University of Padua

Master Degree in Computer Engineering

A.Y. 2019/2020

Operations Research Course Report

Travelling Salesman Problem

Barison Marco

Moratello Matteo

Obetti Christian

Contents

Chapter 1

Introduction

Qui un intro sul problema TSP direi

Chapter 2

Compact Models

Qui la parte dei modelli compatti

Chapter 3

Exact Algorithms

qui la parte degli algorimti esatti

Chapter 4

Matheuristic

Qui la parte dei matheuristic (hard fixing e local branching)

Chapter 5

Stand-alone heuristics

On this chapter we will illustrate some stand-alone heuristics. These algorithms are so called because they provide an heuristic solution without relying on CPLEX or any other solver. The literature in this case is huge: there are plenty of different approaches and each of them can have multiple variations. On this report we will cover some of the major ideas.

5.1 Heuristic solution builder

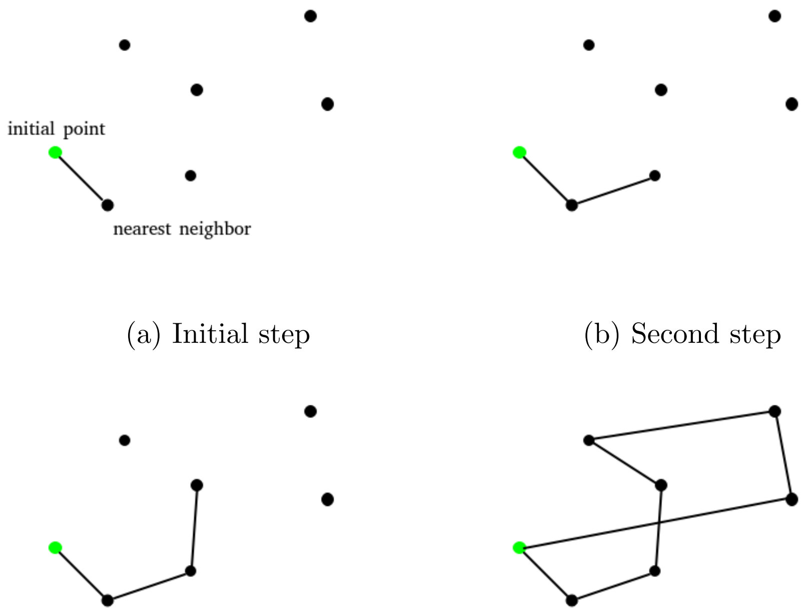
On this section we will illustrate some heuristics algorithm to generate in a rapid way a solution for the TSP problem. Of course the results provided by these algorithms are admissible solutions but not the optimum solution.

5.1.1 Nearest neighborhood

Nearest neighborhood is a greedy algorithm [1] that works in the following way: start from a random point that is considered the first visited node. Then pick as next node the nearest to the last visited (greedy choice) and iterate until all the nodes are visited. This procedure can be repeated until a time limit is reached, each time starting from a different random point, keeping in memory the best solution and eventually update it if a new one is found. The algorithm is pretty simple but provide an admissible solution (that of course is not the best one and in the very most of the cases is quite far from the optimum) in a short time.

On our implementation we decided to trade space for time. Since the operation to compute the nearest point is repeated several time, we decide to store for each point the list of all others points sorted by their distances from that one. This require $O(n^2)$ additional space. Considering the time, instead, to sort the points we implemented a simple *insertion sort*, that is $O(n^2)$, so in total we need an initial $O(n^3)$ (This part could be further improved with a better sort algorithm like *merge sort*

[1] algorithm that always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.



