Third Assignment

INTELLIGENT SYSTEMS

- **Víctor Pérez Piqueras**
- **José Javier Atiénzar González**

UCLM | ESIIAB

Content

Figures

1. Problem Description

The problem that will be faced is the Travelling Salesman Problem (TSP). A salesman must follow a route all weeks. The route starts in a store, goes through a set of cities where the goods are delivered, and finally finishes in the initial store again. The problem consists of finding the order he must visit the cities so that the distance of the whole tour is minimum.

The goal of the search is basically to answer this question: given a set of N cities and the distance between each pair of cities, what is the shortest route that visits each city once and returns to the origin city?

This problem is impossible to be solved doing an exhaustive search, because the number of possibilities is factorial of the number of cities given.

Here we can see 2 possible representations of the problem.

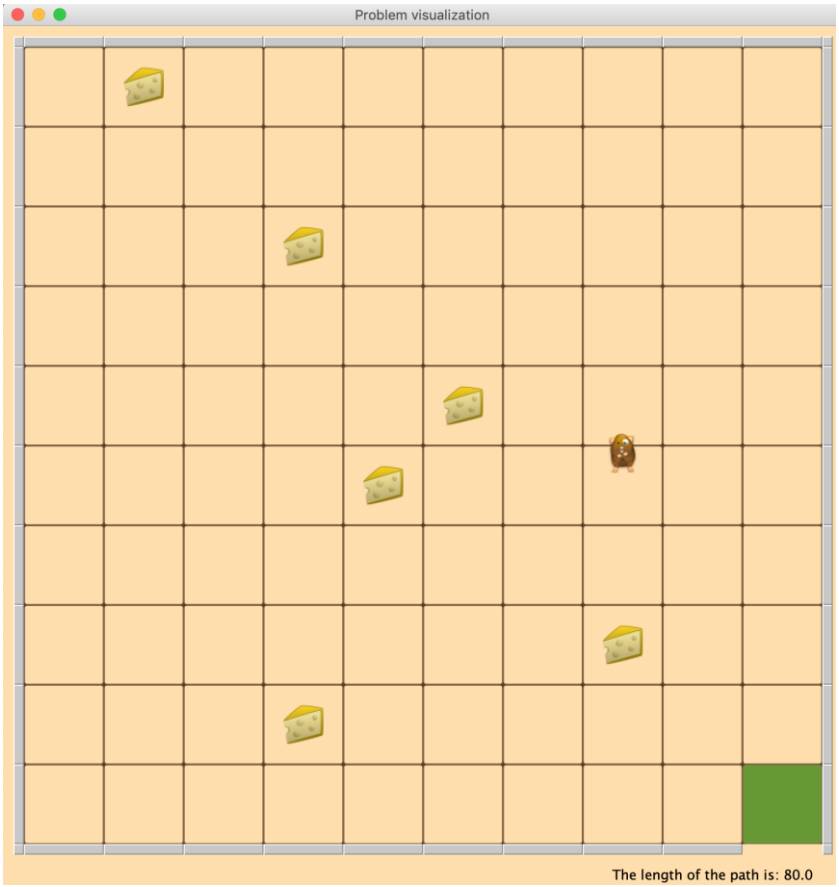


Figure 1. TSP problem with cheeses and hamster.

