LOP Heuristic Search

Alex Amenabar
Alan García
alex.amenabar@ehu.eus
agarcia901@ikasle.ehu.eus

Abstract

The Linear Ordering Problem (LOP) is a well-known combinatorial optimization problem. When the permutation size is large, finding the optimal solution becomes infeasible. Therefore, various heuristic algorithms are used to obtain solutions that, while not optimal, are sufficiently good and can be found in a relatively short time. In this work, two types of algorithms have been implemented and tested: a local search-based algorithm, Simulated Annealing, which iteratively explores better solutions among neighboring candidates, and a genetic algorithm that generates populations of solutions. The results indicate that the implemented algorithms achieve good performance in a short time if the correct parameters values are selected.

Keywords

Optimization LOP Heuristics Simulate Annealing Genetic Algorithm

ACM Reference Format:
Alex Amenabar and Alan García. 2025. LOP Heuristic Search. In . ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The Linear Ordering Problem (LOP) [3] is a well-known combinatorial optimization problem. It is classified as NP-hard [11], meaning that when the permutation size is large, finding the optimal solution becomes infeasible. Consequently, heuristic algorithms [4] are often employed to obtain good solutions. These algorithms use various strategies to iteratively improve solutions and approximate the optimum, although they do not guarantee finding the global optimum solution. Nevertheless, heuristic algorithms can efficiently produce high-quality solutions within a short time.

There are different types of heuristic algorithms, such as local search algorithms [6] and genetic algorithms [8]. In the case of local search algorithms, the process begins with an initial solution, and in each iteration, a neighborhood is generated. One individual from this neighborhood is then selected as the new solution. The neighborhood consists of solutions obtained by applying a specific operation to the current solution. The operation selected for generating the neighborhood and the criteria for selecting the new solution are crucial in local search algorithms, as they directly impact both the algorithm's execution time and its progression.

In the case of genetic algorithms, an initial population of solutions is generated, and by means of crossover [7] and mutation [9] operators, the population gradually evolves obtaining better solutions. Genetic algorithms are inspired by the evolutionary processes observed in biology, where crossover operators combine the characteristics of existing individuals (the parents) to create new better individuals, while mutations occur with a certain probability to introduce diversity into the population.

In this work, two algorithms have been implemented in C/C++, a local search algorithm, simulated annealing [1], and a genetic algorithm. This implementations is available in the following repository[1]. C and C++ have been used to reduce the execution times of genetic algorithms.

2 Problem description and formalization

In the Linear Ordering Problem (LOP), preferences are established between pairs of elements, and the objective is to order them in the best possible way by maximizing the total preference value.

Therefore, the **search space** of the LOP consists of all possible ordered sequences of the given elements.

There are different ways to model this problem, one of which is using a permutation. A solution x is represented as an ordered sequence of elements, so the **encoding space** can be described as follows:

$$x = \{x_0, x_1, \ldots, x_{n-1}\}$$

where n is the number of elements in the LOP. Using this encoding, the total number of possible solutions is $n!$.

Preferences between pairs of elements are used order all the elements. This can be modeled by a matrix $M = [m_{i,j}]_{i,j=0}^{n-1}$, where n is the number of elements in the permutation, and $m_{i,j}$ represents the preference of element i over element j. Using this matrix, the objective function is defined as the sum of the values that are upper the diagonal of the matrix, as it contains the preference values for the permutation considered as the solution. The objective function is given in Equation 1:

$$f(x) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} m_{x_i,x_j}, \tag{1}$$

where m_{x_i,x_j} represents the preference of element x_i over x_j.

3 Simulated Annealing Algorithm

Simulated Annealing (SA) is an heuristc algorithm used for optimization problems. It is inspired by the annealing process in metallurgy, where a material is heated and then slowly cooled to achieve a stable structure. This algorithm follows a similar principle: it begins with an initial solution and iteratively explores neighboring solutions, accepting worse solutions with a certain probability to escape local optima. This probability is controlled by a temperature parameter, which decays over time according to a given cooling function. This cooling strategy prioritizes exploration of the search space when the temperature is high, as worse solutions are more

[1]https://github.com/alanglk/KISA_LOP

Conference'17, Washington, DC, USA
2025. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

likely to be accepted, while exploitation of promising solutions occurs as the temperature decreases.

SA is classified as a local search algorithm because it explores the solution space by making incremental modifications to an initial solution.