(a) Initial step (b) Second step

(c) Third step (d) Final result

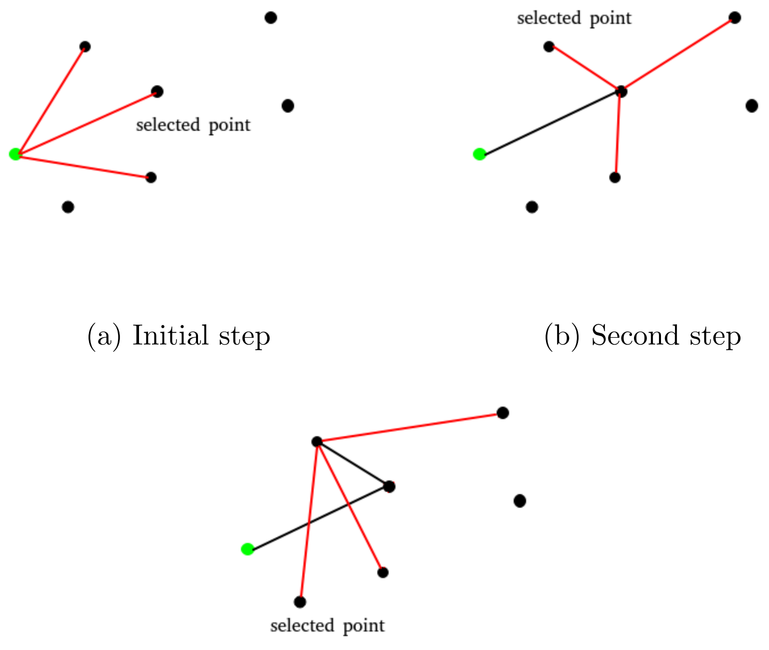
Figure 5.1: Execution of nearest neighbor algorithm

that is $O(n \cdot log(n))$, but since the initial time to order the nodes on our experiments was negligible, we decided to use insertion sort that was easier to implement). So, by paying this initial cost we were able to speed up each iteration of nearest neighborhood, in fact each time we needed to find the nearest point we just had to pick the first available one from the ordered list. Of course in the worst case this is $O(n)$, because it can happen that the firsts elements of the ordered list have already been visited, so I have to run across most of the list until I find an available point. However, what we saw in practice is that, except when there are left few nodes to visit, the first available element is always on the firsts position of the ordered list so the operation to find the nearest neighbor became $\Theta(1)$ in most of the cases (for example, we tried on multiple instances between 100 and 500 points and the average number of access to the ordered list to find the nearest point was between 3 and 6).

The major problem of this technique is that visiting always the next closest point leads to the creation of lots of intersection between edges, that of course are inefficient. A possible solution to improve this initial result will be shown on the next section dedicated to the refining algorithms.
Another problem is that this algorithm is deterministic, so this mean that if it is run twice starting from the same initial point, it produce the same solution. The consequence of this is that the number of different circuit that it can explore is limited to the total number of nodes.

5.1.2 GRASP

GRASP is a variation of nearest neighborhood that introduce a random component. On each iteration, rather than picking the nearest point to the last visited, select the three closest points and choose one of them at random as next visited vertex.
Our implementation was almost the same of nearest neighborhood, with the construction for each point of the initial list of all other points sorted by their distance from that point, with the only difference that first three available points are picked from the list, rather than only one.
The introduction of the random component means that the probability to repeat the same sequence starting from the same point is very minimal, so GRASP can explore lots of different circuit resolving the problem of determinism of nearest neighborhood.



(a) Initial step (b) Second step

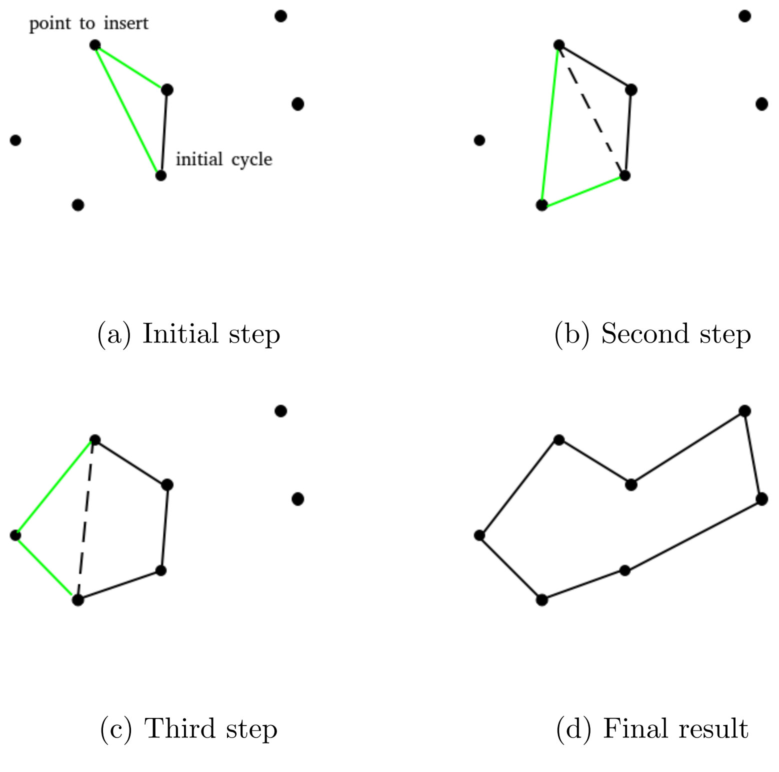
(c) Third step

Figure 5.2: Execution of 3 steps of GRASP. Each time the 3 closest points are considered and one is picked at random.

