Analysis of Algorithms for the Euclidean Traveling Salesman Problem

Iago Zagnoli Albergaria[1], **Manuel Junio Ferraz Cardoso**[1]

[1]Departamento de Computação – Universidade Federal de Minas Gerais (UFMG)

`{iagozag,mjfc}@ufmg.br`

Abstract. *The Euclidean Traveling Salesman Problem (ETSP) seeks the shortest tour visiting a set of points in two-dimensional Euclidean space. This paper examines branch-and-bound and two approximation algorithms, focusing on their complexity and performance when solving the problem. Furthermore, the experimental comparisons highlight trade-offs between solution accuracy and computational efficiency / memory consumption.*

1. Introduction

The Euclidean Traveling Salesman Problem (ETSP) is a classical optimization problem that involves finding the shortest possible path that visits a set of points in a two-dimensional Euclidean space. It is an instance of the Traveling Salesman Problem (TSP) that has been widely studied due to its computational complexity and extensive applications in logistics, transportation, and circuit design. In the Euclidean version, the distance between two vertices is computed using the Euclidean distance in a two-dimensional plane.

This paper focuses on the analysis of algorithms designed to solve the ETSP, specifically with branch-and-bound and two approximation techniques. Branch-and-bound is an exact method that explores possible solutions while pruning suboptimal branches based on bounds. On the other hand, approximation algorithms provide faster results, though they may not always produce optimal solutions. The main goal of this study is to evaluate and compare these algorithms, analyzing their computational complexity, memory usage, and solution accuracy.

The remainder of this article is organized as follows: Section 2 details the implementation of the algorithms and the methods selected for each. Section 3 presents the results of the algorithms across various test cases. Finally, Section 4 provides the conclusions drawn from the analysis.

2. Implementation

2.1. Branch and Bound

The initial implementation of the algorithm used Best-First Search and, during testing, performed well in smaller instances. However, when scaling to larger cases, the algorithm consumed excessive memory, causing the program to crash, and, in some instances, even crashing the computer. As a consequence, there was a switch to Depth-First Search.

First, the initial bound for the graph is calculated, the optimal value is initialized to infinity, and a vector of booleans is created, with only the first vertex marked as visited.

Each state node has five parameters: the lower bound of the remaining graph, the cost up to the node, the level of recursion, the current vertex, and the vector of visited vertices. Moreover, the vector of booleans is passed by reference to save memory.

The initial bound is calculated as the sum of the two smallest edges leaving each vertex, divided by two. This is a lower bound because every path uses two edges per vertex, and selecting the two smallest edges provides the best possible estimation. When moving to a node, the bound is decreased by the sum of the second smallest edge of the vertex being left and the first smallest edge of the vertex being entered, divided by two. In addition, the value of the edge being traversed is increased.

Other than that, the function goes straight to the point. It tries every permutation of the vertices, with the exception that if the bound for a vertex is greater than a previous estimation, that node is pruned from the recursion.

2.2. Twice Around The Tree

It is known that the ETSP is a NP-Hard problem, meaning that no algorithm can find the exact solution in polynomial time. However, if we stop chasing the optimal answer and try to find another one that is close to it, we can be more efficient than that. One of the algorithms that implement this idea is the Twice Around the Tree (TATT) algorithm.

To understand how the algorithm works, we need to recognize a property of the problem: in the optimal path, removing any edge from it results in a tree that contains all the points. In other words, the path will become a Spanning Tree. We will refer to the spanning tree obtained this way as ST.

The TATT algorithm begins by generating a Minimum Spanning Tree (MST) in the given set of points. In this implementation, Prim's algorithm is used to construct the MST. Since this tree is the least weighted spanning tree, its weight will be no greater than that of the ST. By doubling the edges in the MST, it will contain an Eulerian Cycle that is, at most, twice as weighted as ST (and also the optimal path, since ST is not heavier than the optimal answer).

Subsequently, the algorithm finds the Eulerian Cycle contained in this MST (after double all its edges) using the Depth First Search (DFS) algorithm. Finally, any duplicate occurrences of points are removed from the path. For example, if x,y,z are points and y has already appeared in the path, we change the subsequence xyz to xz.

However, in this implementation the edges in the MST are not duplicated. Instead, the DFS simply inserts each point at the end of the path when it is visited during a Preorder traversal of the MST. This results in a path that includes every point exactly once, which is the output of the algorithm (and is at most twice the optimal cost).

2.3. Christofides

Another algorithm that finds a solution that is close to the optimal is the Christofides algorithm, which provides an answer that is closer to the optimal than the one found by the TATT algorithm.