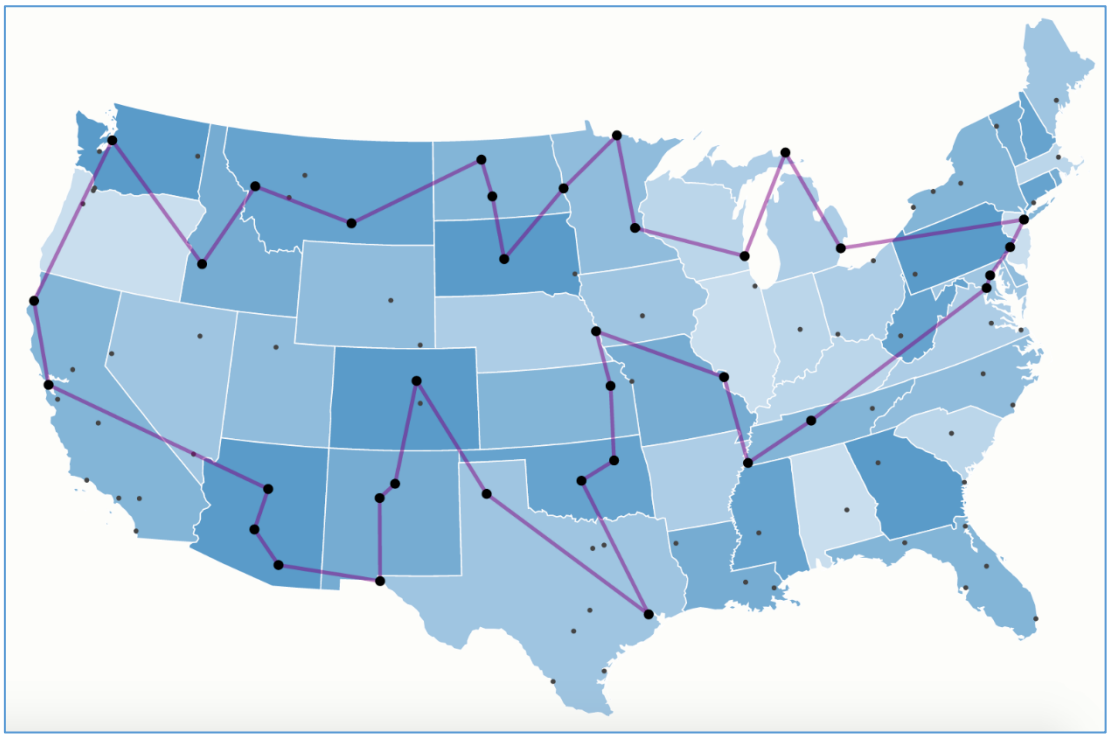


Figure 2: Representation of TSP with EEUU map.

2. Metaheuristics:

TSP is a NP problem and so, we cannot be sure that we will find the best possible solution. To solve this, we make use of the metaheuristics:

They are high-level and problem-independent procedures. They can be used as black box optimization methods since they don't take advantage of the specification of the problem.

Metaheuristics allow obtaining quality solutions in short times. They are used when the search space is very big, for example for permutations, the number of solutions for a problem with size n is close to $n!$. And where there is no information enough about the problem to use an exact solving method.

They make use of two fundamental resources and the usage of information to progressively build better solutions and improve them. This information can be obtained from certain features of the problem that allow evaluating steps in solution building. Like evaluating distances among cities in TSP. And the structure of solutions previously evaluated and their values.

These algorithms should not get stuck examining similar solutions (exploration) and take advantage of the information in order to address the search towards promising regions in the search space (exploitation).

There must be a balance between them.

We can classify these metaheuristics as:

- Trajectory based: Starts with a random solution that becomes a better one eventually.
- Constructive: They use randomness and information to build solutions in steps.
- Population based: Use a set of solutions and its information in order to generate new sets of solutions.

3. Formalization:

To set the problem as an optimization problem we need to define the following:

- **Representation:** the way to codify a solution that we use is: (x1,x2,…,xn). A set of cities in which the index ordering is the order in which the cities are visited. So x1 will be visited and then x2, etc.
- **Evaluation:** to evaluate the solutions and differentiate which are the better we use a function that calculates the distance from the first city to the next and so, calculating the whole route and returning the total distance.
- **Optimization criterion:** for this problem the goal is to MINIMIZE the score or distance. The best solution is the one that returns the shortest path.

4. Algorithms:

There are a bunch of algorithms that make use of metaheuristics. We have selected 3 types that help us to see the different properties and possibilities that bring us their application on the problem. The algorithms can be divided in 2 types:

4.1 Local Search:

Introduces the concept of Neighbourhood. N(x) is the set of configurations within a distance of the configuration x. For this problem, a "distance" can be defined as the maximum level of difference between configurations. I.e. a distance 2 could be having a configuration that differs with other in 2 cities that are visited in a different order. The algorithms that we have implemented are the following:

- Hill Climbing. It is the basic local search algorithm, it is based in the neighbourhood criterion to carry out the search. Is a trajectory-based heuristic, since they build a trajectory from the initial configuration to the solution. Hill climbing stops in the best configuration of its neighbourhood, called local optimum. The configuration with the best score is called global optimum and it is not guaranteed to find it unless the search starts in a very good initial configuration. It is very efficient algorithm since it evaluates few configurations.

The pseudocode is the following, but changing the comparator of the score, to allow it to be a minimization problem→score<currentScore:

Hill climbing algorithm (maximization)

```
1:  function HillClimbing(initialSolution)
2:      currentSolution ← initialSolution
3:      currentScore ← Evaluate(currentSolution)
4:      improves ← True
5:      while improves = True do
6:          improves ← False
7:          neighbors ← GenerateNeighbors(currentSolution)
8:          for each neighbor en neighbors do
9:              score ← Evaluate(neighbor)
10:             if score > currentScore then
11:                 currentSolution ← neighbor
12:                 currentScore ← score
13:                 improves ← True
14:     return currentSolution
```

Figure 3: Hill Climbing pseudocode

- Iterated Local Search: This algorithm consists on applying Hill Climbing once, then perturb the configuration obtained an apply Hill Climbing again. To perturb means to modify the previous configuration. It is based on the idea that near a local optimum there can be some other optima which can eventually improve the final solution. The pseudocode is the following, as said in the pseudocode above, depending on the optimization criterion we can either find a minimum or a maximum changing the "<" operator in the comparison.