5.1.3 Insertion heuristic

This algorithm starts from a simple circuit made with only two points. At each iteration, for each of the remaining vertices to insert, compute the minimal extra mileage. The extra mileage is the cost to insert a point on the circuit. So, for example, if we want to insert the vertex z between vertex x and vertex y, the extra

mileage is: $c_{xz} + c_{zy} - c_{xy}$, where c_{ij} is the cost of the edge (i, j). The fact that for each point we search the minimal extra mileage means that we are looking to the best position where insert that point. Insert vertex that has the best extra mileage into the partial circuit and repeat the procedure until all the points are inserted. The algorithm than can be executed several time, starting every time from different initial points to explore new solutions and updating the incumbent if necessary.
The cost to compute the best extra mileage is $O(n^2)$, so this mean that the entire algorithm is $O(n^3)$.
It is also possible to add the GRASP rule, so rather then selecting the point with the best extra mileage, find the three smallest extra mileages and choose at random between them, in this way the algorithm doesn't converge to the same solution starting from the same initial circuit.



(a) Initial step (b) Second step

(c) Third step (d) Final result

Figure 5.3: Execution of insertion heuristic algorithm

On our implementation to select the initial circuit we decided to pick at random one point and then to select the furthest point from that one. (Of course this is not the only solution: other possibility are, for example, to pick a vertex and its closest one, or to choose two points completely at random.) I addition, at the beginning of the algorithm, we built a matrix where the element i j contains the distance between vertices i and j (So in total we payed an additional $O(n^2)$ in space). We made this choice because on this algorithm distances between points are computed several times and in addition the same distance can be calculated more than once.

5.1.4 Insertion with convex hull

This is a variation of the insertion algorithm. It works in the same way but the initialization is different: rather than starting from an initial cycle made of two random points, compute the convex hull [2] of the instance. Then iterate and insert all the remaining points, that are all inside the convex hull, in the same way of Insertion heuristic.
On our implementation, to find the initial convex hull, we used the Graham Scan algorithm, freely provided by the website www.geeksforgeeks.org. This algorithm is $O(n \cdot log(n))$ and it is performed only once at the beginning so it doesn't impact the overall execution time for the insertion that is $O(n^3)$ as explained on the previous subsection. For more information about the Graham Scan consult `https://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/`

5.2 Refining algorithms

Refining algorithm are so called because they start from an admissible solution that could by provided by any other algorithm and they try to improve it with a series of iteration.

5.2.1 TWO-OPT

The first algorithm we present is TWO-OPT (or 2-OPT). To explain this technique, let's consider two edges, the first delimited by vertices a and b and with cost c_{ab}, the second delimited by c and d and with cost c_{ab} (Both edges belong to an admissible solution provided by any algorithm). Now let's remove these edges and reconnect the vertices in a different way: as result we obtain the edges a, c and b, d with cost respectively c_{ac} and c_{bd}. Now lets compute:

$$\Delta = c_{ab} + c_{ab} - (c_{ac} + c_{bd})$$

that is the cost of the initial removed edges minus the cost of the new edges. If $\Delta > 0$, it means that the new edges are better than the older and the swap is worth because it reduce (improve) the objective function. Of course the new solution produced is still admissible (Figure 5.??).

2-OPT works in this way: start from an admissible solution, consider all the possible couple of edges (that are in total $O(n^2)$) and for each of them compute the Δ (That can be calculated in $\Theta(1)$). Then pick the largest Δ, remove the relative edges and

[2]In geometry, the convex hull or convex envelope or convex closure of a shape is the smallest convex set that contains it.

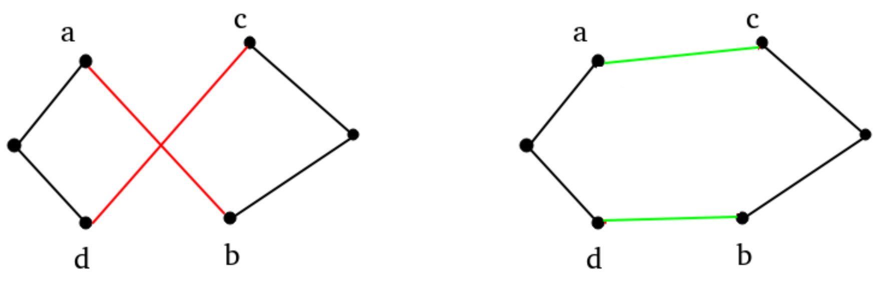


Figure 5.4: The 2-OPT move, edges in red are removed and replaced by green edges.

