Solving the Symmetric Travelling Salesman Problem Using Simulated Annealing And Genetic Algorithms

Fiodorov Cristian and Neagu Alexandru

December 2022

Abstract

In this paper, we compare the performance and the quality of approximation given by two heuristic techniques: Genetic Algorithms and Simulated Annealing. We designed two algorithms (one for each technique) and ran them on 10 well-known instances of the **Symmetric Travelling Salesman Problem**. We show the experimental results obtained with each instance and conclude that the Genetic Algorithm performs faster and generates better results for every instance.

1 Introduction

The Travelling Salesman Problem (abbreviated TSP) is a well-known **NP-hard** problem in combinatorial optimization, with numerous uses in theoretical computer science and operations research. The problem might be simply described as follows "Given a list of cities and the distances between each pair of cities, find the shortest possible path that visits each city exactly **once** and returns to the initial **city**". [3]

The TSP problem might be of the following types:

> **Symmetric**. The distance between two cities is exactly the same, no matter the direction (i.e the distance from city **A** to city **B** is equal to the distance from city **B** to city **A**).
>
> **Asymmetric**. Path might not exist in both directions, or the distance between two cities is different, depending on the direction.

The purpose of this paper is to draw a comparison between the Simulated Annealing Heuristic and a Genetic Algorithm. We will run each algorithm on **Symmetric TSP** instances and decide which is the best option to use.

This paper is organized as follows:

In **Section 2** we describe how the **Symmetric TSP** is visualized and what instances are used in this paper. Also, we give a brief motivation on why to use these particular non-deterministic algorithms to solve this problem.

In **Section 3** we will focus on the following topics:

- How to represent a solution for this problem using permutations of nodes?
- What is the fitness function used by both algorithms and how is it computed?
- How to design a Simulated Annealing heuristic hybridized with an Iterative Hill Climbing algorithm to solve this problem?
- How to design a Genetic Algorithm that uses proper mutation operators (PMX crossover, ERX crossover, IV mutation) to solve this problem?

In **Section 4** we show the results generated by both algorithms for different instances of the Symmetric TSP. Each algorithm had been run 20 times. We will focus on how close the average fitness value computed by the algorithm is to the actual well-known solution.

In **Section 5** we discuss the results and draw some conclusions.

2 Further Problem Description

2.1 Motivation

The TSP problem is an NP-hard problem, meaning that a deterministic algorithm that might solve this problem in polynomial time complexity is not known. A well known deterministic algorithm that gives an exact solution to TSP has O($E * 2^V$) time complexity, where E is the numbers of roads between each pair of cities and V is the number of cities. Thus, for instances that have more than 30 cities, any deterministic approach is totally unfeasible.

The only way of computing good approximations for TSP is to use non-deterministic heuristics such as Simulated Annealing or Genetic Algorithms. These approaches can search a large solution space efficiently and generate fairly good approximations.

2.2 Problem Visualisation

There are several ways to represent instances of Symmetric TSP:

- A cartesian plane containing n points having coordinates $(x_i, y_i), \forall i \in [1, n]$. The **distance** between every pair of points is computed using the Euclidean Distance Formula. Thus, this particular instance of Symmetric TSP might be viewed as an undirected graph with n nodes and $\frac{n \cdot (n-1)}{2}$ edges. The cost of each edge is computed using the Euclidean Distance between two points.
- An undirected graph with n nodes and m **weighted** edges that connects certain pairs of nodes.

In this paper, we will specifically use instances that contains n points numbered from 1 to n, with specific cartesian coordinates. The algorithms should find a circular sequence of distinct points (except for the first and last ones) that minimizes the total distance (i.e the sum of Euclidean Distances between each pair of consecutive nodes in the sequence).

2.3 Instances Of The Problem

The best known solutions for the following instances are computed using values rounded to the nearest integer when computing Euclidean Distances.

Easy

Three instances with **51**, **70** and **101** number of cities. These are the smallest instances and are described below:

- **eil51** - The smallest instance with cities close to each other. The best known solution for this instance is **426**.