Iterated Local Search (maximization)

```
function IteratedLocalSearch( )
    solution ← NULL
    iter ← 0
    x ← GenerateRandomConfiguration( )
    while iter < MaxIters do
        x_p ← Perturbate(x)
        x' ← HillClimbing(x_p)
        if f(x') > f(solution) then
            solution ← x'
        x ← x'
    return solution
```

Figure 4: ILS pseudocode

4.2 Genetic Algorithms:

- They use a set of configurations, that we will call population. Based on the information in the best elements of the population we generate another population that normally leads to better configurations. Genetic algorithms need to evaluate much more configurations than the local search algorithms. They are called evolutionary algorithm since the population

changes over time and improves over the different generations. For each generation the best configuration in the population is selected and used to generate a new population. The generation of the population is done by applying crossover and mutation to the selected individual. Crossover takes two configurations and generates another two that inherit part of their features. Mutation makes random changes on genes independently. This is the procedure to apply it on a code:

Basic Genetic Algorithm scheme

$\mathcal{P} \leftarrow$ GENERATEPOPULATION() ▷ Creates candidate individuals (solutions)
EVALUATE($\mathcal{P}$) ▷ Obtains their score
while *stopCondition* = *false* **do**
  $\mathcal{P}' \leftarrow$ SELECTPOPULATION($\mathcal{P}$) ▷ Selects some individuals by score
  CROSSOVER($\mathcal{P}'$) ▷ Crosses pairs of selected individuals
  MUTATION($\mathcal{P}'$) ▷ Mutates the crossed individuals
  EVALUATE($\mathcal{P}'$) ▷ Obtains the score of the new individuals
  $\mathcal{P} \leftarrow$ COMBINE($\mathcal{P} \cup \mathcal{P}'$) ▷ Forms the new generation

Figure 5: Genetic pseudocode

- Selection: we have implemented 2 types of selection:
  - Tournament: Given a population it will compare the scores of the configurations and add the best one, the one with the lowest score.
  - Rank: The probability of an individual for being selected depends on its position in the ordering based in fitness.

- Crossover: Given two configurations it will randomly choose two points to have two points crossovers, this is called a 2PC. Then, it will keep the cities before the first point and the cities after the second point the same.
  The cities between the 2 points will be placed but following the order in which they appear in the other parent. So, having 2 starting parent configurations it will return 2 new children with permuted cities.

- Mutation: Given a population it will randomly swap two cities.

- Combine: We have implemented 3 types of combination:
  - Replacement: The new population will simply replace the older one.
  - Elitism: It will take the best configuration in the first population and replace for the worst one in the new population.
  - Truncation: It will truncate with the best configurations of both populations into the new one.

5. Implementation of the algorithms:

- **Hill Climbing:** Basic local search algorithm. At each step it moves to the best neighbour. It stops when no neighbour is better than the current configuration. Implements the following methods.

    - **Search:** Initializes the search and sets as the best solution the first random configuration. Using a Boolean called improves we will see if the neighbour is better or worse. While it improves it will check for all the neighbour configurations if there's any that is better than our current solution and generate a new neighbour based on the best solution. If there is a better one, we stop the search and set the new configuration as the best one, this is done in the evaluation method.
    - **GenerateNeighbours:** It will swap by one place the cities of the neighbour and add the configuration to the neighbour's array of configurations and return it.

- **IteratedLocalSearch:** Like the hill climbing problem but this time it will restart using a perturbed version of the first random configuration. It will do as many iterations as we choose.
    - **Search:** Initializes the search and sets as the best solution the first random configuration. For every iteration it will perturb the initial configuration and apply HillClimbing to it.
    - If it is better the solution will be the perturbed configuration with HillClimbing applied and will set the initial configuration to the HillClimbing one and do another iteration.
    - **Perturbate:** This method will randomly swap the values of two positions and return the new configuration.
    - **HillClimbing:** It will do the same as the Search method in the Hill Climbing algorithm explained before except initializing the first configuration.
    - **GenerateNeighbours**: It will swap by one place the cities of the neighbour and add the configuration to the neighbour's array of configurations and return it.

- **Genetic Algorithm:**

    - **Search:** Initializes the search and generates as many random generations ass population size we choose. We have a generation counter so that every generation is applied some methods. First, we will select population, then there's a chance that we have crossover for it, then we mutate it combine it and have a new generation.

- **SelectPopulation:**
  - Tournament: Given a population it will compare the scores of the configurations and add the best one, the one with the lowest score.
  - Rank: The probability of an individual for being selected depends on its position in the ordering based in fitness.
- **Crossover:** Given two configurations it will randomly choose two points to have two points crossovers. Then having 2 starting parent configurations will return 2 new children with permuted cities.
- **Mutation:** Given a population it will randomly swap two cities.
- **Combine:**
  - Replacement: The new population will simply replace the older one.
  - Elitism: It will take the best configuration in the first population and replace for the worst one in the new population.
  - Truncation: It will truncate with the best configurations of both populations into the new one.