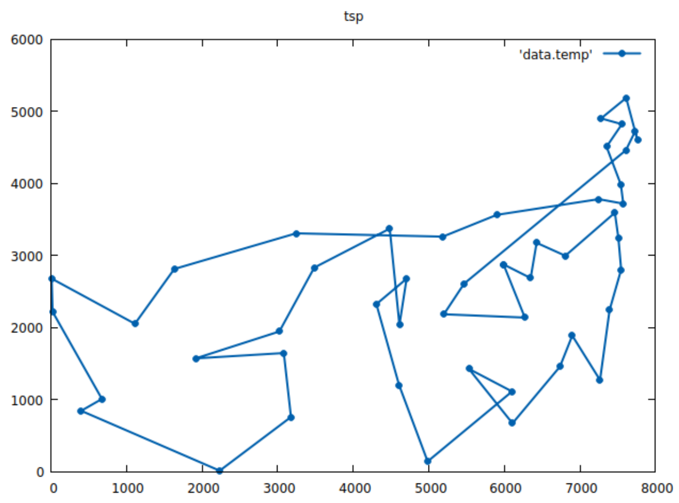
reconnect the vertices as previously explained. After this operation, one of the two parts of the cycle that has been divided by the swap, need to be retraced in order to invert the visit sequence of the vertices and reconnect it to the new edge (This is $O(n)$, but since it is done after a $O(n^2)$ doesn't change the overall cost of the swap). Repeat the procedure until all the Δ are negative, or a fixed time limit is reached.

This algorithm is very good to remove edge intersections, like the ones produced by nearest neighborhood (or GRASP). Here in the table below some examples.

Instance	**GRASP**	**2-OPT**	**Improve**
	(5 sec)	(5 sec)	
att48	41806	35522	15%
eil101	879	691	21%
ch130	9801	6347	35%
ch150	10477	7278	30%
a280	4546	2886	36%
lin318	76940	46235	39%
att532	155010	97487	37%
pr1002	492311	285078	42%

Table 5.1: Values of the objective functions after running on each instance GRASP algorithm for 5 seconds and then 2-OPT for other 5 seconds.

These results were obtained after running GRASP for five seconds and then refining the solution with 2-OPT for other five seconds. Of course this dataset is very small but it gives the idea of the power of this technique, we have, in fact, a significant improve of the objective function in all the cases. To notice that the improve increase with large problem.
It is possible to visualize the action of 2-OPT by plotting the solution before and after the refining:



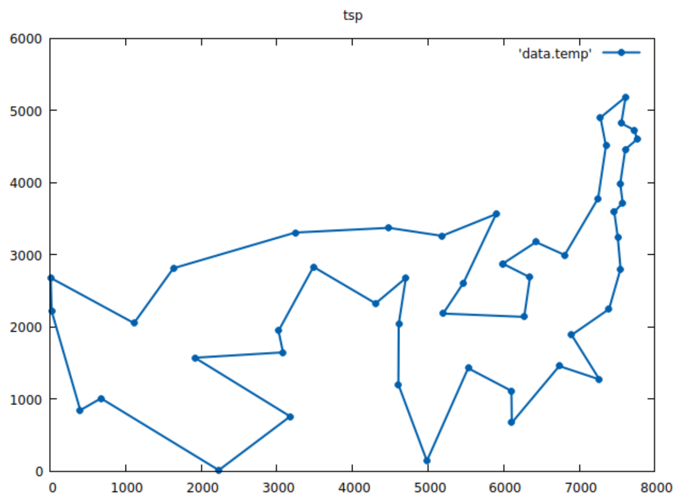


Figure 5.5: Solution of 5 sec of GRASP before (top image) and after (bottom image) a refining of 5 sec

5.2.2 THREE-OPT

3-OPT is a variation of 2-OPT that consider 3 edges for the swap rather than only two. This time there are multiple way to reconnect the vertices of the 3 removed edges, in particular for a triad there are 7 possible moves. Moreover, I have to consider all the possible triad to find the best one and this is $O(n^3)$. So in general 3-OPT is more complicated to implement and due to the fact that is cubic doesn't work well with big problems, for example with instances with more than 10000 points.

