Travelling Salesperson Problem: Handling an Intractable Problem with Approximations

Juan M. Braga F.[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Av. Pres. Antônio Carlos, 6627 – Pampulha – Belo Horizonte – MG – Brazil

juanbraga@ufmg.br

Abstract. *This paper documents the practical aspects of implementing solutions for the Travelling Salesperson Problem, a well-known difficult problem that requires exponential time for the optimal solution. Three algorithms were implemented: an exact solution through the branch-and-bound technique, and two other approximations, the twice-around-the-tree and Christofides algorithms.*

Resumo. *Este artigo documenta os aspectos práticos da implementação de soluções para o Problema do Caixeiro Viajante, um problema bem conhecido e difícil que requer tempo exponencial para encontrar a solução ótima. Foram implementados três algoritmos: uma solução exata através da técnica de branch-and-bound, e duas outras aproximações, os algoritmos twice-around-the-tree e Christofides.*

1. Introducing the Travelling Salesperson Problem

The Travelling Salesperson Problem (TSP) is aptly described by [Chumbley et al. 2013]

> ”A salesperson needs to visit a set of cities to sell their goods. They know how many cities they need to go to and the distances between each city. In what order should the salesperson visit each city exactly once so that they minimize their travel time and so that they end their journey in their city of origin?”

The TSP is a well-known **intractable problem**, which in the context of algorithms means that the time and/or space required to solve the problem grows exponentially with the size of the input, making it impractical to solve even for relatively small inputs.

This project documents the development of three solutions for the given problem, focusing on evaluating the real-world challenges associated with them, such as planning and optimization, making informed decisions on data structures and libraries, and analysing the usage of computer resources.

2. Algorithms

The TSP problem can be conveniently modelled by a weighted graph, with the graph's nodes representing the cities and the edge weights specifying the distances.

To solve it, three algorithms were implemented in this project: an optimal solution using the branch-and-bound technique, and two approximations, the twice-around-the-tree and the Christofides algorithms.

2.1. Optimal Solution

Branch and bound is a general method that tries to divide a problem into smaller subproblems, and then makes informed decisions to discard some of these subproblems to reach the optimal solution.

To consider it in the TSP context, one benefits from **picturing the problem as a tree**, where each node represents a city and each edge a path between two cities. Then, the first node can be any city (since one can start from any city of the optimal path without changing the path distance), and the path from the origin node to a leaf in the graph should represent the total length of that path.

However, simply storing the entire tree will evidently yield an enormous amount of data, as all possible combinations would end up stored in memory. That is why the branch and bound technique must make the decision of which city to visit next based on all of the previous decisions, which can be called a **lower bound**.

The lower bound parameter for selecting the next city is calculated by summing the distance of the current path with the minimum distance to any unvisited city. By maintaining this metric for each partial path, the branch and bound algorithm discards subproblems that cannot possibly lead to the optimal solution, effectively focusing on the most promising paths.

Additionally, there can be made a choice of which method to traverse the "space" of possibilities: **best-first-search** or **depth-first-search**. The *best* variety means that every neighbouring city will have its path calculated before a choice is made, which will allow the algorithm to make better-informed decisions to discard the branches, but will require a more memory. And secondly, the *depth* variety of traversal is the standart search that goes deeper in the tree, which will require less space but may spend more time traversing the branches. As this is an expensive approach, the DFS was chosen.

The branch-and-bound technique has a **Worst-case** time complexity of $O(n!)$, where n is the number of cities, because it may need to explore all possible permutations. However, effective bounding can significantly reduce the number of examined permutations, making it more efficient than a naive brute-force approach. The actual performance depends on the specific instance of the problem and the bounding technique used.

2.2. Approximate Solution: Christofides

The Christofides algorithm was built by leveraging a key insight: many of the edges on the optimal solution are often found on the Minimum Spanning Tree (MST) of the same graph [1]. Therefore, as there are known relatively efficient algorithms for generating the MST, it makes sense to try and make use of this fact.