3.1 Pseudocode

The SA algorithm implemented Here, the pseudocode of the implemented SA algorithm:

Algorithm 1 Simulated Annealing (SA)

```
1: Input: Initial solution $x_0$, Initial temperature $T_0$, Neighborhood function $N(x)$,
   Cooling function $T(T_k)$, max_chain_size
2: Output: Approximate optimal solution $x \in S$
3: Initialize $x \leftarrow x_0$                          ▷ Current solution
4: Initialize $k \leftarrow 0$                            ▷ Iteration count
5: Initialize $st \leftarrow 0$                           ▷ Stagnation count
6: while not stop conditions* do
7:     chain_size ← 0
8:     while chain_size < max_chain_size do
9:         $x_{new} \leftarrow N(x)$                      ▷ Get a neighboring solution
10:        $\Delta f \leftarrow f(x) - f(x_{new})$
11:        if $\Delta f < 0$ then
12:        else
13:            $I \leftarrow$ uniform random from$[0, 1]$
14:            if $e^{\frac{\Delta f}{T_k}} > I$ then
15:                $x \leftarrow x_{new}$                 ▷ Accept the new solution
16:            end if
17:        end if
18:        chain_size ← chain_size + 1
19:    end while
20:    if $f(x) = f(x_{prev})$ then
21:        $st \leftarrow st + 1$                        ▷ Increment stagnation
22:    else
23:        $st \leftarrow 0$
24:    end if
25:    $T \leftarrow T(k)$                                ▷ Apply the cooling function
26:    $k \leftarrow k + 1$
27: end while
28: return $x$
```

Where the stop conditions∗ are: (1) the number of iterations $k \geq$ max_iterations, (2) the temperature $T_k \leq 0$, and (3) the stagnation count $st \geq$ max_stagnation. These conditions ensure the algorithm controls the number of evaluations and terminates when no better solution is found after multiple iterations.

Note that max_chain_size controls the search for neighbors of the current solution. Some works, such as [12], dynamically adjust this parameter based on the problem, but in this implementation, it remains fixed across iterations for simplicity.

3.2 Neighborhood functions

In this implementation, the swap function has been selected as the Neighboring function. The swap function takes two elements of the permutation and changes their positions, and the weight matrix is also updated accordingly.

The neighborhood size for a given solution is approximated as $N(x) = \frac{n \times (n-1)}{2}$.

3.3 Selection of initial temperature

As the temperature controls the probability of accepting worse solutions, it is crucial to set an initial temperature T_0 that allows for sufficient exploration of the search space while maintaining a balance between randomness and convergence towards an optimal solution.

Typically, the initial temperature is set as $T_0 \leftarrow \frac{f_{\max} - f_{\min}}{\log \alpha}$, where α is the acceptance rate. However, since $f_{\max}$ and $f_{\min}$ are usually unknown, two main empirical approaches are commonly used:

(1) Manually selecting T_0
(2) Performing a random walk from the initial solution to estimate $f_{\max}$ and $f_{\min}$

Below is the pseudocode for the random walk approach:

Algorithm 2 Random Walk

```
1: Input: Initial solution $x_0$, $N\_iterations$, $N\_perturbations$, Neighborhood
   function $N(x)$, $\alpha$
2: Output: $T_0$
3: Initialize $f_{min} \leftarrow +\infty$
4: Initialize $f_{max} \leftarrow -\infty$
5: for $i$ in $N_{iterations}$ do
6:     $x_{temp} \leftarrow x_0$
7:     $num\_p \leftarrow$ uniform random from$[0, N\_perturbations]$
8:     for $j$ in $num\_p$ do
9:         $x_{temp} \leftarrow N(x_{temp})$              ▷ Get a neighboring solution
10:        if $f(x_{temp}) < f_{min}$ then
11:            $f_{min} \leftarrow f(x_{temp})$
12:        end if
13:        if $f(x_{temp}) > f_{max}$ then
14:            $f_{max} \leftarrow f(x_{temp})$
15:        end if
16:    end for
17: end for
18: return $\frac{f_{max} - f_{min}}{\log \alpha}$
```

3.4 Cooling functions

In order to update the temperature of the algorithm, three approaches have been implemented:

(1) Linear: $T_{new} \leftarrow T - \beta$
(2) Geometric: $T_{new} \leftarrow T \times \beta$ where $\beta \in (0, 1)$
(3) Logarithmic: $T_{new} \leftarrow \frac{T_0}{\log k}$ where k is the current iteration

4 Genetic Algorithm

Genetic algorithms iterate over a population of individuals, gradually improving the solutions within the population. These algorithms are inspired by the genetic and molecular principles of biology. To generate new individuals, a selection process is performed in which several parents, usually two, are chosen from the population. Their characteristics are then combined to create new offspring. Additionally, mutations are applied to these individuals with a certain probability to introduce diversity on the population.