The first part of the Christofides algorithm is the same as the first part of TATT: it finds the MST (we will call it T) using Prim's algorithm. After this, a subgraph G is created, containing all the vertices with odd degrees in T, and all the edges between these vertices (i.e., a complete subgraph). Since the sum of the degrees of every vertex in any graph is even, the number of vertices with odd degrees of every graph is even, so T has an even number of vertices with odd degree.

The next step is to calculate a minimum perfect matching of G (in practice, a maximum matching is calculated, but we adjust the edge weights to transform it into a minimum perfect matching of G in terms of the original weights). Since G is a complete graph with an even number of vertices, we know that this matching is always perfect. In this implementation, the Blossom algorithm was used. Our implementation has used an open source library, LEMON, to find the matching ([Dezső et al. 2011]).

Let M be the graph that contains all the vertices of G and the edges used in the matching. We create a new graph H that contains all the vertices in the input and all the edges in T and in M. If an edge is in both, we add it duplicated (that is why we have used a data structure that increases the complexity of the algorithm). Now, H is a connected graph that every vertex has an even degree (so, H is an Eulerian Graph).

The final step is to find this Eulerian cycle. The Hierholzer's Algorithm is used to make this. Then it is transformed in a Hamiltonian path the same way as described in the Twice Around The Tree algorithm (the idea, not the way it was implemented): remove any occurrence of any vertex that has already appeared before. The final path produced is the output of the algorithm, with a cost at most 1.5 times the optimal one.

3. Results

Most of the tests were from the TSPLIB library [Reinelt 1991], available at http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. Some additional tests were taken from the work of [Deĭneko et al. 1997].

3.1. Branch and Bound

The branch-and-bound method always provides the optimal answer, but the asymptotic complexity is O(N!). As a result, even for instances with a small number of vertices, the program takes over 30 minutes and produces NA. In fact, this happens with all tests from the TSPLIB library [Reinelt 1991]. Given that, three additional tests were added: two of them are from [Deĭneko et al. 1997], and one consisting of a simple line graph with 15 points (0,0), (0,1), . . . , (0,14).

The first test case is a Kalmanson point set with 10 points that executes in less than 1 millisecond. The second test case is a Demidenko point set with 15 points that is completed in 16 milliseconds. The third test case is a manually created line graph with 15 points designed for comparison with the Demidenko point set. This comparison highlights that the Demidenko point set is an easier case for the ETSP, as it completes in only 16 milliseconds, whereas the line graph, with the same number of vertices, fails to terminate even after 30 minutes of execution. The reason the first two test cases are computed quickly is explained in the article by [Deĭneko et al. 1997].

3.2. Twice Around The Tree

This algorithm gives an approximate answer, not the optimal one. SO, the first analysis will be over the quality of the solution. The graph below shows how close to the optimal solution the found was for every test case:

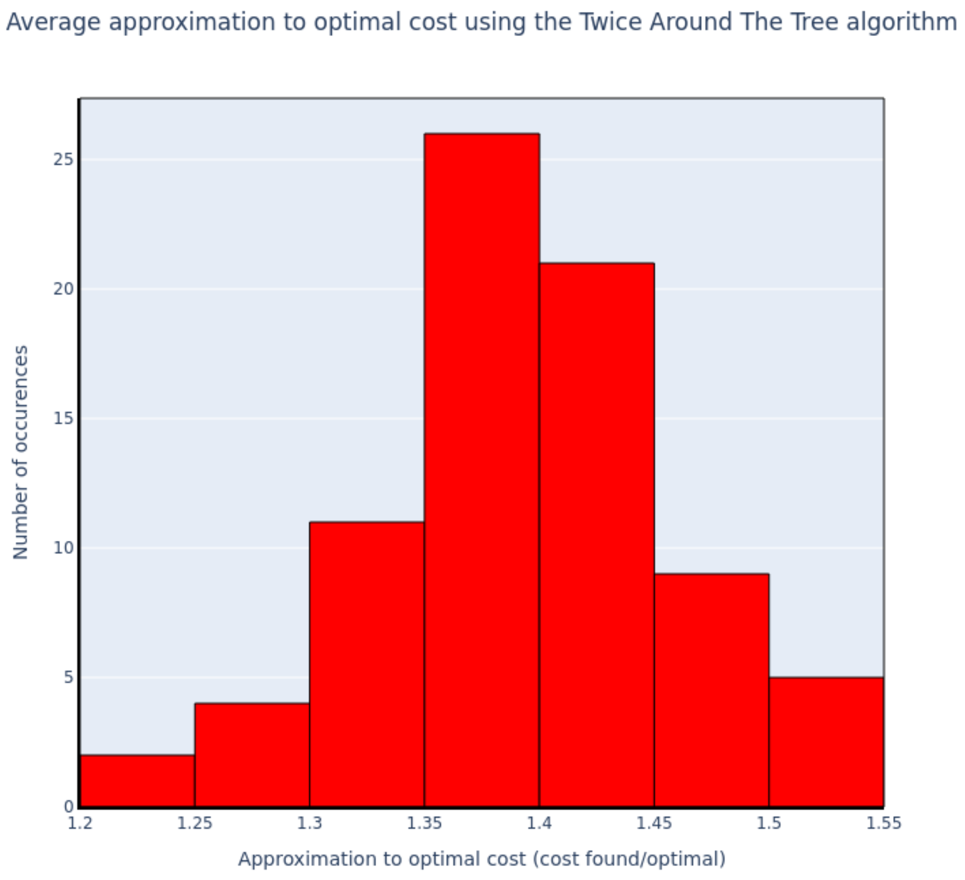

Figure 1. Histogram of approximation rates

The algorithm has a guarantee that the found answer is at most the double of the optimal. But, as we can see, the it usually gives a cost that goes from 1.3 to 1.5 times the optimal.

Let n be the number of vertices. The version of the algorithm used to find the MST is $O(n^2)$, and the DFS algorithm to find the Hamiltonian cycle is $O(n \times \log_2(n))$, since now the graph has $n - 1$ edges. The $log_2(n)$ factor arises because of the data structure used in the adjacency list: the class *multiset* in C++, which is a binary tree (the reason to this is because the Christofides algorithm needs to represent the some edges twice). This results in a total of $O(n)$ of time complexity, which can be visualized in the graph below.

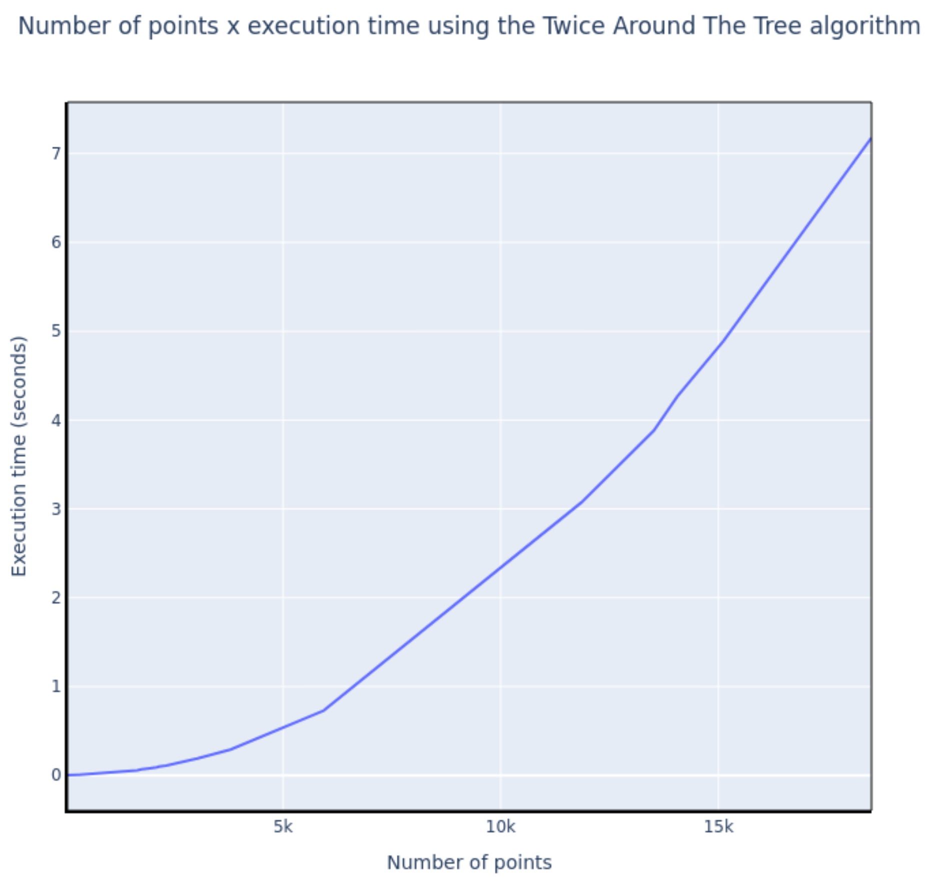

Figure 2. Relation between the number of points and execution time of the TATT

To conclude the analysis, we must check the space complexity: the original graph (the edges between each pair of points) are not created. Instead, we just save the coordinates of each point, which cost $O(n)$ space. The algorithm for generating the MST requires another cost of $O(n)$ space. Since we have $n - 1$ edges in the MST, we also need $O(n)$ space to keep the edges of the MST. To finish it, the DFS have a final $O(n)$ complexity on the worst case. So, this algorithm have a space complexity of $O(n)$.