To form an optimal solution, all nodes of the MST should have an even number of edges, so that the salesperson could enter and leave the city through exactly two edges (also known as an *Eulerian Tour*). However, most MSTs have nodes with an odd number of edges, which will require more edges to be added.

The second part of the solution involves finding good edge candidates to append to te odd edges. Out of multiple available options, the approach used in this implemen-

[1] [Reducible]

tation was the minimmn-length matching of the odd-degree nodes, which tries to find a maximum matching in a bipartite graph with minimum total weight. In other words, it tries to find the best nodes (shortest distance) that connect every pair of the odd-degree nodes of the MST.

Combining the MST with the minimum-length matching will provide us with a very good approximation of the TSP's optimal solution, which can be easily constructed by traversing this graph while taking shortcuts to avoid visiting multiple cities.

According to [Johnson and McGeoch 2003], the Christofides algorithm provides a better **worst-case** guarantee than any other currently known tour construction heuristic: a worst case ratio of 1.5; [and] it also tends to find better tours in practice. However, computing the minimum-length matching on the odd-degree vertices remains the bottleneck of the algorithm.

2.3. Approximate Solution: Twice Around the Tree

The Twice-around-the-tree also uses the MST to generate a solution. The idea is to generate a path by traversing this MST twice, once in each direction (which means cities are allowed to be visited more than once). This can be done by a Depth-First Search (DFS) traversal.

To improve the quality of the solution, the TAT algorithm then tries to shortcut the path by adding extra edges to it. These edges are chosen in such a way as to reduce the overall path length. The algorithm continues to add edges until the path length does not change significantly.

This algorithm has a **worst-case** guarantee of 2 (also called *2-approximate*), meaning that the total cost of the path found is at most twice the cost of the MST.

One can implement an optimisation and reach an algorithm similar to the *Approx-TSP-Tour* described by [Cormen et al. 2009], and this approach has polynomial time: its complexity is $number\_of\_cities^2$.

3. Implementation Choices and Execution Instructions

The programming language used was Python 3.12, and the program was executed in a Python Virtual Environment on the Ubuntu 23.10 Linux distribution, on a computer with a Intel Core i5-4690K processor and 8GB of RAM at 2133MHz.

The **datasets** were selected by the professor from the TSPLIB [Reinelt 2013]. They are composed of `.tsp` files with the node key and its x and y location in the cartesian plane. The graphs are to be constructed by considering that all cities are connected to all other cities, forming a totally connected graph.

The chosen **data structure** for the approximate algorithms was the graph implementation by the `networkx` library in python [Developers 2023], as recommended by the professor in the project specifications. To prevent excessive use of Python loops, the internal `networkx` functions were also used, such as `minimum_spanning_tree` , `eulerian_circuit` and `min_weight_matching`.

The **libraries** `time` and `memory_profiler` were chosen to keep track of time and allocated memory, as they provided the most convenient interface. It must be noted

that the library to monitor the memory gives a significant increase in memory use, so the implementation allowed for two executions: one for testing time and performance and another to keep track of the memory.

Execution Instructions:

- Make sure that a compatible Python version and the libraries used (See above) are installed.
- Select the desired datasets to be used in the `dataset_schema.txt` file.
- Make sure the respective dataset `.tsp` files are in the `datasets/` subfolder.
- Choose which algorithm will be run and wether to skip memory monitoring in the `main.py` file.
- Run the code in a terminal with the following command: `python main.py` or `python3 main.py`.
- The results will be located on a file called `statistics.csv`.

4. Experiments

The qualities of each algorithm will be evaluated through four metrics: the time elapsed to find the solution, the amount of memory used and the quality of the solutions found compared to the optimal solution (see Section 3 for information on libraries used).

4.1. Memory Use

The first and most important aspect when dealing with exponential optimisation algorithms such as the TSP is the use of memory. To make informed decisions about all possible combinations, it is often necessary to store information about many or all of the previous decisions, not to mention storing the entire problem.

