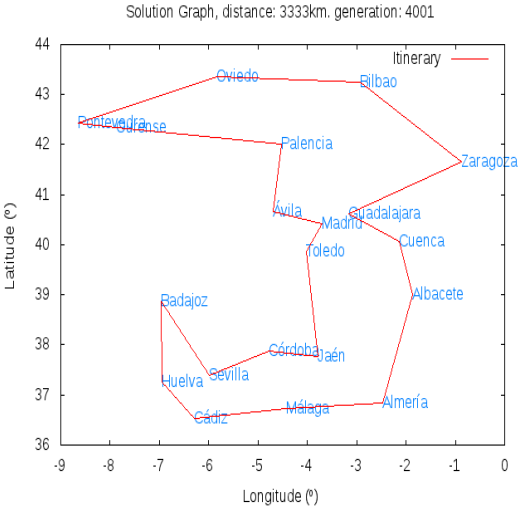
5 Summary and improvements

Amongst the different strategies we have developped and tested for the TSP using GA, we choose the *inversion*, *insertion* and *swap* mutator operator as the best performers for this problem, existing hardly no material difference between them. For the recombination operators, *edge-3* and *order* methods seem to be the best performers. Because of our particular implementatoin, *edge-3* is able to achieve better solutions with fewer number of generations, although regarding to the CPU time consumed, *order* operator is a best performer method. We can then apply either *edge-3* with smaller number of generations or *order* operator with a greater number of generations, getting similar solutions. Because of *order* has slightly better performance, we have preffered to use it in the final results. A more optimized implementation of *edge-3* would problably lead to better numbers, specially if we want to scale up the problem size for tens of thousands cities. At the same time, it has been left for futures versions the paralellization of the code, using GPU, which could improve the overall performance of our implementation.

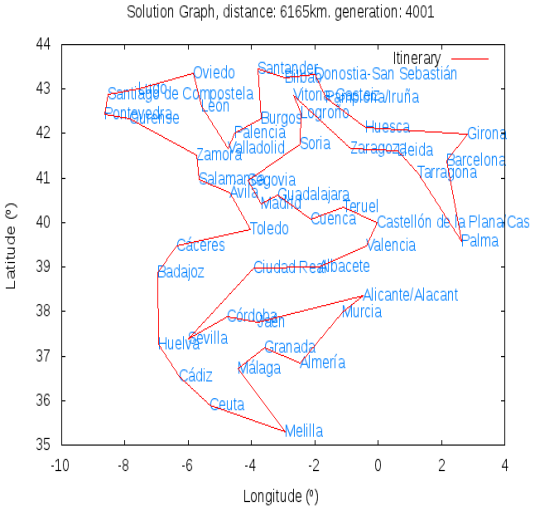
For selection, *exponential ranking scheme* with *shape factor* of 0.1 seems to produce the best results for the problem, outperforming completely the other method implemented, *fitness proportional scheme*. Finally, for this shape factor, being rather selective, we have found that the results are quite insensible to the probability of mutation and recombination.

In figure (13) we can observe the solutions generated for different sizes of the TSP problem. By visual inspection, we can conclude that the results are quite good, being optimal or close to optimal in the biggest sizes. Observe that although it is not inmediate to check whether we achieve optimal solutions for big size problem (think that for the 1000 cities problem, the number of possible permutations is $1000! \approx 4.02^{2567}$), visual inspection is able to give us a good insight about the quality of the results. It would have been interesting to compare these results against the generated by heuristic methods such as the one described on [3] and compare the differences in the final path length. In figure (14) we can observe the evolution of the population for each of the problems. Best individual, Average individual and Worst individual for each generation have been plotted. For small problems, the algorithm rapidly converge to the solution, being this optimal. Thus, we would not need the algorith to go through so many generations. As the problem size gets biggers, the algorithm needs more generations to improve the solution. Also, as we can state, population dispersion, i.e., the fitness difference between the worst and best individual on the population, is rapidly reduced to a minimun, given that the selection preassure chosen is high. The right tail of the curve is almost flat, which means that the algorithm have reached a local (possibly global in some cases) minimum, reaching the final solution. Once we reach the flat part of the graph, the improvements to the best solution are insignificant, since they are based on undoing the little overlappings that still remain on the final itinerary.