5.3 Metaheuristics

A metaheuristic is a higher-level procedure or heuristic designed to find, generate, or select a heuristic (partial search algorithm) that may provide a sufficiently good solution to an optimization problem, especially with incomplete or imperfect information or limited computation capacity [3]

5.3.1 Multi-start

After performing a refinement technique, like 2-OPT, if the time limit is not reached, the algorithm stop on a local minima of the objective function. Of course at this point we want to escape from this local minima and search for a better one that hopefully is also the optimum. A possible solution is the multi-start approach, so each time we restart from the beginning, providing another initial solution and refining it. On each iteration, if the algorithm that provide the initial solution is well randomized (like GRASP) we will stop on a different local minima. So we can keep track of the best one and return it when the time limit for this procedure is reached. The main problem of this technique is that is not very efficient because each time I restart all the information we obtained on the previous local minima is unused. More powerful technique will be shown on the following subsections.

5.3.2 Variable neighborhood search

Variable neighborhood search (VNS) provide a simple yet powerful way to escape the local minima. Let's assume that we have completed the refinement using 2-OPT. At this point we can perform a random move with a bigger research radius, a 3-OPT or a 5-OPT for example. Since this move is completely random, it will make the solution worse but it will also provide the "kick" that we need to escape the local minima. Now we can restart with 2-OPT until a new minima is found (that could also be the optimum).
In synthesis, VNS, works with two phases that alternate continuously: an intensification phase (2-OPT) where the algorithm move toward a minima and a diversification phase where VNS perform a 3-OPT or greater random move to exit from the minima.

5.3.3 Tabu search

Created by Fred W. Glover during the 80s, Tabu search provide an alternative procedure to VNS to escape from local minima. Let's assume that we have performed a refinement and we have reached a local minima, let's call this solution x_k. At this point, every 2-OPT move will make the objective function worse. So we choose and perform the least pejorative move, obtaining the solution x_{k+1}. Now, if we make

[3]R. Balamurugan; A.M. Natarajan; K. Premalatha (2015). "Stellar-Mass Black Hole Optimization for Biclustering Microarray Gene Expression Data". Applied Artificial Intelligence an International Journal.

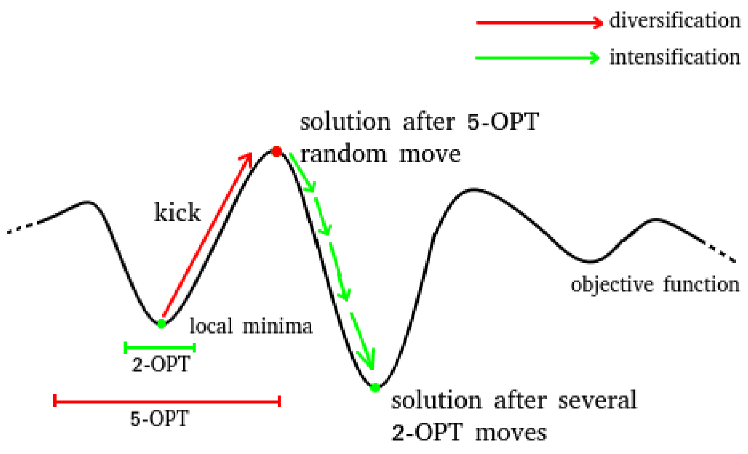


Figure 5.6: Representation of an objective function and the "kick" of a 5-OPT random move

another move following the 2-OPT rule, the algorithm will return to the solution x_k by simply swapping again the edges used on the last step, cause this is the only move that improve the objective function. To avoid this, the idea is to insert these edges on a tabu list that is a list of edges that cannot be used for a 2-OPT move. This means that the move from x_k to x_{k+1} became forbidden. So we repeat this last step, performing the least pejorative move and each time inserting the edges used in the tabu list, until we reach a point where the objective function will return to improve. Of course we don't insert edges into the tabu list if the move is not pejorative.
To describe the algorithm we asserted that the tabu list is made of couples of edges, but this was only for a description purpose. In fact the general idea for this algorithm is that the tabu list can be made of any element that prevent to return back after a pejorative move. So for the TSP problem we can use, for example, couples of edges, a single edge or a single point.