3.3. Christofides

Let's follow the same order of the analysis as for the TATT algorithm: starting with the quality of the answer. We can observe one advantage of this algorithm in the next graph:

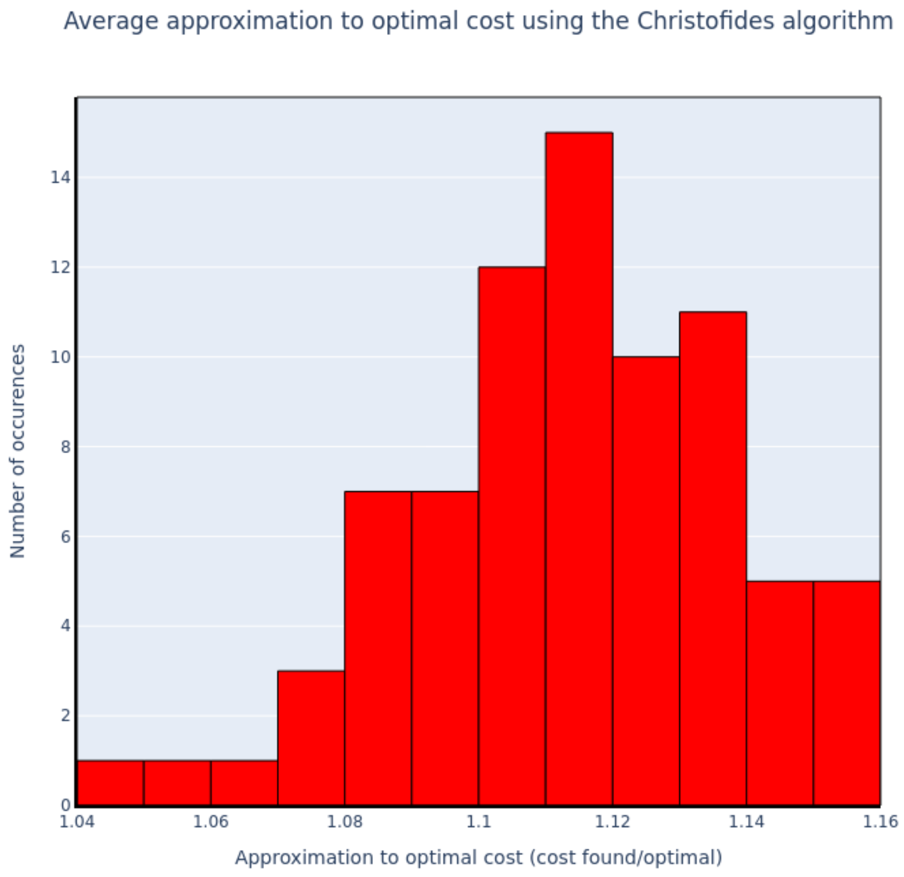

Figure 3. Histogram of approximation rates

As we can see, the Christofides algorithm provides a better answer: it usually results in a solution that is between 1.08 and 1.16 times the optimal. In fact, it has a maximum distance of the optimal of 1.5 times the optimal. It looks pretty good, but there are some disadvantages, that will be discussed below.

The first is time complexity: as in TATT, the algorithm creates a MST, which takes $O(n^2)$. However, the next step is more complex than this. Using the same notations as in subsection 2.3, a minimum perfect matching in G is calculated using the Blossom algorithm, and this has a complexity of $O(|V(G)|^3)$. So, this depends of how many vertices with an odd degree exist in T, but in general this is the longest step of the Christofides algorithm. The next step — find an Eulerian cycle using Hierholzer's algorithm — can be done in $O(|E(H)| \times log_2(|E(H)|))$. We know that $|E(H)|$ is at most $n - 1 + \frac{n}{2}$ ($n - 1$ edges from the MST and $\frac{n}{2}$ edges from M), so we can express the complexiy as $O(n \times log_2(n))$. Finally, the transformation of this cycle into a Hamiltonian cycle is done in $O(|E(H)|) = O(n)$, then the final complexity is $O(n^2 + |V(G)|^3)$.

As we can see in the graphic, the complexity of the algorithm tends to be dominated by the Blossom algorithm (as the factor $O(n^2)$ is also present in TATT algorithm, and it is quite faster than Christofides).