For this project, the chosen data structure implementation (See Section 3) works by storing the entire graph in memory, so simply storing the dataset is the first memory bottleneck.

For the algorithms themselves, see the allocated memory for each test case in Figure 1.

The graph displays a similar memory footprint for both algorithms. However, to put things in perspective, we will analyse the Figure 2.

The first thing that must be noted it that for each algorithm there was a dataset where the allocated memory surpassed the available RAM capacity (See Section 3 for computer specifications) and storage in the `swap` [2]file was triggered. This can be seen in the dataset `rl5915` execution with the Twice-around-the-tree, in Figure 2 (a). This behaviour is similar in the Christofides algorithm.

Secondly, from the `rl11849` dataset onward the computer did not have enough memory and swap to even build the graph in the executions without memory monitoring (See Figure 2 (a)), causing the computer to crash[3]. As the use of the memory_profiler gives

[2]Swap or Page file is a section of a hard drive used as a temporary location to store information when the computer's RAM (random-access memory) is fully utilized. It has significantly slower speeds when compared with the primary memory.

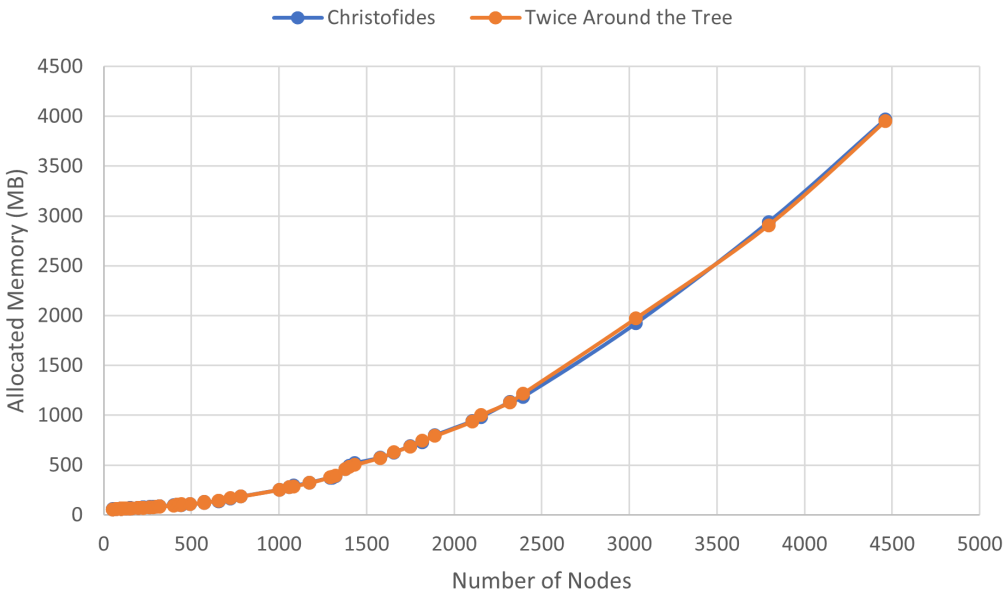

Figure 1. Chart comparing the allocated memory between the Christofides and the Twice-around-the-tree algorithms.