The handle of the tabu list is very important and different strategies can have a big impact on the algorithm. Of course if we continue inserting element we will reach a point where there aren't possible moves. So we limit the size of the tabu list. The size is called tenure. When the list is full, the older element are removed. The tenure is another parameter to set. A possible way to do this is to set a minimal and a maximal value and let the tenure oscillate among them during the algorithm. So we will have diversification phases when the tenure is big and there are lot of forbidden move and intensification phases when the tenure is small and the algorithm has lot of freedom. This is called reactive tabu search. Another shrewdness is that if we encounter a move that is forbidden but that improves the incumbent, we will not perform that move but we will update the value of the incumbent in any case

(Aspiration criterion).

5.3.4 Simulated Annealing

This metaheuristic algorithm take inspiration from the annealing technique in metallurgy that consist on heating the material and then cooling it in a controlled way. When the temperature of the materials is high, the atoms inside it it vibrates and move in a chaotic way. As the temperature decreases, the atoms reduce their state of agitation until a stable point is reached.
The simulated annealing works in a similar way. Let's establish a parameter, the temperature T, that initially has an high value that gradually reduce during the iterations of the algorithm. The algorithm starts from an already provided solution. On each iteration we perform a random 2-OPT (or 3-OPT) move that of course can be also pejorative. The fundamental idea is that the probability to accept the move is related to the temperature: when T is high, the system is chaotic and we accept with high probability all the random move. As the system get cooler we increase the probability to reject a bad move (so we increase the probability to perform a good move). When the temperature is close to 0, we only make meliorative moves until we reach a minima.
Considering the objective function, we have that, when T is high, the algorithm is randomly exploring solutions. Only when T became low the algorithm moves toward a minima that hopefully is the optimum.

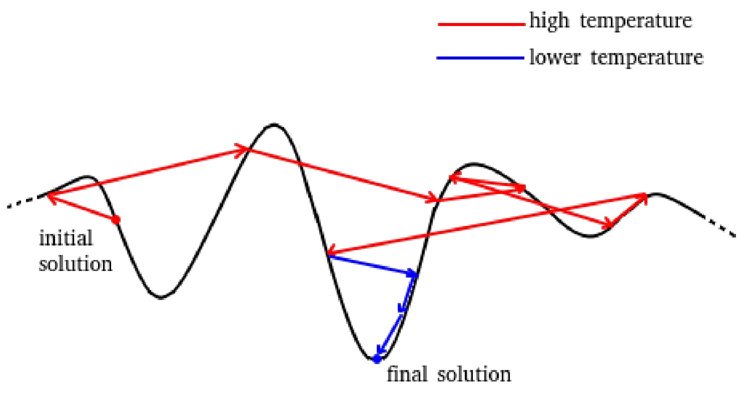


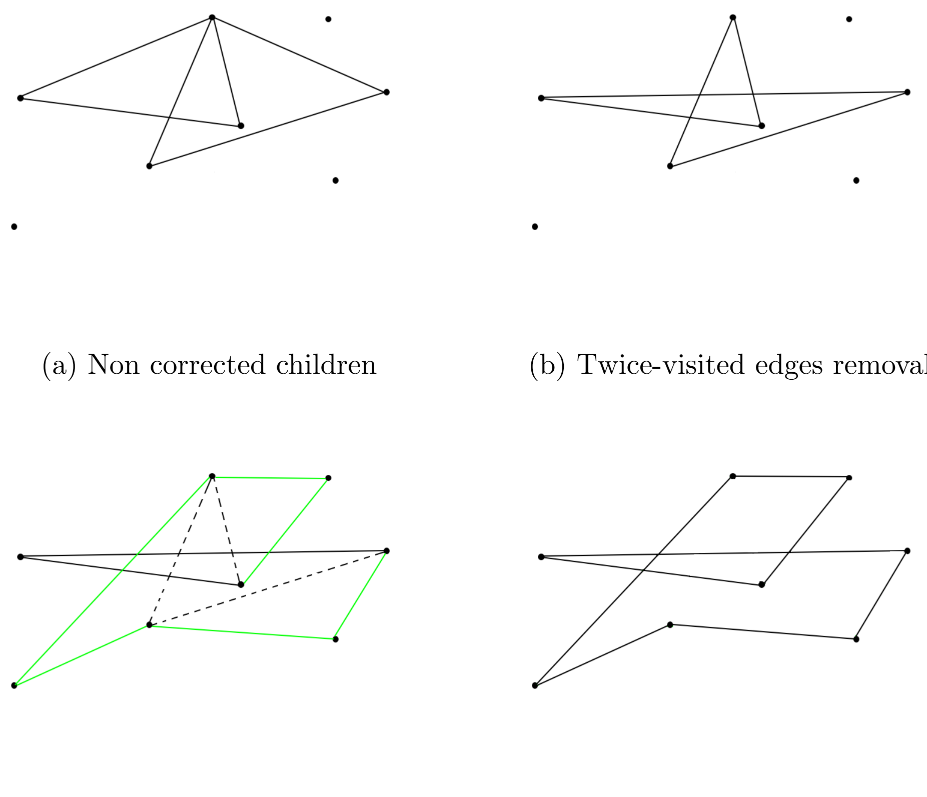
Figure 5.7: Example of the work of the simulated annealing. Note the initial random exploration with high temperature and the move toward the minima with a lower temperature