6. Performance evaluation

Instructions of use:

In order to run the genetic algorithm in the arguments we have to write the following:

tsp.TSP 100 10 1 -- GeneticAlgorithm 200 100 100 10 rank replacement

Where rank can be replaced by tournament to use tournament selection. And where replacement can be exchanged by truncation or elitism to choose the combination strategy.

6.1 Local Search Comparison

- Firstly, we wanted to see the differences in applying the 2 algorithms that use local search and see which of them is more efficient.
- Executing the same problem for the 2 different algorithms we have obtained the following data:

Given the problem: *tsp.TSP 100 50 2*

Hill Climbing:

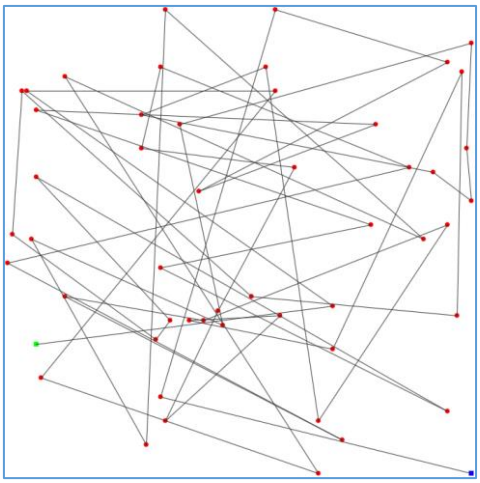


Figure 6: Hill Climbing solution

Best Score: 2673.78

Number of evaluations: 2451

We can see that it is not a very good solution.

Iterated Local Search:

- Number of Iterations: 100

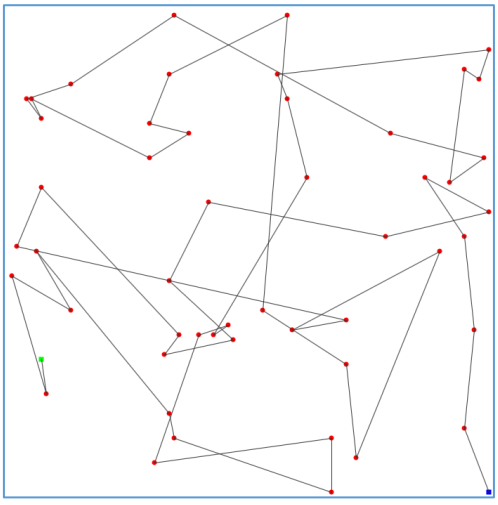


Figure 7: ILS 100 iterations

  - Best Score: 1074.63

- Number of evaluations: 245100

Choosing a small amount of iterations does not improve the solution obtained by the Hill Climbing algorithm.

- Number of Iterations: 500

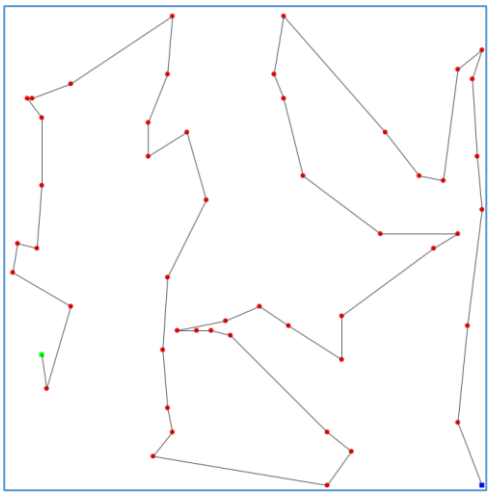


Figure 8: ILS 500 iterations

- Best Score: 633.03
- Number of evaluations: 1225500

Increasing the iterations of the algorithm the obtained solutions are better, but it makes the algorithm calculate much more evaluations.

- Number of iterations: 1000

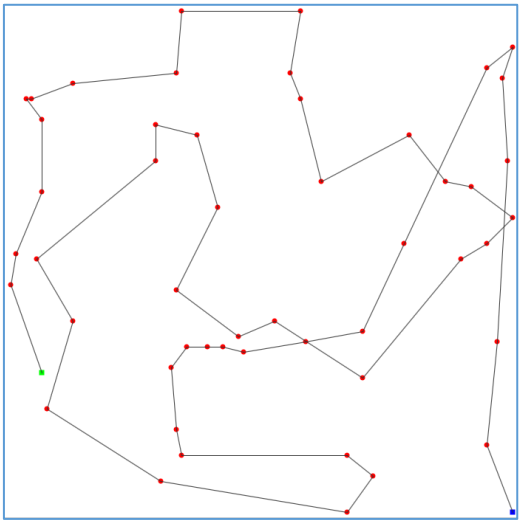


Figure 9: ILS 1000 iterations

- Best Score: 715.71
- Number of evaluations: 2451000

As the perturbation performed to the configurations is randomly performed, choosing a higher amount of iterations does not imply that the final solution will be always better.