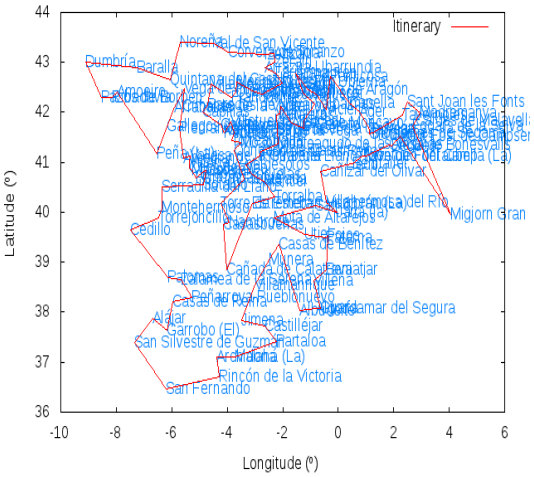
Simulation for the $n = 1000$ problem has been executed in 390570ms or 7min in a small laptop, which could satisfy the requirements of most applications of the TSP, where we do not need probably an online instant solution for the scheduling and planning of the trip. At the same time, as we discussed in the results, no special effort have been done in optimizing the best performant operators, or in parallelizing the application using GPUs, which could lead to important improvements with little changes in the code. Also, as previously discussed, metrics such as *success rate, SR* or *average number of evaluations to a solution, AES* metrics do not apply to the TSP, since, for big size problems, it is difficult to calculate the optimal solution. However, as discussed in [3], the human eye have been proved to be a good judge to declare whether a solution is quite good or bad based on the resulting graph, even if we cannot conclude whether the optimal have been reached.



