Heuristic Search and Genetic Algorithms Comparison: Traveling Salesman Problem

Emilio Cortina Labra[0000−0002−8191−6023]
UO257322@uniovi.es

Intelligent Systems. Bachelor Degree in Software Engineering.
EII. Universidad de Oviedo. Campus de los Catalanes. E-33007. Oviedo

Abstract. The Traveling Salesman problem is far from being a trivial task to compute. Over the years several approaches have been taken to solve it for a given number of cities. We will take a look at the problem itself, its applications and how it can be solved.
For this we will use different methods, one of them being State-space Search Algorithms and the other Genetic Algorithms, both completely different.
Finally we will draw conclusions on how these two compare on solving the problem.

Keywords: TSP · A* · Genetic Algorithms · heuristics

1 Introduction

The Traveling Salesman Problem has been a well known pathfinding task since it was first described. In this article we will take a look at its history, how it is defined and the impact that it has on different fields. Then we will introduce some algorithms that can be used to solve this problem.

The first of them being Heuristic Search Algorithms. These algorithms are a special kind of State-space Search Algorithms that make use of knowledge about the problem they are solving to compute the solution. We will specifically see two implementations: A* and Static-weighted A*.

The other group of algorithms we will study are Genetic Algorithms. They are search and optimization algorithms based on the mechanisms of natural evolution.
After these algorithms have been introduced, we will see how to apply them for solving the Traveling Salesman Problem. The problem needs to be modeled following the structure of each approach.
Finally we will carry out different experiments and draw the conclusions about how these algorithms have behaved solving the problem.

2 Traveling Salesman Problem (TSP)

2.1 History

"In 1832, a German traveling salesman published a handbook describing his profession. His name is unknown; he only stated that the book was written by

"one old travelling salesman." However, he has come down in history thanks to a rather simple and obvious observation. He pointed out that when one goes on a business trip, one should plan it carefully; by doing so, one can "win" a great deal of time and increase the trip's "economy." Two centuries later, mathematicians and scientists are still struggling with what is now known as the "Travelling Salesman Problem" (TSP)" [1]

2.2 Description

The problem defines a set of cities along with the associated cost of travel between each pair of them. The goal of the problem is to find the cheapest path or circuit that goes through every single city exactly once, and finally returns to the starting position.

Given the simplicity of the statement of this problem, it may seem like an easy task to solve. It's only about thinking on the different possible paths, add the costs of each one and then compare the outcomes to find out which was the one with the least total cost. This is feasible to compute even for a human when we are dealing with small instances of the problem, but it is not trivial at all to compute for bigger instances of the problem, even for powerful computers.

As it can be seen in Table 1 the number of the different possibilities grows rapidly with the number of cities defined in the instance of the problem.

Table 1. Number of possibilities for each instance size of the problem.

Number of cities	Number of possibilities
5	5! = 5*4*3*2*1 = 120
25	25! = 25*24*23*...*3*2*1 = 15,511,210,043,330,985,984,000,000
100	100! = 93,326,215,443,944,152,681,699,238,856,266,700,490,715,968,264,381,621,468,592,963,895,217,599,993,229,915,608,941,463,976,156,518,286,253,697,920,827,223,758,251,185,210,916,864,000,000,000,000,000,000,000,000

2.3 Applications

Many different fields make use of the TSP to solve several problems. Some of these use cases are obvious, like the planning of school bus routes or the routing of a laundry van, but others can seem more far-fetched at first. A few applications of the problem will be discussed in this section.

Logistics This is the category that the TSP was originally described for. Different transport industries are using systems that calculate routes by means of the

TSP. Although it was Merill Flood [4] who first introduced the idea of applying the problem to plan school bus routes back in 1956, this application is still in use nowadays, with several companies building software solutions for it.

Genome sequencing By making analogies of the problem, researchers working on genome sequencing at the National Institute of Health have used the TSP to construct radiation hybrid maps. They integrated local maps into a single radiation hybrid map for a genome, the cities being the local maps and the cost of travel being a measure of how likely one local map will immediately follow another [5].