- Comparison of iterations over the algorithms:

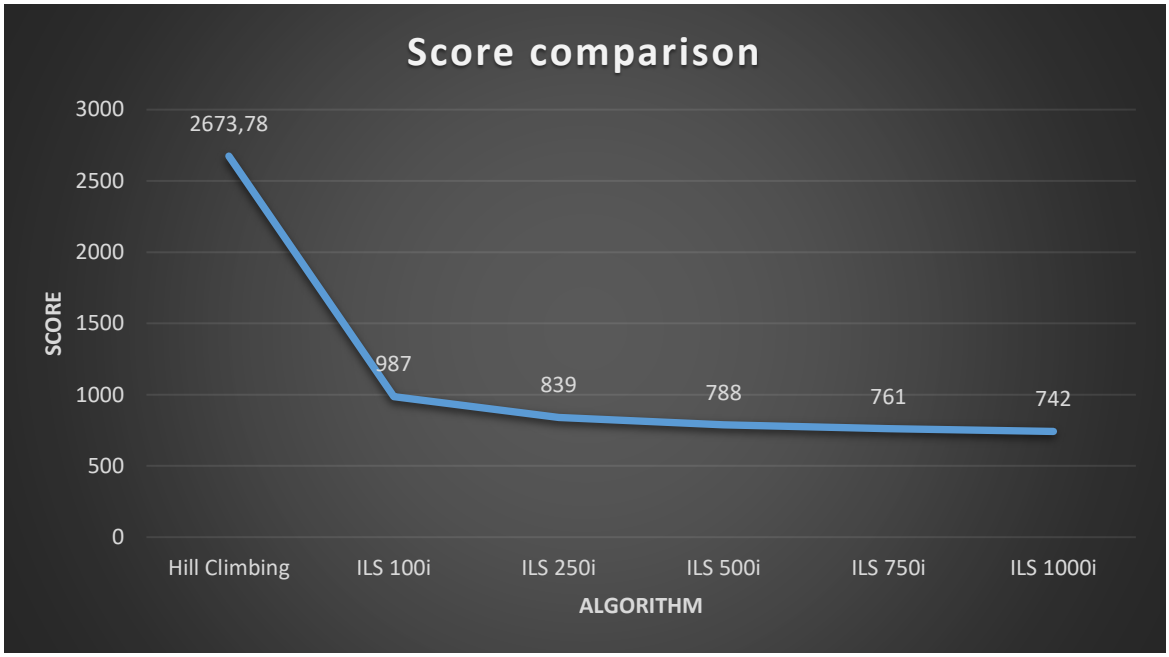


Figure 10: Score comparison for Local Search algorithms

Conclusion: As the figure 10 shows, Iterated Local Search represents an improvement compared to the Hill Climbing algorithm, and depending on the problem, a higher number of iterations can lead to better results. It represents an improvement over the Hill Climbing applied alone.

It is remarkable that after reaching a certain number of iterations, the computational cost keeps increasing due to the high number of evaluations needed, but the solutions don't get much better. So, it is important to choose a correct number of iterations.

6.2 Genetic algorithm:

As this algorithm has lots of schemes, we divided the analysis in parts, using the same problem to isolate the differences that provokes the use of an operator over other. The genetic algorithm uses a selection, crossover, mutation and combination schemes. The thing is to find which of the possible operators returns better performance in the problem:

- **Selection schemes:**
  We have implemented 2 schemes:
  - Tournament: Given a population it will compare the scores of the configurations and add the best one, the one with the lowest score.
  - Rank: The probability of an individual for being selected depends on its position in the ordering based in fitness.

For the following problem this is the score data obtained:

Problem: tsp.TSP 100 10 1 -- GeneticAlgorithm 100 10 100 8 rank replacement

Score using Rank	Score using Tournament
318,9	299,8
307,44	279,38
297,75	318,9
297,75	288,46
288,46	279,38
297,75	297,75
279,38	297,75
299,8	297,75
318,9	299,8
297,75	297,75

Table 1: Selection schemes

Conclusion: The difference between these 2 schemes is very small for this problem, so we cannot infer anything new.

- **Replacement scheme:**
  - Replacement: The new population will simply replace the older one.
  - Elitism: It will take the best configuration in the first population and replace for the worst one in the new population.
  - Truncation: It will truncate with the best configurations of both populations into the new one.

Conclusion: for the different replacement schemes, we have not found any difference in the results.

- **Mutation:**

Varying the mutation percentage can lead the problem to different solutions, if the mutation factor is low, less changes will appear, but if it is very high, it could lead to bad solutions.