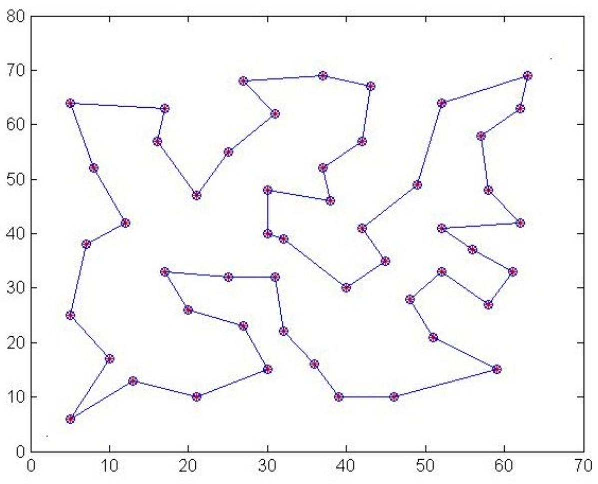


Figure 1: eil51 solution

- **st70** - A slightly bigger instance with moderately low distances between points. The best known solution for this instance is **675**.

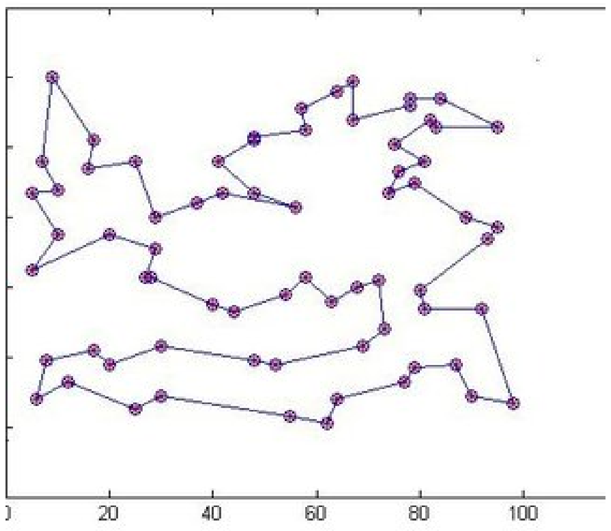


Figure 2: st70 approximate solution

- **eil101** - The biggest instance between the easy ones. The best known solution for this instance is **629**.

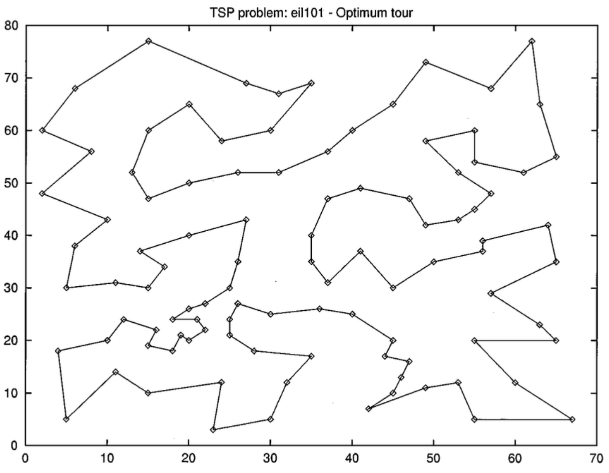


Figure 3: eil101 approximate solution

Medium

Four instances with **52**, **100**, **130** and **225** points. The Euclidean Distances between points are large and it is a complicated task to generate fairly good approximations. These instances are described below:

- **berlin52** - Even though the number of points is small, the distances are quite large. The actual well-known solution for this instance is **7542**.