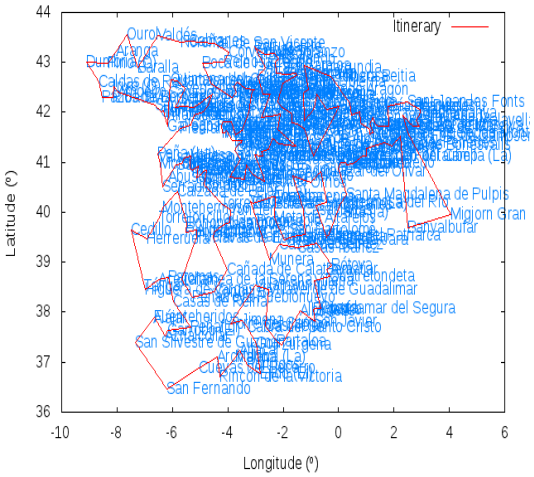
(a) Solution for $n = 20$



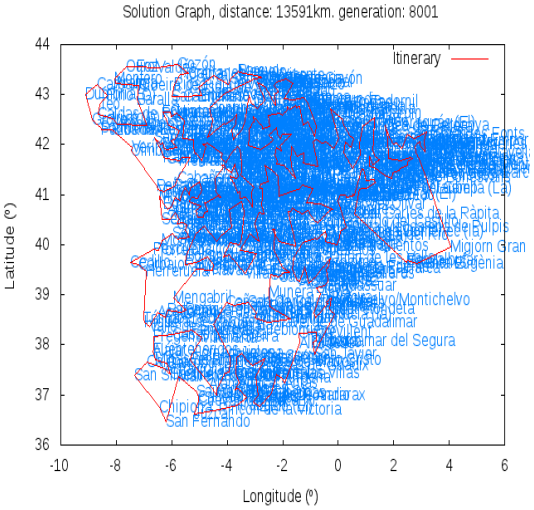
(b) Solution for $n = 50$



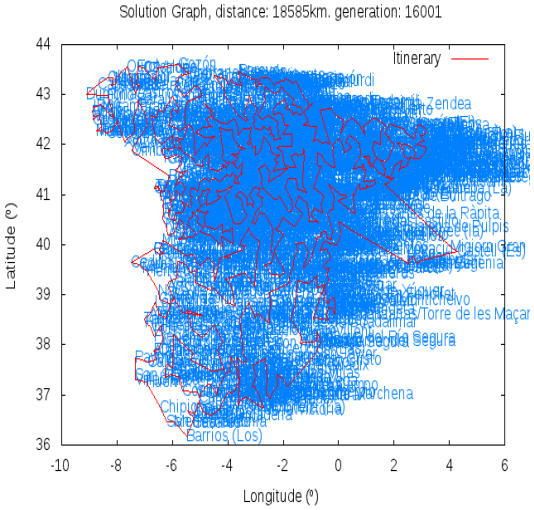
(c) Solution for $n = 125$



(d) Solution for $n = 250$

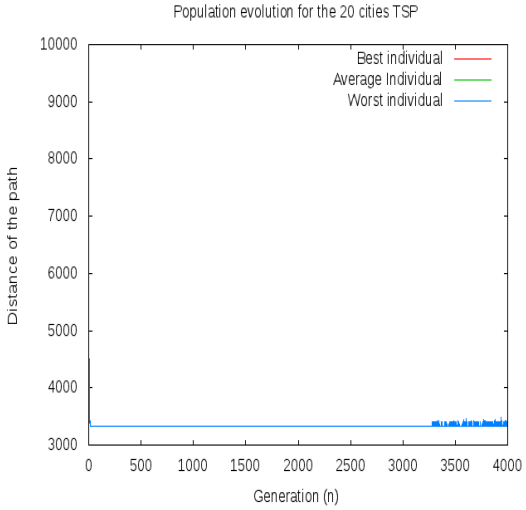


(e) Solution for $n = 500$

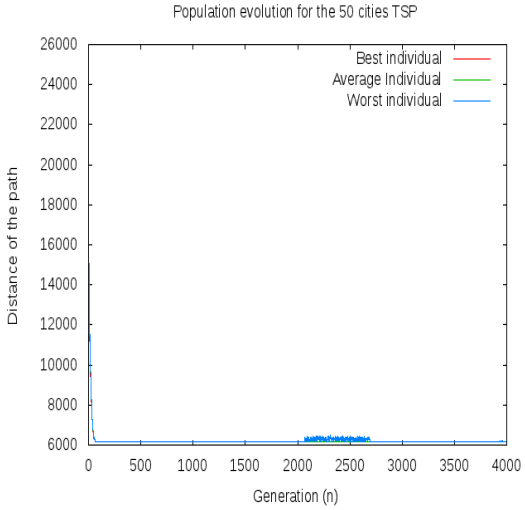


(f) Solution for $n = 1000$

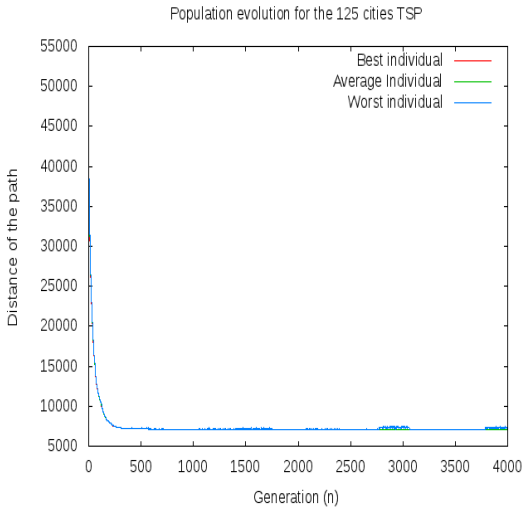
Figure 13: Solutions for the TSP, using *inversion* mutation, *order* crossover, *probability of mutation* 0.7, *probability of recombination* 0.2, *shape factor* 0.2 and *population size* of 100