In these algorithms, it is crucial not only to improve the solutions within the population but also to maintain diversity. If all individuals become too similar, the algorithm may converge prematurely to a local optimum. Therefore, the genetic operators used must be designed to enhance solutions while also preserving diversity.

4.1 Operators

To generate new individuals, parents must first be selected from the population. There are various methods for this. One approach is elitist selection [2], where the best individuals are chosen. However, this can lead to premature convergence to local optima [5].

Another common method is tournament selection, where a subset of k individuals competes to determine each parent [10], an the best of them is selected to be a parent. If the chosen value of k is appropriate, the tournament will include enough individuals to select high-quality parents while avoiding the consistent selection of the best individuals, thereby promoting diversity.

After selecting the parents, the offspring must be generated using two genetic operators: crossover and mutation. The crossover operator combines the characteristics of the two parents to create offspring. A widely used method for permutations is order crossover, in which a part of the permutation is copied from one of the parents, and the remaining values are filled in order from the other parent. Once the offspring is generated, a mutation operator may be applied to introduce further diversity. Common mutation operators include the swap and insert operations.

Finally, after generating the offspring, the individuals of the new population must be selected. There are several approaches to this. One option is elitist selection, where the best individuals are chosen from both the previous population and the offsprings. Alternatively, the new generation can consist exclusively of the offspring. More sophisticated selection strategies can also be employed to balance exploitation and exploration within the algorithm.

4.2 Implementation details

The Algorithm 3 reflects the structure of the code implemented for the genetic algorithm. The operators used are the tournament selection, the order crossover and swap operators to generate the offsprings, and finally the elitist selection between the original population and the offsprings to select the new generation individuals.

First, an initial population is generated randomly. Then, the previously mentioned operators are used in each iteration. For each pair of offsprings, two parents are selected using the tournament selection. The algorithm continues running until the maximum number of allowed iterations is reached or a stopping criterion is met.

Finally, the value of k depends on the number of individuals in the population. Let n_p be the population size, then, k is computed as $k = \frac{n_p}{3}$. This ensures that k is large enough for the tournament to consistently select good parents, while not always choosing the best ones, thereby preserving diversity for a longer time.

5 Experimental results

This section shows the results obtained with the two algorithms, and the comparison of the two algorithms.

5.1 Simulated Annealing

The balance between exploration and exploitation in the SA algorithm is illustrated in Figure 1. During the initial iterations, the higher temperature increases the likelihood of exploration, causing greater fluctuations in the objective function values. As the temperature decreases in later iterations, exploitation becomes dominant, leading to more stable values.

Multiple executions of the SA algorithm are presented in Table 1. These results indicate that both linear and logarithmic cooling strategies perform poorly. The algorithm converges slowly, finds suboptimal solutions, and requires a large number of iterations. In all cases, the algorithm terminated because the maximum number of iterations was reached.

Algorithm 3 Genetic Algorithm

```
1: Input: Initial solution $x_0$, $n_p$, $n_p$ $p_{cross}$, $p_{mut}$, $k$, $max\_steps$, $stop\_criteria$, $stop\_criteria\_value$
2: Output: $best\_result$
3: population ← generate_population($x_0$)
4: step ← 0
5: stop ← False
6: while step < max_steps or stop = False do
7:     generated_children ← 0
8:     while generated_children < $n_p$ do
9:         parents ← tournament(population, k)
10:        if random_number(0,1) < $p_{cross}$ then
11:            offspring1, offspring2 ← order_crossover(parents)
12:        else
13:            $offspring1 \leftarrow parent1, offspring2 \leftarrow parent2$
14:        end if
15:        if random_number(0,1) < $p_{mut}$ then
16:            swap(offspring1, offspring2)
17:        end if
18:    end while
19:    population ← elitist_selection(population, offsprings)
20:    best_result ← max(best_result, max_result(population))
21: end while
22: return best_result
```

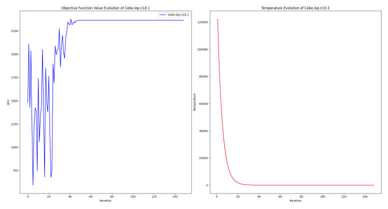

Figure 1: Simulated Annealing: Exploration vs. Exploitation in the Cebe.lop.n10.1 problem with geometric cooling function.