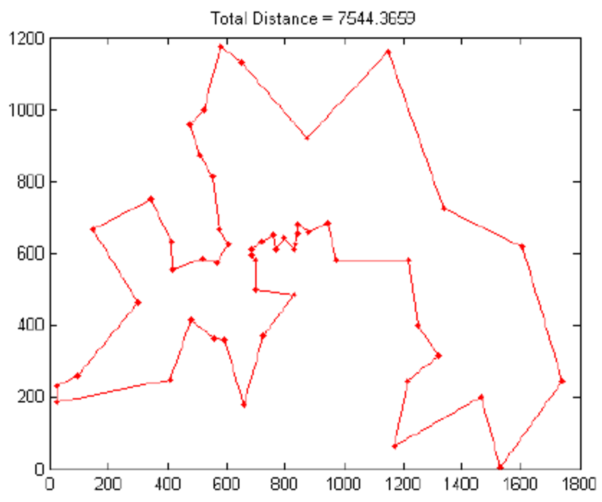


Figure 4: berlin52 solution

- **kroB100** - A relatively small number of cities with much larger Euclidean Distances between points. The actual well-known solution for this instance is **22141**.
- **ch130** - A slightly bigger instance with the points closer to each other. The best known solution for this instance is **6110**.
- **ts225** - The biggest instance between the medium ones. The best known solution for this instance is **126643**.

Hard

Three instances with **417**, **1000** and **13509** points. These are the biggest instances analyzed and the results generated for them are way worse than the optimal solution. These instances are further described below:

- **fl417** - A slightly bigger instance with the points. Even though the distance between cities is relatively small, the big number of cities makes it hard to generate good aproximations for this instance. The best known solution for this instance is **11861**.
- **dsj1000** - A big instance where the points are enormously far away from each other. The best known solution for this instance is **18659688**.
- **usa13509** - This is the biggest instance solved. The best known solution for this instance is **19982859**.

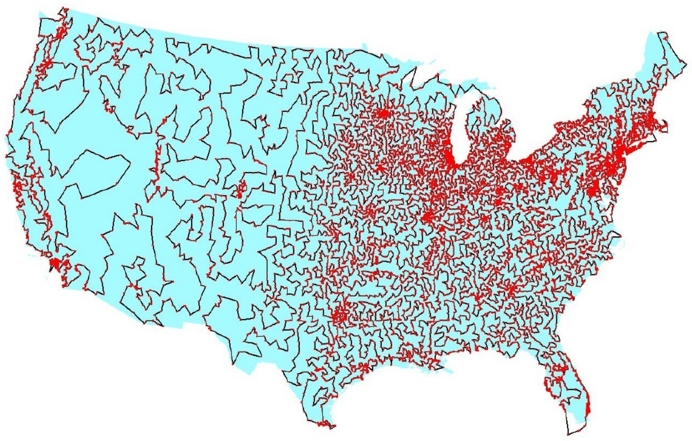


Figure 5: usa13509 approximate solution

3 Algorithms

3.1 Solution Representation

There are several well-known representations of a TSP solution including Binary Representation, Ordinal Representation, Path Representation and Matrix Representation.

In this paper, **Path Representation** is preferred. It is by far the most intuitive and common solution representation when it comes to the TSP problem. The main advantage of this representation is the big amount of crossover and mutation operators that has been developed over time.

Let n be the number of cities from the TSP instance. Each city has a unique identifier from 1 to n. We represent an individual $P = [p_1, p_2, \cdots, p_n] : p_i \neq p_j, \forall i \neq j$ as a path that starts and ends at city p_1 and visits every point in the following way

$$\textbf{PATH} = (p_1, p_2), (p_2, p_3), \cdots, (p_{n-1}, p_n), (p_n, p_1)$$

In the notation above (p_i, p_j) is an unoriented road from city p_i to city p_j. Note that each city p_i is represented by two double numbers $p_i.x$ and $p_i.y$ that corresponds to the cartesian plane coordinates of the city.

3.2 Fitness Function

Having an individual $P = [p_1, p_2, \cdots, p_n]$ that is a candidate solution for the problem, we need to calculate the total distance of the path encoded by P. In our algorithm, the total distance of the path is computed using the following formula:

$$D(P) = \sum_{i=1}^{n} \left\lfloor \sqrt{(p_i.x - p_{i+1}.x)^2 + (p_i.y - p_{i+1}.y)^2} \right\rceil$$

In the formula above, the $n+1$ value corresponds to 1 (in order to compute the total distance in a cyclic manner). The goal is to compute the global minimum value of $D(P)$ for all paths P.