Telescopes The problem can also be used with locations that cannot be physically reached or visited, for example with planets, stars and galaxies. For observing these objects telescopes must be set into position. The process of rotating the equipment is called *slewing*, and for bigger telescopes it is a complicated and time-consuming movement that has to be performed by computer-driven motors. By making use of the TSP the aim is to minimize this slewing movement for a given set of celestial objects. In the TSP these objects would represent every city, and the costs is the slewing time to move the telescope from one object to the next [3].

3 Heuristic search algorithms

Heuristic search algorithms are a special kind of State-space search algorithms. Contrary to uninformed state-space search algorithms, Heuristic algorithms make use of knowledge about the problem they are trying to solve in order to lead the search towards more promising areas of the search space, so that fewer nodes are expanded to find the solution.

However, the use of knowledge about the problem comes with a trade-off, which is the computational cost of taking into account this information.

3.1 A* algorithm

This is algorithm is the most popular choice for pathfinding, as it's flexible and can be used in a wide range of domains [6]. A* algorithm is a special case of the Best-first search algorithm [8,9], that defines the *evaluation function* in a particular way, given any node n from the search space:

Definition 1. *g*(n) is the cost of the shortest path from the starting position to n.*

Definition 2. *h*(n) is the cost of the shortest path from the starting n to the nearest objective.*

Definition 3. *f*(n) = g*(n) + h*(n) is the cost of the shortest path from the starting position to an objective passing through n.*

Definition 4. *C* = f*(initial) = h*(initial) is the cost of the optimal solution.*

But these are ideal values that cannot be exactly calculated for big problems in a reasonable amount of time. If we knew these values for every node, we would have a fast algorithm that only expands states that lead to the optimal solution.

However, we can work with approximations of the functions above.

Definition 5. *g(n) is the cost of the shortest path from the starting position to n, found so far.*

Definition 6. *h(n) is the heuristic function.*

Definition 7. *f(n) = g(n) + h(n) is the evaluation function we will use.*

Heuristic function The heuristic function *h(n)* is an approximation of *h*(n)*. Designing the heuristic requires some knowledge about the domain that is usually obtained by analyzing the problem. This function deeply determines the behaviour and properties of the A* algorithms, such as its admissibility, monotony, consistency... [10].
For example, using a more informed heuristic, that is, one with a value is closer to that of *h*(n)* will result on A* expanding fewer nodes at the cost of computational overhead. On the contrary, using a less informed heuristic results on less computational overhead at the cost of expanding more nodes to find the optimal solution. It can also be the case in which an optimal solution is not found if the heuristic is not admisible, that is *h(n)>h*(n)*.

3.2 Static-weighted A* algorithm

It is usually assumed that increasing the weight on the heuristic will decrease search time, and that greedy search will provide the fastest search. Although there is no formal guarantee for this, weighted A* relies on this mechanism to speed up the searching process, and is the most popular satisficing algorithm for heuristic search [11].

For achieving this, static-weighted A* defines the following evaluation function:

$$f(n) = g(n) + \varepsilon * h(n) \tag{1}$$

where

$$\varepsilon > 1 \tag{2}$$

Static-weighted A* has a bias towards states that are closer to the goal [7].

4 Genetic algorithms

Genetic algorithms are search and optimization algorithms based on the mechanisms of natural evolution. By mimicking this process, they are able to "evolve" (iterative process) solutions to real world problems if they have been suitably encoded [12].

Genetic algorithms start with an initial population of chromosomes that represent potential solutions. From this initial population it evaluates each chromosome and determines its fitness. This is a metric that represents quality of the solution, it is analogous to the adaptation degree of organisms in natural evolution.

The algorithm then continues by making the selection process. The selection process starts the transition from one generation to the next by giving individuals with large fitness more probability to be selected. After the selection occurs the crossover is performed. The crossover operator takes chromosomes that have been selected from the previous generation and combines them to create the children chromosomes that will make up the next generation. Finally the mutation process takes place, where the chromosomes resulting from the previous phase experience small variations on their structure in order to introduce more diversity in the population.

This is mainly how a genetic algorithm works, but some aspects such as parameter tuning have been omitted for the sake of simplicity. Some aspects worth noting is the importance of genetic diversity in the populations. A higher diversity is key for the algorithm to converge properly. This may seem like an easy task to achieve through random generation of the initial population, but diversity is also lost every time the genetic operators are applied, which can result in the algorithm converging prematurely if the different parameters are not tweaked appropriately.