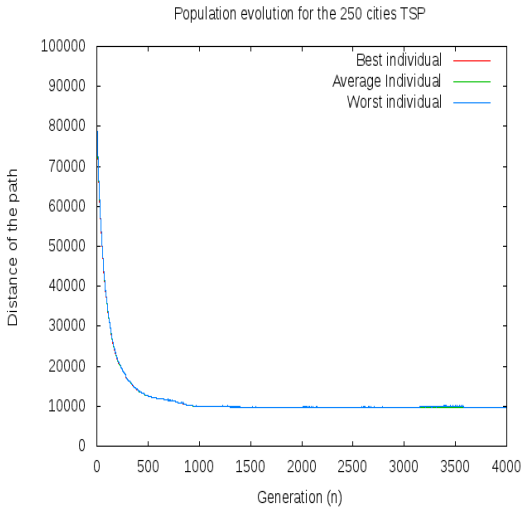
(a) Population Evolution for $n = 20$



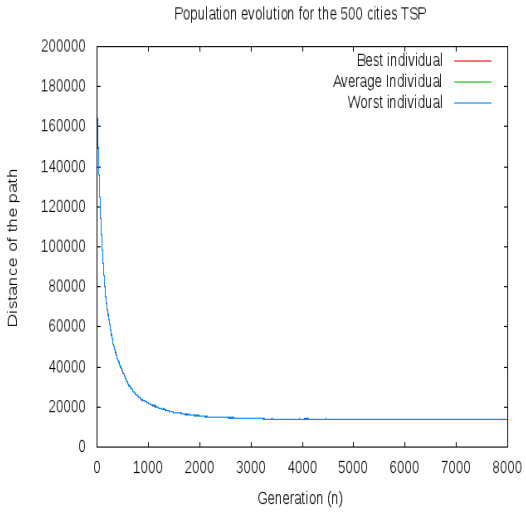
(b) Population Evolution for $n = 50$



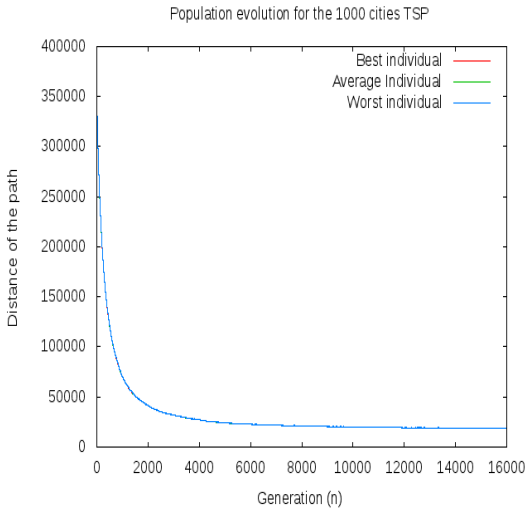
(c) Population Evolution for $n = 125$



(d) Population Evolution for $n = 250$



(e) Population Evolution for $n = 500$



(f) Population Evolution for $n = 100$

Figure 14: Evolution of the population for the TSP, using *inversion* mutation, *order* crossover, *probability of mutation* 0.7, *probability of recombination* 0.2, *shape factor* 0.2 and *population size* of 100

6 Appendix

A Program Compilation

Program building process has be done using the standard make gnu tool. To compile and link the program just invoke the `make` command on the root directory. The binaries will be found either under the `bin/debug` or `bin/release` directories, depending on the `dbg` flag specified at `config.mk` file. To link the program and generate the binary, we need the Boost Libraries properly installed in our system. The `libraries.mk` file contains the list of libraries that we need for the linking process.

B Program Execution

The program `TSP.bin` is invoked from the command line, passing all required parameters needed to the execution of the algorithm. For a comprehensive list of parameters, we can invoke:

```
victor@kokomero:~/TSP.bin --help
Allowed options:
  --cities arg                          Path to the csv file storing the cities
                                        and their coordinates
  --numCities arg (=0)                  Number of cities to be sampled from the
                                        city file. 0 or omit for all cities.
  --mutation arg                        Mutator operator, one of: swap,
                                        insertion, scramble, inversion
  --crossover arg                       Crossover operator, one of: pmc, edge,
                                        order, cycle, optedge
  --selection arg                       Selection operator, one of: ranking,
                                        fitness
  --generations arg                     Number of generations
  --population_size arg                 Number of individuals in the population
  --mutation_prob arg (=0.25)           Mutation Probability
  --crossover_prob arg (=0.70)          Recombination Probability
  --shape_factor_selection arg (=0.10)  Shape factor to determine how selective
                                        we are on the probability distribution
                                        for mating and survival selection
  --test                                Run mutation and crossover tests
  --help                                print out help message
```

The parameters with a default argument provided, `numCities`, `mutation prob`, `crossover prob` and `shape factor selection` do not need to be specified in the command line. If not specified, the program will consider the default values. Any other parameter should be specified, obtaining an error message with the missing compulsory parameter if we fail to do it.