A Genetic Algorithm is usually defined as a maximization algorithm so, for each individual P from a population we compute the fitness value

$$F(P) = \frac{MAX - D(P)}{MAX - MIN + C} + 0.01, \ 0 \leq i \leq N - 1$$

In the formula above, MAX and MIN are the maximum and minimum $D(P)$ values from the population. C is a small constant to avoid division by zero in case the maximum and the minimum values are the same. We may observe that the values $D(P)$ and $F(P)$ are inversely proportional. The minimum value of $D(P)$ corresponds to the maximum value $F(P)$.

3.3 Heuristic Method

In order to test the quality of the Genetic Algorithm, we compare it to another heuristic method. We have designed an algorithm that uses Simulated Annealing heuristic hybridized with First Improvement neighbourhood search.

The set of parameters which comprise the cooling schedule dictate the rate at which simulated annealing reaches its final solution. Converging too quickly can result in sub-optimal solutions while taking too long to determine a solution can result in an unnecessarily large computational effort that would be impractical in a real-world setting.

Unlike a simple Hill Climbing approach, Simulated Annealing has the chance to choose a worst neighbour that maybe will lead to a better solution. This chance depends on the temperature, and it gets smaller and smaller as the temperature decreases, which means that the method explores a large space at the start, but as time passes it narrows down it's search.

A simple overview of the algorithm is shown below

1. At first, we set the starting temperature T and define a $T_0 = 501.982$ (which is just a random number we choosed because it performed well on some tests) that will help to calculate the next values of T.

2. We generate a random individual P.

3. We iterate a fixed number of times and choose a random neighbour of N to evaluate. We describe further what a neighbour is in the **Mutation** subsection 3.4.

4. If $D(N) < D(P)$, we set $P = N$, otherwise we select the worse individual N with probability $p = e^{\frac{-|D(N)-D(P)|}{T}}$.

5. After we finish the iterations from step 3, we lower the temperature using the formula : $T = T_0 \cdot 0.98k$, where k is the number of iteration the outer loop has made. We stop the whole outter loop when the temperature reaches condition $T \leq 10^{-7}$

3.4 Genetic Algorithm

Genetic Algorithms are faster and more efficient compared to other methods. They are suited for parallel implementation, optimizing both continuous and discrete functions and delivering good results fast.

The main advantage of a Genetic Algorithm is the fact that the search is not biased toward a locally optimal solution. A genetic algorithm uses the mutation operator to avoid the convergence to suboptimal solutions. The crossover operator is used in order to explore a multitude of regions at a fast enough pace. These features make the Genetic Algorithm to be an obvious choice over most gradient descent algorithms.

The Genetic Algorithm designed in this paper is a classic one. It iterates through 2000 generations with a population of 200 individuals. In each generation a specific Selection Method, Mutation Operator and Crossover Operator is applied on the population. In the end, the fittest individual from the final population (the individual P with the highest $F(P)$) is considered to be the optimal solution of the problem.

Selection

For TSP, we decided to implement the **wheel of fortune** selection mechanism. This mechanism works as follows:

- For a population Pop we assign a selection probability $prob(P)$ to each individual, directly proportional to its fitness value $F(P)$. According to these probabilities we select 200 individuals for the next generation. Note that there is a chance for the same individual to be selected multiple times.

To sum up, this selection mechanism perfectly encapsulates the "survival of the fittest" evolutionary principle.

Mutation

We have implemented two different variations of the mutation operator. Each of them works well for particular instances of the problem, depending on the number and the location of the cities.

The first variation of mutation operator it is applied on each individual $P = [p_1, p_2, \cdots, p_n]$ from the current population in the following way:

- Firstly, the individual is encoded into a permutation $G = [g_1, g_2, \cdots, g_n]$. This encoding is based on the identity permutation $I = [1, 2, \cdots, n]$.
  Let **indexOf**(p_i) be the corresponding index of p_i in array I. The encoding process is done in the following way:

  1. For each i from 1 to n repeat step 2.
  2. $g_i =$ **indexOf**(p_i) and $I = I \setminus \{p_i\}$

- Let $\mathbf{rand}(1, n)$ be a random number from interval $[1, n]$. Iterate on the encoded permutation G and for each element g_i apply the following operation with some probability of mutation:
  - $*$ $g_i = \mathbf{rand}(1, n)$
- Decode the obtained permutation G by repeating the same process described by encoding, but this time on G.

