Comparing A* Search and Genetic Algorithms for the Travelling Salesman Problem

[1]Eduardo Blanco Bielsa, Juan Gómez Tejeda, Fernando Sáenz de Santa María Modroño. Sistemas Inteligentes. Grado en Ingeniería Informática del Software. EII. Universidad de Oviedo. Campus de los Catalanes. Oviedo.

Abstract. This article presents two approaches to solving the classic Travelling Salesman Problem (TSP): one based on metaheuristics and another on genetic algorithms. The TSP involves finding the shortest route that visits a set of cities and returns to the starting city without repeating any city. We evaluate the performance of these approaches using various data sets and compare the results obtained between them, in addition to comparing them with known theoretical results. We also analyse the performance of the approaches in various instances of the problem, which provides a deeper understanding of their behaviour. Our results reveal significant details about the effectiveness of these methods in addressing the TSP, thus contributing to the field of combinatorial optimization and offering valuable insights for future research in this domain.

Keywords: A*, genetic algorithms, TSP, heuristic search, crossover operator.

1 Introduction

In the field of combinatorial optimization, the classic travelling salesman problem or Travelling Salesman Problem (TSP) poses a fundamental challenge: finding the route that visits a set of cities and returns to the starting city without re-treading any ground.

Given the complexity of this problem, the use of multiple search techniques have emerged as crucial tools. In this context, we explore solving this problem from two widely known approaches: **heuristic search** and **genetic algorithms**.

This study investigates the effectiveness of these two techniques in solving the TSP, analysing their performance, cost and quality in various configurations. Our goal is not only to deepen understanding of these methods, but also to propose new perspectives for efficient resolution of complex combinatorial optimization problems, with significant applications in the real world and computer science. Therefore, throughout the sections of this study we expose key definitions, algorithms used in the search for solutions, different applications of the TSP, comparisons between the different techniques, the results obtained, and conclusions of the study, all in search of a correct understanding of it.

[1]Authors' names by alphabetical order

2 The travelling salesman problem

[1] The Travelling Salesman Problem (TSP) answers the following question: given a set of cities and the distances between each pair of them, what is the shortest possible route that visits each city exactly once and returns to the starting city? This is an NP-Complete problem within combinatorial optimization, really important in operational research and computer science.

[2] The TSP has various applications even in its simplest formulation, such as: X-ray crystallography, electronic circuits, vehicle routes and mask layout in motherboard productions.

[3] Due to the interest generated by the problem, several methods have been discovered to try to solve it. Some of the classic resolution methods are: *brute force approximation*, *dynamic programming* and *Christofides algorithm*. However, in recent years new methods have also been discovered such as *Geometry of Polynomials*, *Fractional Solutions* and *rounding techniques*.

3 Search algorithms

For this study, the following two search algorithms have been chosen:

The ***A** **algorithm** is a graph search algorithm where a function, called *f(n)* being *n* a specific state, is used to select which states to expand and cut, so that the minimum values of that function are prioritised. In this case, this function *f(n)* consists of the sum of the cost to that state *n* (called *g(n)*), and a heuristic function *h(n)*. It is important that this heuristic function *h(n)* [4] is well informed, as it gives better solutions and avoids the expansion of unnecessary nodes, thus reducing the problem resolution time.

The **Genetic Search algorithm** [5] is based on the generation, selection, crossing and mutation of populations of individuals for a certain number of generations. This selection is carried out using the *fitness(n)* function (a function that measures how good an individual is for the given problem) to select the better individuals, later used for cross breeding with other good individuals in hopes to find offspring better than its parents.

4 Application of these techniques to the TSP

4.1 Heuristic search using A*

To apply the A* algorithm to the problem, it is important to talk about the search space used and the heuristics that will be evaluated later.

4.1.1 The search space

The TSP search space consists of all combinations of valid paths, a total of N! possible states for a number N of cities. By always fixing a city as the start, this is reduced to (N-1)! states. In any case, the complexity of the problem is not significantly reduced.

To describe a state of the space, a tuple is used that contains a list of the cities already visited, and the current city. These states do not represent solutions to the problem, only subproblems to be solved. To retrieve the solution, that is, the order in which the cities should be visited, the algorithm stores in each node, apart from its state, the node from which it was expanded.

The described tuple does not contain the initial city, as it is fixed in the code and is not necessary for the complete description of a state.