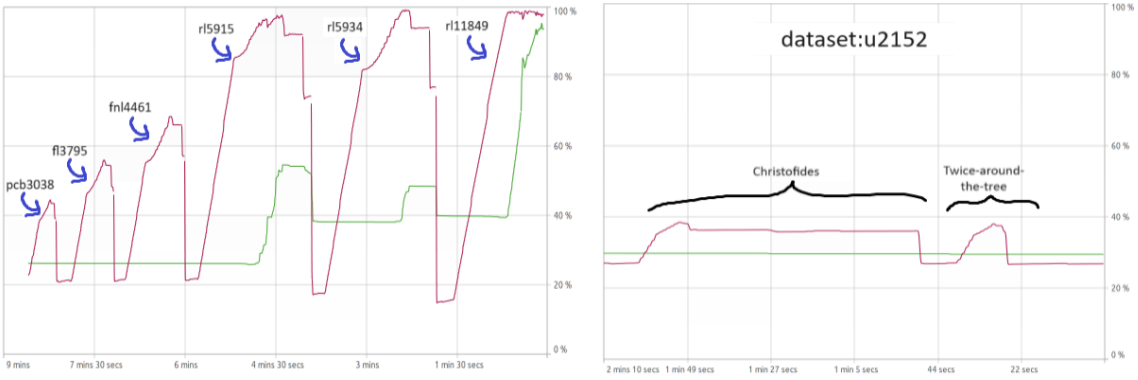

Figure 2. (a) Graph showing memory use for the Twice-around-the-tree algorithm. (b) Graph showing memory use for the Christofides algorithm.

us a significantly larger memory footprint, executing the smaller `rl5915` and `rl5934` datasets with it was also not possible.

The Figure 2 (b) displays how the memory footprint for both algorithms is quite similar (using a small dataset to fit the execution of the Christofides algorithm).

The student was not able to implement a useful branch-and-bound solution to compare with the approximations. Every attempt, either in python or C++, has caused an explosive use of memory, crashing the computer even for the smallest dataset. The student is forced to conclude that trying to solve this problem with this approach is not practical in the absolute sense, although there may be trickier and more efficient ways of implementing. He is, however, assured that choosing the *Depth-First-Search* was the

[3]Even though the computer crashed when reaching maximum RAM + swap usage, the student noticed that running in Microsoft Windows instead of Ubuntu in such circumstances did not cause a crash. Unfortunately, the available machine with Windows was underpowered and would take much longer.

right intuition.

4.2. Solution Quality

The graph of the Figure 3 shows the ratio $\frac{ApproximateSolution}{OptimialSolution}$.

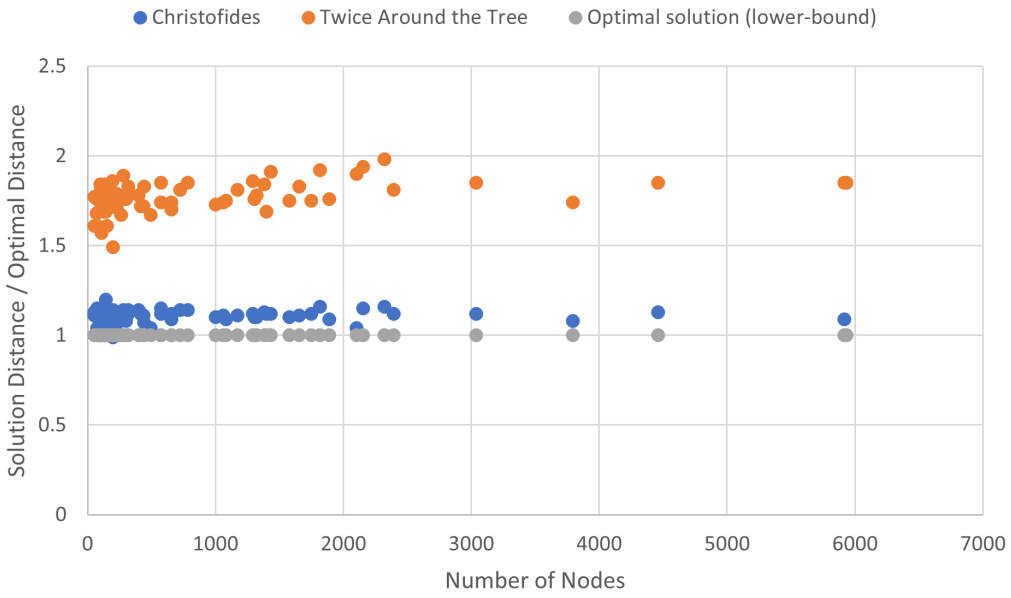

Figure 3. Chart comparing the quality of the approximate and optimal solutions.