The steps described above are also useful for the Simulated Annealing heuristic. We define a neighbour N as the encoded permutation G where the $*$ operation is applied **exactly** once.

Inversion Mutation

The above mentioned variation of mutation doesn't work very well for big instances of the problem. We implemented another variation called inversion mutation (IVM). This operator doesn't iterate through each gene anymore. However, it does the following operation on the entire population one single time according to a specific probability of mutation:

- Extracts a **random** subarray $[p_l, p_{l+1}, \cdots, p_r]$ from individual P.
- Insert the **reversed** subarray $[p_r, p_{r-1}, \cdots, p_l]$ into any position of the remaining array $P \setminus \{p_l, p_{l+1}, \cdots, p_r\}$.

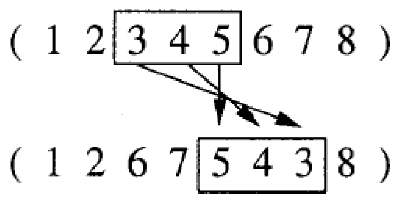


Figure 6: IVM example

Adaptive Parameters

Using the same mutation probability might disadvantage the performance of the algorithm. Sometimes, a larger mutation probability for better and faster exploration of different regions is needed. However, there are situations when exploration should be neglected and further exploitation of the current region should be promoted. In these cases, lower mutation probabilities are needed.

Thus, we make the mutation probability **adaptive** to obtain a better balance between **exploration** and **exploitation**.

We compute a different mutation probability for each individual from the current population in the following manner:

Let F_{max} be the maximum fitness value from the current population

Let $\overline{F}$ be the average fitness value based on each individual from the current population

For each individual P_i we compute its mutation probability pm_i using the following formula

$$pm_i = \begin{cases} k_1 \cdot \dfrac{F_{max} - F_i}{F_{max} - \overline{F}}, & F_i \geq \overline{F} \\ \\ k_2, & F_i < \overline{F} \end{cases} \tag{1}$$

The constant k_2 is a big mutation probability we set for individuals with below-average fitness values. The constant k_1 directly controls how big our adaptive mutation probability might be.

We may observe that in case of $F_i = F_{max}$ the mutation probability pm_i will be equal to 0. In case this happens, we should assign pm_i to a small enough probability (we used 0.008).

Crossover

We have chosen two well-known variations of the crossover operator. We are alternating between the two depending on the problem instance. Let P_1 and P_2 be the individuals selected for crossover.

Partially Mapped Crossover

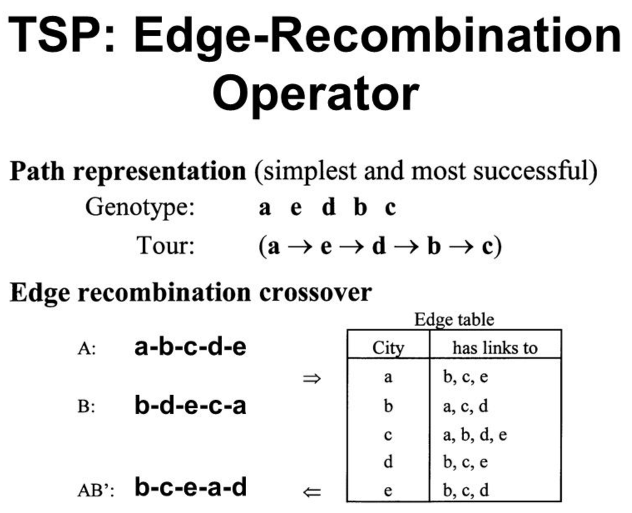
This variation worked really well for small instances. It creates a single descendant C based on P_1 and P_2. The descendant is generated in the following manner:

- Select a subarray $[l, r]$ from the first parent. Copy the corresponding genes from P_1 directly into the descendant C.
- Iterate through the segment $[l, r]$ of the second parent P_2 and select each value p_i that hasn't already been copied to the descendant. Let's define **indexP2**(c_i) as the corresponding position of the value c_i in the array P_2. For each selected value p_i do the following:
  1. Initialize $j =$ **indexP2**(c_i).
  2. As long as $j \in [l, r]$ repeat step 3, otherwise jump to step 4.
  3. $j =$ **indexP2**(c_j)
  4. Assign $c_j = p_i$
- After all the values from the step above has been processed, check if there are some position i where c_i is not assigned yet. For those positions, assign $c_i = p_i$.

Edge Recombination Crossover

This crossover variation works well for symmetrical TSP. This operator tries to preserve the edges of the parents as much as possible. The method works as follows:

- Let $P_1 = [x_1, x_2, \cdots, x_n]$ and $P_2 = [y_1, y_2, \cdots, y_n]$ be the parents selected for crossover. According to the solution representation discussed in section 3.1 both parents can be represented as **cycles**. We analyze each value $x_i = y_j$. The adjacent nodes of x_i in P_1 are x_{i-1} and x_{i+1} (note that $i - 1$ and $i + 1$ are considered cyclically. Adjacent nodes of y_j in P_2 are defined in the same way. Thus, we define **adjList**(x_i) as a list of all the distinct neighbours of x_i considering **both** parents. To sum up, for each value $x_i (= y_j)$ we update the adjacency list as follows:
  - Insert $x_{i-1}, x_{i+1}, y_{j-1}, y_{j+1}$ into **adjList**(x_i). If two values are the same, only one of them is be inserted. Furthermore, each node can have minimum 2 and maximum 4 neighbours.
- On the above described graph, this algorithm should be performed:
  1. Choose an **unvisited** city $p \in [1, n]$ at random. Remove node p from all the adjacency lists that contain it. If all the nodes are visited, the algorithm stops here.
  2. If there are still some nodes remaining in **adjList**(p), select a node v from it with the **fewest** entries in its own adjacency list. Append node p to the descendant. Assign $p = v$ and repeat step 2.
  3. If the adjacency list of node p is empty, append p to the descendant and go to step 1.
- After these steps are performed a single descendant C is obtained.



TSP: Edge-Recombination Operator

Path representation (simplest and most successful)

Genotype: **a e d b c**

Tour: **(a → e → d → b → c)**

Edge recombination crossover

A: **a-b-c-d-e**

B: **b-d-e-c-a**

AB': **b-c-e-a-d**

Edge table

City	has links to
a	b, c, e
b	a, c, d
c	a, b, d, e
d	b, c, e
e	b, c, d

Figure 7: ER example

4 Results

The proposed optimization algorithm, as well as the Simulated Annealing algorithm, are tested on a set of 10 instances (described above), the difficulty of which varies from low to high. For each instance, 20 runs were performed to make an average value. While running the Genetic Algorithm, we iterated through a set of parameters in order to get the best results and listed them below.

Instance Name	Best value	Average value	SD value	Actual value	Average time
eil51	427	444	11.33	426	3159.7
st70	686	727.1	22.75	675	4404.5
eil101	708	734.7	16.97	629	6198.15

Table 1: Genetic Algorithm - Low difficulty instances

Instance Name	Best value	Average value	SD value	Actual value	Average time
eil51	589	662	39.91	426	67083.35
st70	1242	1411.55	64.3	675	93235.35
eil101	1168	1277.25	46.11	629	132919.55

Table 2: Simulated Annealing - Low difficulty instances

Instance Name	Best value	Average value	SD value	Actual value	Average time
berlin52	7833	8198	200.17	7542	6362.15
kroB100	25148.0	27058.55	1333.44	22141	6661.3
ch130	7952.0	8362.85	243.58	6110	8707.5
ts225	261547	289208.35	12846.74	126643	39780.5