4.1.2 Heuristics used

Heuristic design using problem relaxation consists of simplifying the problem methodically to convert it into a problem solvable in polynomial time. To do this, a series of restrictions of the original problem are established, and some are ignored. Creating heuristics using this method ensures their admissibility and consistency [6]. In addition, it allows comparing the dominance of the generated heuristics.

Regarding the TSP, given a state in which k >= 1 cities have been visited and we are in city X, we are looking for a subset of arcs from the graph of minimum cost, with the following restrictions:

R1. It has N-k+1 arcs.
R2. The arcs touch the initial city, the not visited cities, and X.
R3. That the set of arcs is degree 1 for both the initial city and X, and degree 2 for the not visited.
R4. That the arcs connect between them the initial city, those not visited, and X.

h1 is the only heuristic studied here that does not correspond to a relaxation of the problem and establishes a minimum bound for the solution. It uses a residual graph composed of arcs that connect: the initial city, the current one, and those not visited among them. From this graph, it selects the minimum arc that leaves each city and adds their costs. By sticking to a minimum cost bound of the problem, it is an admissible heuristic.

h2 is the simplest heuristic studied here. It is derived from relaxing R2, R3 and R4. Thus, it looks for the set of N-k+1 minimum cost arcs from the residual subgraph. Given that h2 relaxes the TSP a lot, it could be said that it is quite uninformed.

The h3 heuristic comes from relaxing both R3 and R4 restrictions. Thus, it tries to obtain a subset of N-k+1 arcs that touch all cities in the residual graph with lowest cost.

This could leave cities disconnected but is a better approximation than h2 which does not even consider visiting all cities.

Obtaining the value of h3 is an iterative process over a matrix. This matrix represents all arcs in the residual graph. In each iteration, a minimum cost arc is obtained and its value is added to total. In addition, the row and column where this arc was located are removed from the matrix indicating that route has already been taken leaving one city and arriving at another. Process continues until the expected number of arcs is reached, and the accumulated sum is returned.

h_MST, like the last two, is derived by problem relaxation. Specifically, this heuristic only relaxes R3. Thus, what it calculates is the cost of the minimum expansion tree of the residual graph.

There are algorithms that solve this problem in polynomial time. In our case, we have chosen Kruskal's algorithm [7]. For this, a class was obtained that implements the algorithm [8]. This heuristic is very well informed, being better than both h2 and h3. However, Kruskal's algorithm is not simple, with a complexity of O(N log N), which could make it impractical in the real world for very large and complex graphs.

4.2 **Search through genetic algorithms**

The search space for genetic algorithms is represented differently than for A* search. We can make the same simplification of fixing a city as the initial one, and thus the gene is composed of a list of the rest of the cities in any order, which describes the path to be taken.

The crossing of two individuals must maintain the gene as a permutation of the order of the cities. Therefore, there are two main options: uniform crossover and two-point crossover. The first consists of randomly copying some genes from one parent, and "filling in" with the genes from the other in order. The two-point crossover selects a randomly sized section from one of the parents to copy to the offspring, and "fills in" with the genes from the other parent in order, skipping cities already added by the first parent. This second one is intuitively better, as it is able to preserve good path sections from the parents, if there are any. The mutation of an individual is done by swapping two adjacent cities, selected randomly.

The fitness function should reward less costly paths. To do this, the cost of the path described by the gene is calculated and introduced into a constantly decreasing positive function. One could simply use the inverse function (1/x), but the problem with this is that it rewards good paths too little compared to bad ones, in the cost range we are working with. We have found that it is best to use a function with the form $a^{b/x}$, where x is the path cost, a is a parameter greater than 1 to indicate how much more you want to reward less costly paths, and b is a parameter that indicates an approximate expected cost of the solution. This adequately rewards individuals with lower cost and significantly improves the solutions found.

5 Experimental study

5.1 Design of the experimental study

To carry out the experimental study, the described above must be executed. Specifically, we used a computer with an Intel i7-12800 HX 2 GHz processor. For the A* study, each heuristic was run a total of 20 times to adequately measure its performance. In addition, it was run on 4 different instances of the problem of different sizes, two of them obtained from the TSPLIB [9] (gr17 and gr21) and two of them created by us, of size N=12 and N=19.