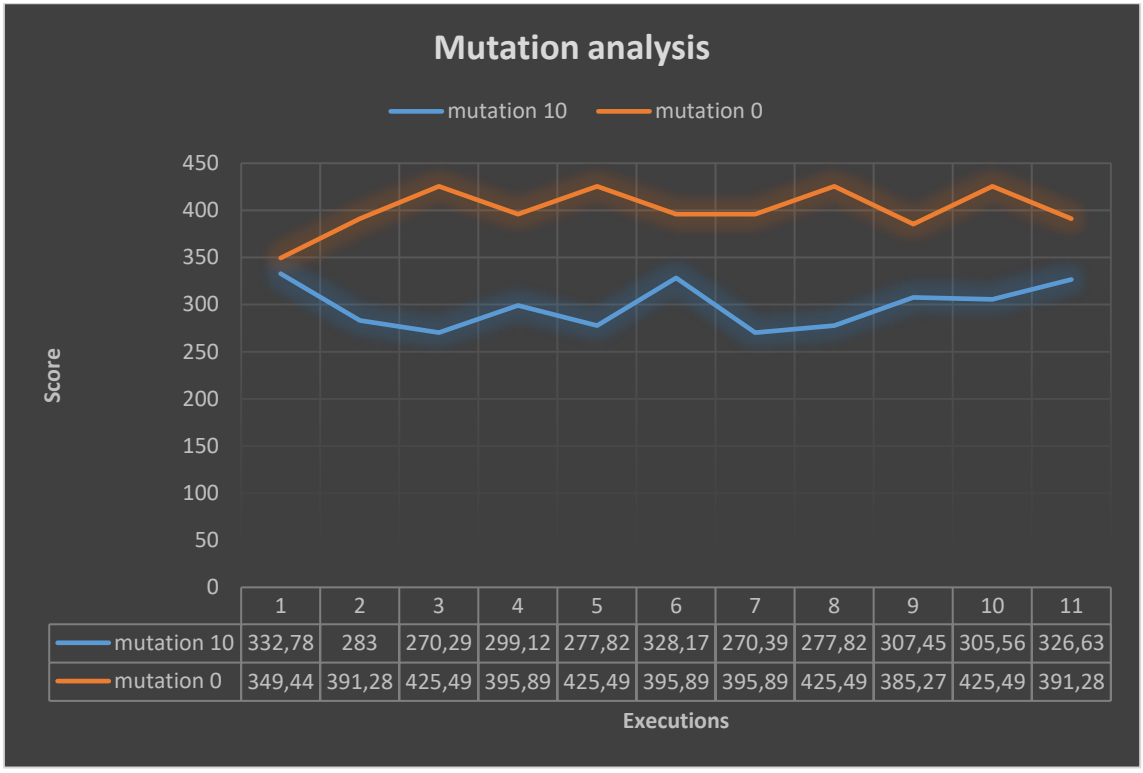
For this study, 8 cities are chosen, as the mutation swaps 2 cities, it has more importance in problems with few cities.

Problem configurations used:

tsp.TSP 100 8 1 -- GeneticAlgorithm 4 100 100 10 rank elitism

tsp.TSP 100 8 1 -- GeneticAlgorithm 4 100 100 0 rank elitism

Applying the different mutation probabilities, the data returned is the following:



	1	2	3	4	5	6	7	8	9	10	11
mutation 10	332,78	283	270,29	299,12	277,82	328,17	270,39	277,82	307,45	305,56	326,63
mutation 0	349,44	391,28	425,49	395,89	425,49	395,89	395,89	425,49	385,27	425,49	391,28

Figure 11: Mutation comparison

As we can see in the Figure 11, using a genetic algorithm with no mutation factor provides worse solutions.

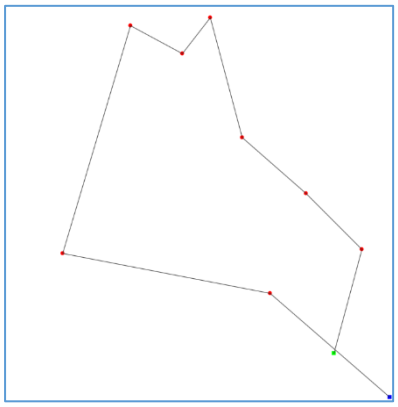


Figure 13:Mutation 10%

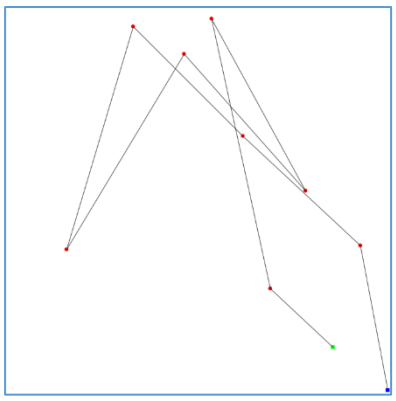


Figure 12: Mutation 0%

Conclusion: Mutation allows to find new solutions that could be better or the problem. Although the other operators had not shown any difference for the problem, the mutation probability difference in performance is visible.

- **Comparison between population size and iterations:** Another interesting analysis is to see if it is better to use a higher number of generations or a higher number of population, or if there is a middle point between them in terms of efficiency.