Table 3: Genetic Algorithm - Medium difficulty instances

Instance Name	Best value	Average value	SD value	Actual value	Average time
berlin52	10971	11718.8	396	7542	75032.15
kroB100	56789	63995.45	3161.04	22141	144449.4
ch130	18323	19560.85	638.87	6110	190566.55
ts225	638801	725202.7	48249.95	126643	304759.1

Table 4: Simulated Annealing - Medium difficulty instances

Instance Name	Best value	Average value	SD value	Actual value	Average time
fl417	39574	43917.95	2353.41	11861	102531.4
dsj1000	200504484	206290551.7	3276938.48	18659688	81356.15
usa13509	1702641335	1716429130	7324779.39	19982859	796139

Table 5: Genetic Algorithm - Hard difficulty instances

Instance Name	Best value	Average value	SD value	Actual value	Average time
fl417	117249	134578.7	7642.24	11861	607227.9
dsj1000	272872374	278119658.5	3524165.81	18659688	1621800.6
usa13509	1945341813	1968541294	9554314.57	19982859	372453.35

Table 6: Simulated Annealing - Hard difficulty instances

5 Conclusions

After analyzing the results generated by the Genetic Algorithm and by the Simulated Annealing Heuristic, the following conclusions may be drawn:

- The Genetic Algorithm is able to obtain fairly good results, that are close to optimum, for low and medium difficulty instances. The Simulated Annealing heuristic obtained 2 to 4 times worse results for these instances.
- For **Hard** difficulty instances, both algorithms obtained results that are not close to the optimum solution, but considering the difficulty of those instances the results are acceptable. Also, even for this level of difficulty, The Genetic Algorithm was superior to the Simulated Annealing heuristic.
- For big instances, the Simulated Annealing heuristic is able to obtain results that are close to those obtained by a Genetic Algorithm.
- The Genetic Algorithm was able to explore the space of a very hard instance (usa13509) way faster than Simulated Annealing. Even though the approximation is not great for this particular instance, it is still a fairly good point to start. Usually, these kind of instances requires particular heuristics based on the location of the points.
- There are a lot of possibilities to implement a Genetic Algorithm. We could pick other selection methods, crossover probabilities, and genetic operators to generate better solutions for particular TSP instances. Testing every combination of such operators and parameters is a hard and long process

References

[1] P. Larranaga, C.M.H. Kuipjers, R.H. Murga, I. Inza and S. Dizdarevic. *Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators.*

[2] Chunhua Fua, Lijun Zhangb, Xiaojing Wangc, Liying Qiaod. *Solving TSP Problem with Improved Genetic Algorithm.*

[3] Wikipedia. *Travelling Salesman Problem.* `https://en.wikipedia.org/wiki/Travelling_salesman_problem`

[4] Eugen Croitoru. *Teaching: Genetic Algorithms.* `https://profs.info.uaic.ro/~eugennc/teaching/ga/`

[5] Ruprecht-Karls-Universität Heidelberg. *Problem Instances For Symmetric TSP.* `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/`

[6] Ruprecht-Karls-Universität Heidelberg. *Optimal solutions for symmetric TSPs.* `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html`

[7] Rubicite. *PMX Crossover Description.* `https://www.rubicite.com/Tutorials/GeneticAlgorithms/CrossoverOperators/PMXCrossoverOperator.aspx/`

[8] *ER crossover image.* `https://slideplayer.com/slide/8783766/`

[9] *Solution Image for eil51* `https://www.researchgate.net/figure/The-Best-Solution-Achieved-by-ABCSA-for-Eil51_fig2_271132498`

[10] *Solution Image for st70* `https://www.researchgate.net/figure/The-Best-Solution-Achieved-by-ABCSA-for-St70_fig4_271132498`

[11] *Solution Image for eil101* `https://www.researchgate.net/figure/eil101-TSP-problem-An-optimum-state_fig4_227125718`

[12] *Solution Image for berlin52* `https://www.researchgate.net/figure/The-optimal-solution-of-Berlin52_fig2_221901574`