For the study of genetic algorithms, due to their random nature, each combination was run 30 times. Apart from the same four instances mentioned, it was tested with the two crossover operators described above, uniform crossover and two-point crossover. Mutation of an individual was randomly performed 10% of the time, after the crossover operation. The parameters a and b of the fitness function have been initialised as a=3 and b=10000.

5.2 Experimental results

5.2.1 A* search

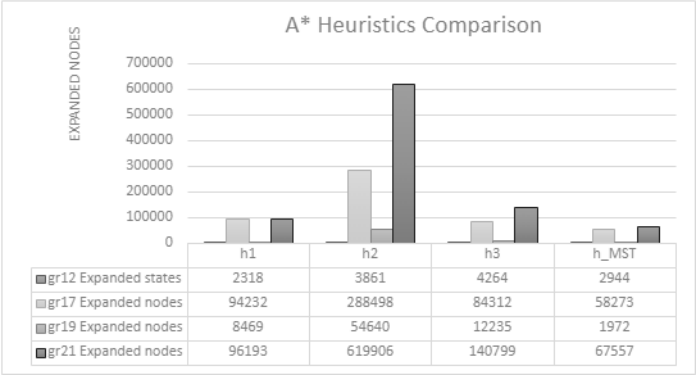


Fig 1: Comparison chart of expanded nodes for different heuristics.

In Figure 1, we show the comparison in terms of expanded nodes using A* search between the heuristics h1, h2, h3 y h_MST.

We firmly believe that the heuristics *h1* and *h2* meet the **admissibility** property, as they support the theoretical results by finding the optimal solution in all our tests for each instance. However, the results obtained with the heuristics *h3* and *h_MST* are sufficient to determine that their implementation generates a non-admissible heuristic,

opposite to what was described theoretically. This implies the incorrect implementation of these functions.

Table 1: Re-expanded states for each instance and heuristic.

Instance	gr12	gr17	gr19	gr21
h1	1	2	0	0
h2	0	0	0	0
h3	357	13467	2081	19309
h_MST	0	0	0	0

Also, the implementation of *h3* is non-monotonic. As seen in Table 1, the A* algorithm re-expands states when using this heuristic, which is a sign of non-monotonicity. The non-monotonicity of h1 is demonstrated in a similar way. Regarding the monotonicity of h2 and h_MST, it cannot be confirmed given their behaviour in the observed instances, beyond saying that, from the theoretical study, they should be monotonic, and that nothing observed contradicts this.

5.2.2 Genetic algorithms

As can be seen in Figure 2, the order crossover (ox) operator finds the lower minimum costs of the two operators (yellow line), while the uniform crossover (ux) finds the lower average costs of the two (blue line).

Regarding convergence, it can be seen that for smaller problem sizes, a much faster convergence occurs. In the case of the size 12 problem, which converges almost at the beginning of the process, the size 17 at 50 generations, the size 19 at around 125 and the size 21 just start to converge at 200 generations, at the end of execution. However, the order crossover (ox) operator converges faster than the uniform crossover (ux) in all cases, as can be seen in the graphs.

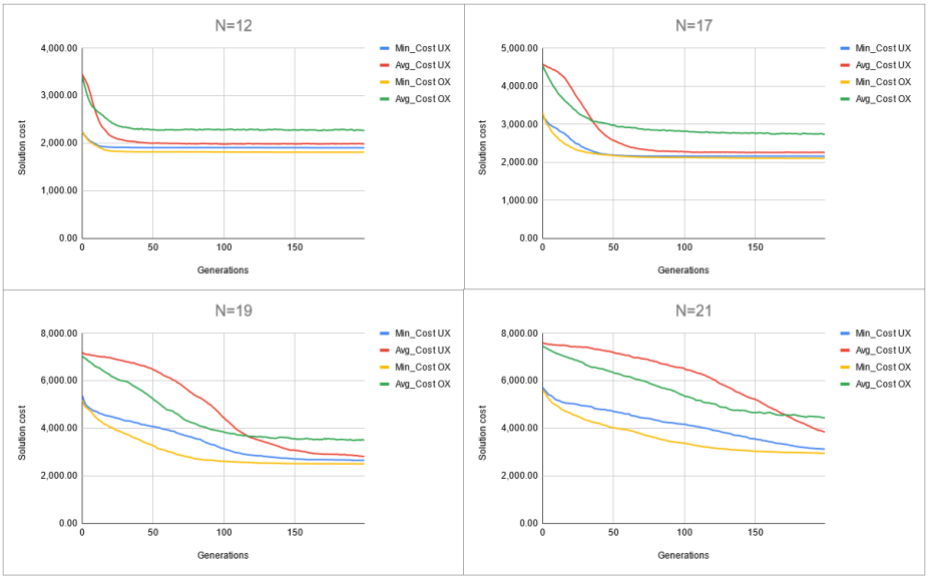
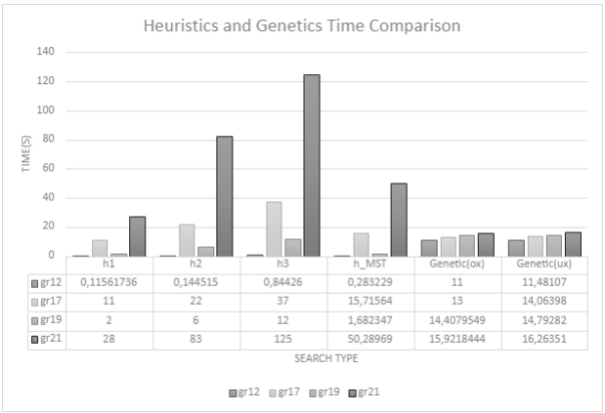