  For this comparison this problem configuration was used:

tsp.TSP 100 10 1 -- GeneticAlgorithm 1000 500 100 10 tournament replacement.

Where the variables are size and nº generations.

The data we obtained is represented in Table 2 and 3.

Size\Nº Gen	100	200	300	400	500	600	700	800	900
Size 20	300	294	294	291	291	288	285	285	285
Size 50	292	293	293	294	291	288	285	285	285
Size 200	291	291	290	289	289	285	285	285	281
Size 1000	279	279	279	279	279	279	279	279	279

Table 2: Score for Generations size and number of generations.

Size\ Nº Gen	100	200	300	400	500	600	700	800	900
Size 20	1020	4020	6020	8020	10020	12020	14020	16020	18020
Size 50	5050	10050	15050	20050	25050	30050	35050	40050	45050
Size 200	20200	40200	60200	80200	100200	120200	140200	160200	180200
Size 1000	101000	201000	301000	401000	501000	601000	701000	801000	901000

Table 3: Number of evaluations for Generations size and number of generations.

If we take a look at Table 2 and Table 3, we can clearly see that as what determines the quality of solutions is actually the number of evaluations, we can see that as the evaluations grow the score obtained is better, and the number of evaluations is directly proportional to the population size and the number of generations.

For some sizes having worse score when increasing generations like in Size 50 with 300 generations and 400 generations is just pure randomness because of the mutation chance and is not something to worry about since in the Figure 11 we can see that the tendency is to have better solutions as we increase the number of generations.

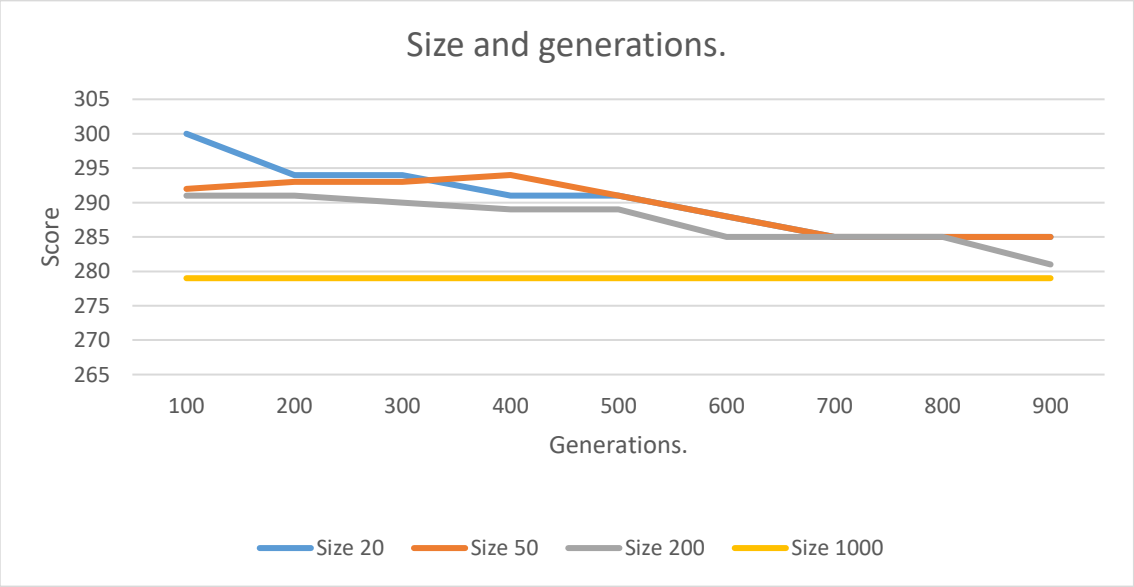


Figure 14: Size and generations comparison.

As we can see in Figure 14 for all the sizes, we can see that as the number of generations increase the algorithm tends to find better solutions. The same happens with the size,

for the same amount of generations, as we have a bigger generation size the algorithm finds better solutions, the lower the score the better the solution is.

Conclusion: The correlation that we can find here is that, the smaller the populations are, in order to obtain the most efficient solutions, the more generations we will need. In other words, the bigger our population is the lesser generations we will need.

This is due to the fact that what matters is the number of evaluations which is calculated with *(PopulationSize*number of generations)+PopulationSize*. We could find the best solution easily by having a lot of generations and a big population size, which will mean very having a big number of evaluations which translates into very big evaluation times, which is not efficient.

6.3 Comparison between different algorithms:

Another interesting study is to put the three algorithms working over the same data to see which of them returns a better solution, better performance and use of resources. The following tables show the values returned when applying the 3 algorithms over the same problem:

Algorithm	Score	Evaluations	Time(s)
Hill Climbing	2815.08	2451	0
ILS[1]	725,53	2451000	28
Genetic[2]	700,21	501000	7

Table 4: tsp.TSP 100 50 1 comparison

Algorithm	Score	Evaluations	Time(s)
Hill Climbing	5478,62	9901	0
ILS[3]	1297.74	9901000	246
Genetic[4]	1174,52	1001000	25

Table 5: tsp.TSP 100 50 1 comparison

Here a plot to visually see the differences:

[1] ILS parameters: 1000.