The best results were obtained using the geometric cooling strategy with $\beta = 0.8$. The initial temperatures were set based on 10,000 iterations, a maximum of 100 perturbations per iteration, and $\alpha = 0.75$ for random walk exploration.

However, the implemented random walk is constrained to a local region, as the number of perturbations determines the maximum exploration distance. Additionally, since each iteration resets the initial solution as the starting point, further exploration is restricted. To address this limitation, the temperature was increased in some cases to enable broader exploration.

It is also interesting to point out that despite this implementation doesn't outperform the benchmark objective value it achieves very close results at a reasonable time with the geometric configuration.

5.2 Genetic Algorithm

The results obtained by the GA are presented in Table 2, which summarizes the outcomes for different problems under various configurations. Each row corresponds to a specific problem and configuration, detailing the best-known solution for the problem, the best result found by the implemented algorithm, the population size, the number of participants in the tournament selection (k), the

Table 1: Results of SA with different configurations for different problems.

Prob.	Best	Our	T_0	u	β	ch	st	It.	T(s)
N-r100a2	145270	142964	109322	1	1.1	100	100	100000	2497.02
N-r100a2	**145270**	144178	109322	2	0.8	100	100	452	11.36
N-r100a2	145270	85553	109322	3	0	100	100	100000	2495.94
N-r150a0	360978	338585	248389	1	11.73	100	100	21173	1166.28
N-r150a0	**360978**	360003	234637	2	0.8	100	100	852	47.06
N-r150a0	360978	200854	1173180	3	0	100	100	100000	5507.92
N-r200a0	654604	632267	363832	1	3.64	100	100	100000	9707.53
N-r200a0	**654604**	653029	363832	2	0.8	100	100	1717	167.21
N-r200a0	654604	362781	363832	3	0	100	100	100000	9707.89
N-r250a0	1019120	543058	484330	1	2.35	100	100	100000	15069.72
N-r250a0	**1019120**	1017164	480009	2	0.8	100	100	2111	318.80
N-r250a0	1019120	568569	505920	3	0	100	100	100000	15072.21

Table 2: Results of GA stopping when there are no improvements after 10 iterations.

Problem	Best	Our	N	k	It.	N. obj.	T(s)
N-r100a2	145270	137629	100	33	166	16700	2.99
N-r100a2	145270	136943	150	50	139	21000	3.94
N-r100a2	145270	140756	200	73	262	52600	10.36
N-r150a0	360978	350280	150	50	519	78000	31.80
N-r150a0	360978	333162	200	73	225	45200	19.13
N-r150a0	360978	350404	300	100	421	126600	53.59
N-r200a0	654604	587561	200	73	316	63400	48.49
N-r200a0	654604	636285	300	100	479	144000	108.38
N-r200a0	654604	649221	400	133	737	295200	215.29
N-r250a0	1019120	931046	250	83	260	65250	79.97
N-r250a0	1019120	985634	375	125	593	223344	263.55
N-r250a0	1019120	960817	500	166	601	301000	366.69

number of iterations executed, the number of objective function evaluations, and the execution time.

The results in the table reflect the state of the algorithm upon convergence, where convergence is defined as 10 consecutive iterations without improving the best-found solution.

Each problem has been tested using three different population sizes: n individuals (where n is the number of elements in the permutation), $\frac{2n}{3}$ individuals, and $2n$.

The results indicate, on the one hand, that the algorithm's performance is a bit inconsistent. While it often finds solutions very close to the best one founded, there are instances where the results are noticeably worse. Moreover, in these cases, the number of iterations executed is very low, probably due to the elitist selection of the new population. Elitism can lead to premature convergence by reducing diversity within the population. When diversity is lost, all solutions tend to be around a local optimum, making further improvement difficult. Increasing the number of iterations required for convergence would likely enhance the results. However, this highlights that while elitist selection has its advantages, it also presents certain challenges.

On the other hand, it can be observed that increasing the population size usually improves the results, probably because having more individuals makes it harder to lose diversity, preventing premature convergence. Despite this, it sometimes happens, leading to poor results.

Finally, it can be observed that the larger the problem, the longer the algorithm takes to converge. This is because a greater number of possible solutions makes it more difficult to lose diversity within the population. Overall, although the results are not always optimal in some cases, the algorithm is generally able to achieve good solutions in a relatively short time, which is one of the main goals of this type of algorithm.

5.3 Algorithms comparison

Overall, both algorithms demonstrate the ability to achieve good results for most problems. Comparing them, it is evident that when the parameters selected for simulated annealing are well-suited to the optimization problem, the results tend to be better—often coming very close to the best-known solution. However, the genetic algorithm generally maintains greater consistency, approaching more the optimal solution in most cases.