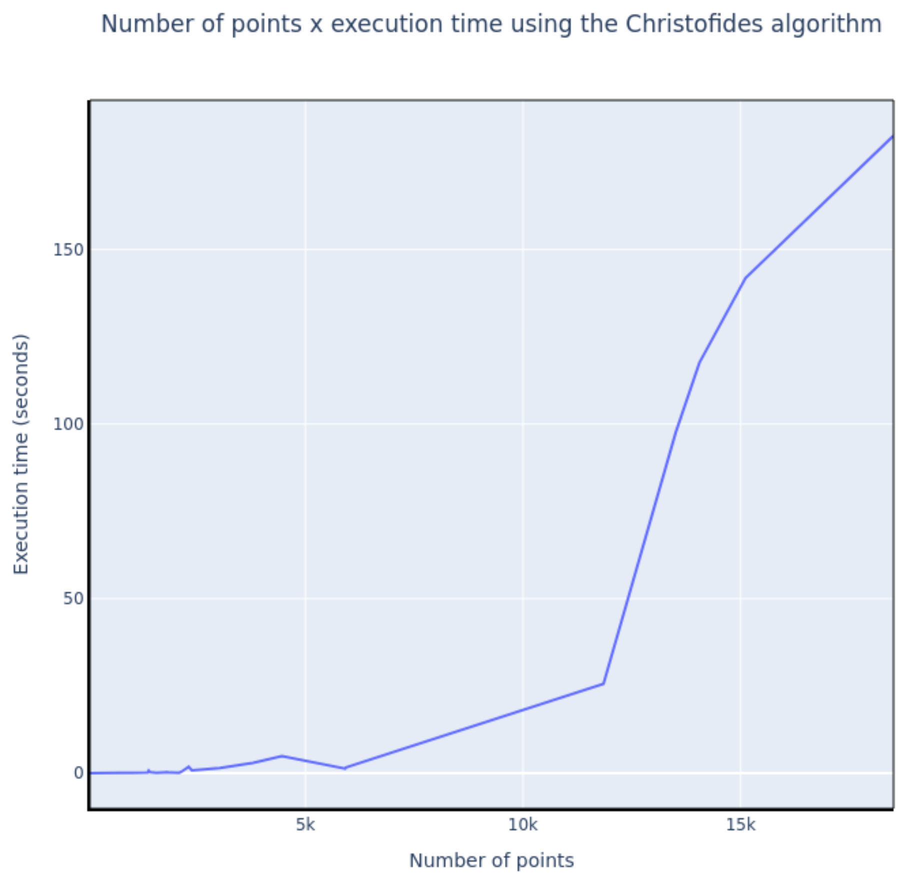

Figure 4. Relation between the number of points and execution time of Christofides algorithm

Another disadvantage of Christofides algorithm is the space complexity. Here, it is really necessary to create the complete induced subgraph of the vertices with an odd degree in the MST. Let m be the number of these vertices. That takes $O(m^2)$ of space complexity. The Blossom algorithm also takes $O(m^2)$. The last part of the algorithm is cheaper: the creation of the graph H requires $O(|E(H)|)$ (therefore, $O(n)$), and the Hierholzer's algorithm needs the same space. So, the total space complexity of the algorithm is $O(n + m^2)$, with the second term, m^2, usually being larger than the n term.

4. Conclusion

As you could see, with a Branch and Bound algorithm, it takes so much time to find the optimal answer, even in small instances. So this algorithm has very specifics applications in real life, given that we can not let a computer calculating the answer for days, or even more than this.

However, Christofides and the TATT algorithms have practical applications. Christofides algorithm gives a better answer, so maybe you is thinking that it is always better to use it instead the TATT algorithm. But the Christofides algorithm is way more expansive than TATT, in space and time.

So, with a large set of points/vertices, maybe it's better use the TATT algorithm in order to get a faster answer (unless you have time to wait, and the memory it takes). But, to small or medium set of points, maybe it is worth it to use the Christofides algorithm, given that the answer is closer to the optimal.

References

[Dezső et al. 2011] Dezső, B., Jüttner, A., and Kovács, P. (2011). Lemon – an open source c++ graph template library. *Electronic Notes in Theoretical Computer Science*, 264(5):23–45.

[Deĭneko et al. 1997] Deĭneko, V. G., Van der Veen, J. A., Rudolf, R., and Woeginger, G. J. (1997). Three easy special cases of the euclidean travelling salesman problem. *RAIRO - Operations Research*, 31(4):343–362.

[Reinelt 1991] Reinelt, G. (1991). Tsplib—a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384.