[2] Genetic algorithm parameters: *1000 500 100 10 rank replacement.*

[3] ILS parameters: 1000.

[4] Genetic algorithm parameters: *1000 1000 100 10 rank replacement.*

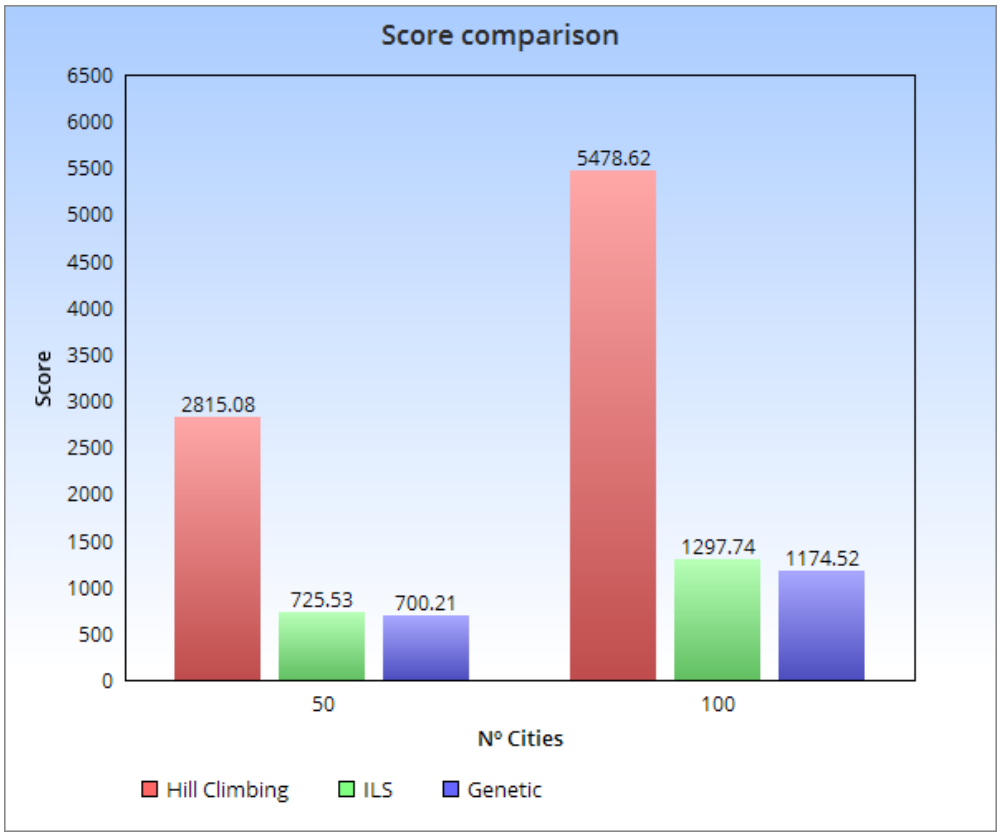


Figure 15: Score comparison

Conclusion: While Hill Climbing needs few evaluations and returns a very bad solution, ILS is reaching quite better solutions, but having a cost in evaluations that grows a lot, as the problem size increases.

On the other hand, genetic algorithm performs better than the other 2, while its computational cost remains lower, depending only on the iterations passed by parameter and the population size. These parameters could be increased to find even better solutions, with a cost just a little bit higher than the needed for this scenario.

7. Final Conclusions:

- When choosing what algorithm to use we must take into account what we really want, if we only want a local improvement and not to find the optimal global solution for a problem it is enough to use Hill Climbing.
- Hill Climbing is not a very useful tool when applied alone, but it can be really powerful if it is applied mixing other algorithms with it, like the ILS, helping to obtain local optima in a very efficient way, or improving solutions obtained by other algorithms.
- Genetic algorithms show us the importance of choosing a good scheme of selection and replacement, and also to take into account the weight that we give to the mutation and crossover operators, as these operators could lead the

algorithm to find whether a good or a bad solution for the problem in more or less time.

- When talking about number of generations and population size in genetic algorithms, it is important to know that having small fixed populations implies that in order to find the most optimal solutions it is necessary to have a big number of generations. Having big populations will make the problem have a bigger diversity, but will require more calculations and so, will be less efficient solving the problem.
- To sum up, different problems need the use of different algorithms. For simple problems, a Local Search algorithm like ILS could be a really good solution, as it would involve few resources and design to develop an algorithm that returns good solutions.
- For a more complex problem, this could not be enough, and then it would be required a genetic algorithm. They are more complex and require the design of more functions, but they can improve the solutions vastly, with a minimum use of resources. And it is an elegant way of solving these kinds of problems without checking millions of possible solutions.