For a more detailed explanation on genetic algorithms see [12].

5 Solving the Traveling Salesman Problem

5.1 Problem solving using heuristic search

For solving the Traveling Salesman Problem using heuristic search we will define every state as follows: In every state we have already visited a subset of the cities, now we need to find a path that connects the last visited city with the starting city.

Some heuristic functions have also been defined and will appear in the experiment:

Definition 8. *h_0 is the null heuristic with value 0 for every node.*

Definition 9. *h_1 is the minimum cost arcs heuristic.*

Definition 10. *h_2 is the sum of the minimum arcs heuristic.*

Definition 11. *h_3 is the spanning tree heuristic.*

We will take a look at how these heuristics behave while carrying out the experiment.

5.2 Problem solving using genetic algorithms

For defining each chromosome we will use permutations. A permutation (i_1,...,i_n) represents an ordered tour i_1 to i_2, ..., i_n to i_1.

The evaluation function computes the total cost of the path encoded in the chromosome. The fitness function returns 0 if there is any repeated city in the chromosome (invalid solution), or the inverse of the evaluation function (a higher value of the evaluation function means a longer path that translates in less quality) otherwise.

As crossover operator we are going to use a Two-Point Crossover operator. It takes two parents a copies a subsequence from the first parent directly to the offspring, the remaining cities it copies from the second parent in relative order.

For the mutation operator we will perform a simple random swap in the order of two cities.

6 Experimental study

Different experiments will be carried out in every subsection. However, the conclusion of each subsection will be used in the next, so the order is important.

6.1 Heuristic comparison

We will start by making an analysis of the different heuristics that we have defined with the standard A* algorithm. For this experiment we will use an instance with 6 cities and measure the nodes expanded by every heuristic. Notice that we will not consider the execution time of the algorithm in this experiment. Due to the small size of the instance this metric is not sufficiently reliable.

Every heuristic found the optimal solution to the problem. However, as it can be seen in Figure 3 there is a difference in the amount of nodes expanded by each heuristic. We are interested in h_3 and is the one we will use in the following sections, as it seems to be the most efficient managing memory.

6.2 Static Weighting the heuristic

Taking h_3 (we decided before it was the most interesting of all), we want to see what effect adding a static weight has on the solutions found by the algorithm. We have now chosen a bigger instance of 24 cities for this problem. First, we take some values for ε in the interval [1,2].

As it can be seen in Figure 3 the biggest gap happens between 1 (the heuristic without weighting) and 1.2, then it tends to stabilize. It is important to note also

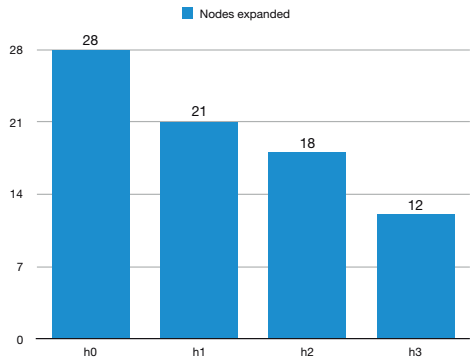

Fig. 1. Amount of nodes expanded during the execution of the algorithm using each heuristic function.

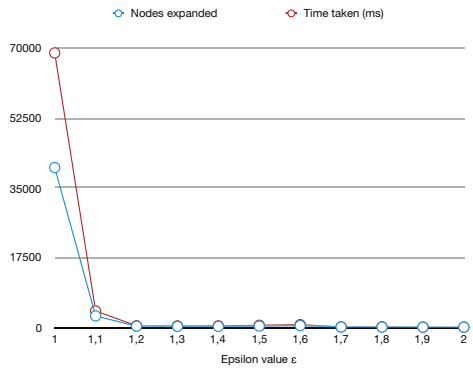

Fig. 2. Nodes expanded and time taken for the different values of ε with the heuristic h_3.

that the algorithm found the optimal solution for values less or equal than 1.5 (included). For greater values of ε the solutions are further from the optimal, but fewer nodes are expanded resulting in better performance. For bigger instances of the problem, it becomes necessary increasing the weight of the heuristic if we don't want to run out of memory and time.