5.3.5 Genetic Algorithm

The genetic algorithm is inspired by the evolution theory, formulated by Charles Darwin during the 18th century. The algorithm starts with a population where each individual that belong to the population itself is a solution to the problem, so in our case each individual is a solution for the TSP. Each individual has also a fitness that is defined as the cost of the solution. On each iteration, the individuals that have the highest fitness are removed from the population: maintaining the parallelism with the evolution theory we can say that the individuals that didn't adapt to the environment (high fitness) died. On each iteration we have also that couples of individuals reproduce and the new elements substitute the ones that we have removed (so the population size remain constant). The children generated inherit characteristics from both their parents: a simple mechanism is just to pick part of the edges from one parent and the remaining from the other. Of course in this way the children is not a solution of the TSP problem cause some node may be visited more than once or may miss, so a corrective function is required.
After some iteration we have that the average fitness improve due to the fact that we remove bad solution and we generate new solution from the better ones. When the algorithm end, the champion of the population, that is the solution with the best fitness, is the final result.

Our first implementation of this algorithm was pretty simple. We generated a population of a size decided by the user using the GRASP algorithm. We created a probability mass function (PMF) normalizing the sum of the fitness, so on each iteration we randomly chose the elements to remove based on that mass function. This means that during the firsts iterations, when all the individual have almost the same fitness, the element are almost randomly remove, while when the algorithm progress and there is a more evident difference between good and bad individuals, with high probability only the worst solutions are removed. The number of element to remove on each iteration (that is equal to the number of children to generate) is a parameter decided by the user.
To reproduce two elements we used the following technique: we chose a random number x between 0.3 and 0.7. Then we picked the first x visited node from one of the parent and the remaining from the other. At this point we used a corrective function that remove all the points visited twice and insert the missing ones using the insertion algorithm described in 5.1.3.

The couples to reproduce are simply chosen at random from the population and we didn't implement any mechanism to control that same individual aren't choose more than once. We made this choice because on this algorithm the population have often big size, in general more than 10000 individuals, so the probability to pick the exactly same individuals more than once is very low. After some run, plotting the average fitness and the best fitness, we could observe the expected behaviour (Figure 5.??): the average fitness continuously decrease until it get closer to the best fitness (that improve only sometimes).


(a) Non corrected children (b) Twice-visited edges removal

(c) Insertion of remaining element (d) Final result

Figure 5.8: Example of the steps of a correction function

The problem of our implementation became evident after testing it on different instances: the solutions had lot of intersection (like the ones generated by the GRASP algorithm) that of course don't belong to the optimum. An example of this can be seen on figure 5.??. So we tried to introduce the 2-OPT someway inside the genetic. A possibility that we first considered but we immediately discarded was to just apply the 2-OPT algorithm on the final solution provided by the genetic. This didn't make lot of sense because we did a lot of effort to produce a result with the genetic and then we discard a consistent part of it using the 2-OPT: a simple GRASP plus 2-OPT would be more efficient.
Then we decided to add the 2-OPT to the correction function