An example of successful program invokation could be:

```
victor@kokomero:~/TSP.bin --cities "../../data/ciudades.csv" --numCities 200
--mutation inversion --crossover optedge --population_size 200 --generations 600
--mutation_prob 0.45 --crossover_prob 0.45 --shape_factor_selection 0.01
--selection ranking
```

which will find the solution for the TSP, using 200 cities randomly sampled from file `ciudades.csv`, using *inversion* as mutator operator, *optimized edge-3 crossover* as recombinator operation, a mutation probability of 0.45, a recombination probability of 0.45, a population size of 200 individuals at each generation,using *ranking selection* as selection method, with *shape factor* of 0.01 for the assigment of

selection probabilities in the *ranking selection*. The algorithm will run for 600 generations and will stop at that point, returning the best individual of the last generation.

In order to test the program, we provide several dataset files with an increasing number of cities. The data file can be found under the `data` directory. Files with a number at the end of the filename contain that given number of cities, and can be used to test the algorithm with a given size without specifiing the `--numCities` parameter. This way we will get always the same cities at each simulation, which can be useful for the debugging process. The file `ciudades.csv` contains 8023 cities from Spain.

```
victor@kokomero:~/ls data/
ciudades1000.csv ciudades125.csv ciudades2000.csv ciudades250.csv
ciudades30.csv ciudades4000.csv ciudades500.csv ciudades5.csv ciudades.csv
ciudades10.csv ciudades15.csv ciudades20.csv ciudades25.csv
ciudades35.csv ciudades40.csv ciudades50.csv ciudades8.csv
```

After an execution, a set of files in the current working directory will be generated. For the program invocation above, we would get the following results printed on the standard output:

```
GA Parameters:
Mutation Probabiliy: 0.45
Recombination Probability: 0.45
Max number of Generations: 600
Population size: 200
Mutator operator: Mutator: Inversion
Cross Over operator: Crossover: Optimized Edge
Selection operator: Ranking Selection with exponential probabilities

Execution of the algorithm:
Generation: 150
Generation: 300
Generation: 450
Generation: 600
Itinerary for the TSP:
Villalbilla de Gumiel,Valderrobres,Villalba de Guardo,Finestrat,Alcañizo,
Albanchez de Mágina,Espartinas,Garcibuey,Castro del Río,Massanassa,Salmoral,
Soria,Quéntar,Betanzos,Baños de Montemayor,Reyero,Pomer,Fortaleny,
Sant Esteve de Palautordera,Almiserà,Sueca,Balmaseda,Escacena del Campo,Salmerón,
Arroyomolinos,Mesas (Las),Torrecilla del Monte,Alcalá del Júcar,Muriel,
...
```

And in the working directory the plot files, in `.png` format, containing the evolution of the GA results for each generation. The `*solutionmap*.png` files shows the given solution path for a given generation. Finally the `.csv` file contains the numerical data of the best, worst and average fitness in the population, as well as the user-CPU time consumed at eatch generation.

```
victor@kokomero:~/ls
inversion_optedge_popsize200_muP0.45_recomP0.45_shape0.01_problemsize_200_bestfitted.png
inversion_optedge_popsize200_muP0.45_recomP0.45_shape0.01_problemsize_200_bestfitted_vs_time.png
inversion_optedge_popsize200_muP0.45_recomP0.45_shape0.01_problemsize_200.csv
inversion_optedge_popsize200_muP0.45_recomP0.45_shape0.01_problemsize_200_range.png
inversion_optedge_popsize200_muP0.45_recomP0.45_shape0.01_problemsize_200_solutionmap_150.png
inversion_optedge_popsize200_muP0.45_recomP0.45_shape0.01_problemsize_200_solutionmap_300.png
inversion_optedge_popsize200_muP0.45_recomP0.45_shape0.01_problemsize_200_solutionmap_450.png
inversion_optedge_popsize200_muP0.45_recomP0.45_shape0.01_problemsize_200_solutionmap_600.png
inversion_optedge_popsize200_muP0.45_recomP0.45_shape0.01_problemsize_200_solutionmap_best_.png
```

References

[1] Robert Demming and Daniel J. Duffy. *Introduction to the Boost C++ Libraries, Volume I - Foundations*. Datasim Education BV, 2010.

[2] A. E. Eiben and J. E. Smith. *Introduction to evolutionary computation*. Natural computing series. Springer-Verlag, 2003.

[3] Keld Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.