6.3 Genetic algorithm approach

For this experiment we have used an instance with 24 cities, a population size of 50 chromosomes, a total amount of 50 generations and a probability of crossover of 1. We also have implemented elitism, that is, the best chromosome in each generation will be preserved to the next.

What we want to test is how the probability of mutation affects the execution of the algorithm. The mutation process adds diversity to the population, and the diversity of the population is supposed to determine how the algorithm will converge.

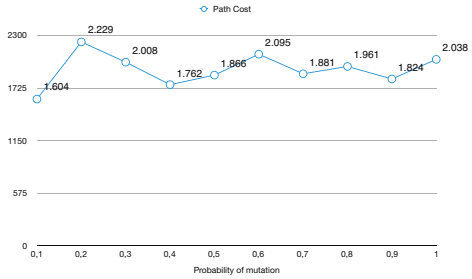

Fig. 3. Minimum path cost in the population for every value of the probability of mutation.

In Figure 3 we can see that even although the probability of mutation is supposed to be tightly related to the obtained solution, in our case doesn't seem to be that way. Another reason for this results can be that genetic algorithms rely on many random variables and is more difficult to draw conclusions from tweaking parameters.

7 Conclusions

As it can be seen in the different experiments, there is a constant trade-off between optimality of the results and execution time. Where genetic algorithms can manage to give a solution for bigger problems, even if it is far from the optimal, state-space search algorithms struggle to accomplish the same.

However, when it comes to smaller instances of the problem, like the ones tested in the different experiments described, state-space search algorithms take

the lead. A* and Static-weighted A* have proved to give optimal and nearly optimal solutions while outperforming the genetic approach.

Even for bigger instances, if we set a higher value for the weight of the heuristic we can obtain much better solutions in shorter time compared to the genetic algorithm, so it is safe to assume that State-space Search Algorithms are more suited to solve the Traveling Salesman Problem than Genetic Algorithms.

References

1. Pacha-Sucharzewski, Mateusz (2011). Analysis of the "Travelling Salesman Problem" and an Application of Heuristic Techniques for Finding a New Solution. Undergraduate Review, 7, 81-86. Available at http://vc.bridgew.edu/undergrad_rev/vol7/iss1/17
2. Belal Ahmed et al 2017 IOP Conf. Ser.: Mater. Sci. Eng. 263 042085
3. David L. Applegate (2011). The traveling salesman problem: a computational study. Princeton University Press, Year: 2006
4. Flood, M. M. 1956. The traveling-salesman problem. Operations Research 4, 61–75.
5. R. Agarwala, D.L. Applegate, D. Maglott, G.D. Schuler, and A.A. Schaffler. A Fast and Scalable Radiation Hybrid Map Construction and Integration Strategy. https://www.ncbi.nlm.nih.gov/genome/rhmap/manuscript.ps
6. Amit's Thoughts on Pathfinding - Introduction to A*. http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html Last accessed October 23rd 2019.
7. Maxim Likhachev. A* and Weighted A* Search. Carnegie Mellon University. https://www.cs.cmu.edu/~motionplanning/lecture/Asearch_v8.pdf Last accessed October 23rd 2019.
8. Best-first search. CSE 326: Data Structures, Summer 2003. University of Washington Computer Science and Engineering https://courses.cs.washington.edu/courses/cse326/03su/homework/hw3/bestfirstsearch.html Last accessed October 23rd 2019.
9. Best-first search. Wikipedia. https://en.wikipedia.org/wiki/Best-first_search Last accessed October 23rd 2019.
10. Amit's Thoughts on Pathfinding - Heuristics. http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html Last accessed October 23rd 2019.
11. Christopher Wilt and Wheeler Ruml. When does Weighted A* Fail? University of New Hampshire http://www.cs.unh.edu/~ruml/papers/wted-astar-socs-12.pdf
12. David Beasley. An Overview of Genetic Algorithms: Part 1, Fundamentals. University of Cardiff http://mat.uab.cat/~alseda/MasterOpt/Beasley93GA1.pdf