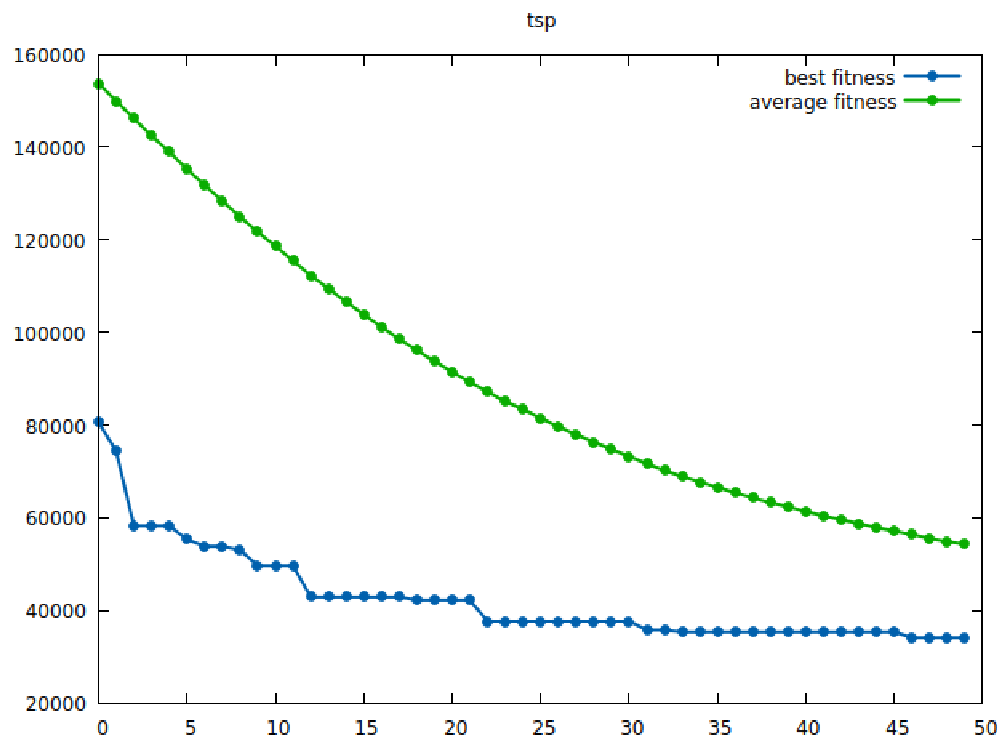


Figure 5.9: Plot of the average fitness and best fitness during the execution of a genetic algorithm.

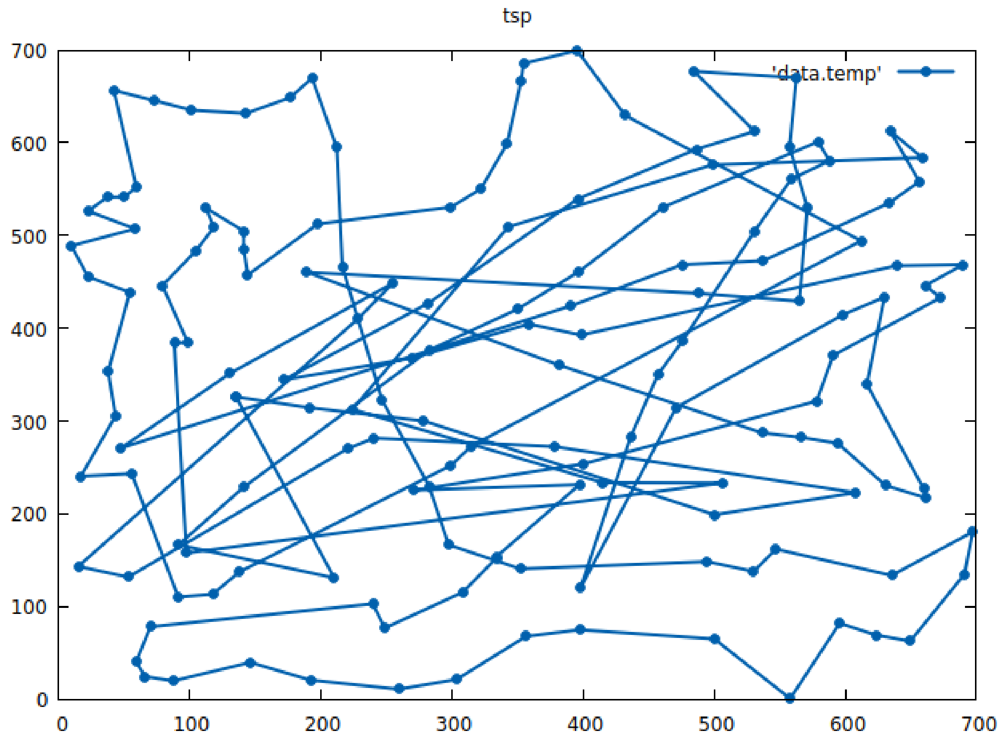


Figure 5.10: Solution of the instance ch150 provided by our first implementation of the genetic. Note the big number of edge intersections.