Regarding execution times, both algorithms with the best-found configurations take nearly the same time. However, selecting a bad cooling strategy in the SA algorithm or setting a high population size in the GA algorithm can significantly make big difference in execution times for obtaining bery simmilar objective function values.

6 Conclusions

In this work, two heuristic algorithms have been implemented: simulated annealing and a genetic algorithm. These algorithms were developed in C/C++ to achieve the best possible performance, which has been successfully attained, as reflected in the results. This is particularly important for heuristic algorithms, as in some cases, it may be necessary or beneficial for execution to be as fast as possible.

The SA algorithm implementation includes an empirical method for selecting the starting temperature. Experimental results suggest that the geometric cooling strategy performs best and the *chain_size* parameter plays a crucial role for obtaining a balance between performance and exploration.

Moreover, this work highlights the challenges involved in correctly implementing and using these algorithms. In the case of simulated annealing, there are multiple parameters that need to be fine-tuned, making it difficult to find the optimal configuration. Similarly, for genetic algorithms, the proper selection of operators is crucial and far from trivial.

In some contexts, algorithms that achieve good results in just a few iterations—despite the risk of premature convergence—may be suitable, especially when the priority is to obtain decent solutions as quickly as possible. However, in other scenarios, the focus may shift toward achieving the best possible results, regardless of execution time. The algorithm implemented in this work sometimes struggles with premature convergence, though this issue can be mitigated by running multiple experiments.

Additionally, further experimentation with the parameters of genetic algorithms could lead to different outcomes. Factors such as population size, the number of individuals participating in the tournament, and the probabilities of mutation and crossover can significantly influence the algorithm's behavior and performance.

References

[1] Emile Aarts, Jan Korst, and Wil Michiels. 2005. *Simulated Annealing*. Springer US, Boston, MA, 187–210. doi:10.1007/0-387-28356-0_7
[2] Chang Wook Ahn and R.S. Ramakrishna. 2003. Elitism-based compact genetic algorithms. *IEEE Transactions on Evolutionary Computation* 7, 4 (2003), 367–385. doi:10.1109/TEVC.2003.814633
[3] Josu Ceberio, Alexander Mendiburu, and Jose A. Lozano. 2015. The linear ordering problem revisited. *European Journal of Operational Research* 241, 3 (2015), 686–696. doi:10.1016/j.ejor.2014.09.041
[4] H. A. Eiselt and C.-L. Sandblom. 2000. *Heuristic Algorithms*. Springer Berlin Heidelberg, Berlin, Heidelberg, 229–258. doi:10.1007/978-3-662-04197-0_11
[5] Pablo Estevez. [n. d.]. Optimización Mediante Algoritmos Genéticos. ([n. d.]).
[6] Holger H. Hoos and Thomas Stützle. 2015. *Stochastic Local Search Algorithms: An Overview*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1085–1105. doi:10.1007/978-3-662-43505-2_54
[7] Padmavathi Kora and Priyanka Yadlapalli. 2017. Crossover Operators in Genetic Algorithms: A Review. *International Journal of Computer Applications* 162 (3 2017), 34–36. Issue 10. doi:10.5120/ijca2017913370
[8] Annu Lambora, Kunal Gupta, and Kriti Chopra. 2019. Genetic Algorithm- A Literature Review. In *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*. 380–384. doi:10.1109/COMITCon.2019.8862255
[9] Siew Mooi Lim, Abu Bakar Md Sultan, Md Nasir Sulaiman, Aida Mustapha, and K. Y. Leong. 2017. Crossover and mutation operators of genetic algorithms. *International Journal of Machine Learning and Computing* 7 (2 2017), 9–12. Issue 1. doi:10.18178/ijmlc.2017.7.1.611
[10] Anupriya Shukla, Hari Mohan Pandey, and Deepti Mehrotra. 2015. Comparative review of selection techniques in genetic algorithm. In *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*. 515–519. doi:10.1109/ABLAZE.2015.7154916
[11] Gerhard J. Woeginger. 2003. *Exact Algorithms for NP-Hard Problems: A Survey*. Springer Berlin Heidelberg, Berlin, Heidelberg, 185–207. doi:10.1007/3-540-36478-1_17
[12] Xin Yao. 1992. Dynamic neighbourhood size in simulated annealing. In *Proc. of Int'l Joint Conf. on Neural Networks (IJCNN'92)*, Vol. 1. Citeseer, 411–416.