Fig 2: Comparison graphs of crossover operators for different problem sizes.

5.2.3 Execution times

As can be seen in Figure 3, in all heuristics there is a trend where execution time increases with size. It can also be seen that search using heuristic *h1* is the fastest to find the solution, even though heuristic *h_MST* expands fewer states, possibly because the heuristic calculation is more computationally costly. In addition, it can also be seen that there is not really a significant difference in execution time between both crossover operators.



	h1	h2	h3	h_MST	Genetic(ox)	Genetic(ux)
gr12	0,11561736	0,144515	0,84426	0,283229	11	11,48107
gr17	11	22	37	15,71564	13	14,06398
gr19	2	6	12	1,682347	14,4079549	14,79282
gr21	28	83	125	50,28969	15,9218444	16,26351

Fig 3: Graph with comparison data between execution times of heuristic search and genetic algorithms

Lastly, genetic algorithms are much more consistent regarding execution time than the A* algorithm, since both cross operators take practically the same time for all problem sizes, taking longer than *h1* in the case of N=12, N=17, and N=19, but less in the case of N=21. Therefore, it can be said that genetic algorithms are more suitable for large problem sizes than A*, as they find usable solutions in a much better time, although they are worse for small problem sizes, as they do take longer without always finding optimal solutions.

6 Conclusion

Genetic algorithms are significantly more stable in terms of execution time, and scale much more predictably with the size of the problem. However, they do not guarantee an optimal solution. In fact, as observed, convergence on a good solution takes longer for larger problem sizes, so more and more generations may be needed as N increases.

A* search is deterministic, and with a good heuristic that is both fast and admissible, like h1, it ensures to find the optimal solution; although there is no certainty of execution time as it is not necessarily consistent with the problem size.

When choosing one or the other, it will depend on one's needs: if the optimal solution is necessary, and it is not going to be executed often, it would be best to use A*. However, if a good solution is enough, and you don't need an optimal one, and you value being able to limit the execution time to a maximum bound, the search by genetic algorithms is clearly superior.

7 Bibliography

[1] Wikipedia. Problema del viajante.
[2] InTech, Donald Davendra, Teodora Smiljanic. Travelling Salesman Problem, Theory and Applications. (2010).
[3] Leanne White, RoadWarrior. Solving the Travelling Salesman Problem (2023).
[4] Investigation of the * (Star) Search Algorithms: Characteristics, Methods and Approaches https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=831ff239ba77b2a8eaed473ffbfa22d61b7f5d19
[5] STEPHANIE FORREST: Genetic Algorithms https://dl.acm.org/doi/pdf/10.1145/234313.234350
[6] Pearl, J.: Heuristics: Intelligent Search Strategies for Computer Problem Solving, pp. 115-116. Addison-Wesley Publishing Company, Reading, Massachusetts (1984).
[7] Kruskal, J. B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society. **7**, pp. 48–50 (1956).
[8] Javatpoint. Implementation of Kruskal's Algorithm in Python.
[9] Reinelt, G.: TSPLIB–A Traveling Salesman Problem Library (1991).