The Twice-around-the-tree stays consistently below the expected theoretical worst case (Section 2.3) of twice the optimal solution, and the Christofides solutions find paths that are consistently better and closer to it, also keeping the tight 1.5 quality ratio specified in Section 2.2.

4.3. Execution Time

The execution time of the algorithms can be compared with the following Figure 4.

The Christofides algorithm takes significantly longer to run when compared to the Twice-around-the-tree. In fact, it appears to form an exponential growth, but that must be put into perspective as the student has diagnosed that the algorithms exceeded the available RAM memory (See Section 4.1). This fact can explain why the `fnl4461` has shown such a drastic increase in execution time: the use of swap memory has slowed it down, and from this point onward the graph will not display its ideal behaviour. In other words, this does not constitute as evidence that Christofides algorithm is exponential (See Section 2.2).

This perfectly illustrates how difficult of a problem it is to optimise exponential algorithms, even for relatively small cases. The Twice-around-the-tree, although having a much worse quality then the Christofides, is significantly faster and should provide good-enough answers for the right situations.

5. Conclusions

Implementing the algorithms provided valuable insights into the real-world aspects of solving the TSP. It demonstrated the trade-offs between solution quality, execution time,

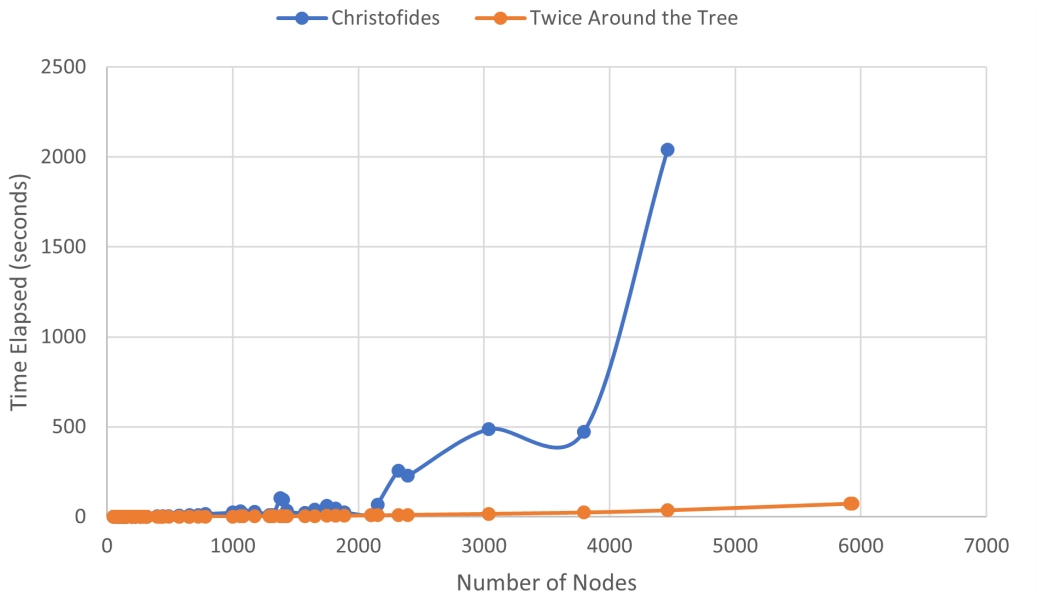

Figure 4. Chart comparing the time elapsed between the Christofides and the Twice-around-the-tree algorithms.

and memory requirements, and highlighted the limitations of exponential algorithms, as the branch-and-bound technique can quickly become intractable for even relatively small instances of the problem.

For the approximations, the Cristofides algorithm consistently outperforms the twice-around-the-tree algorithm in terms of solution quality, maintaining a tight 1.5 quality ratio with the optimal solution. However, it is significantly slower. The twice-around-the-tree algorithm, on the other hand, offers a reasonable trade-off between solution quality and execution time.

The results suggest that the twice-around-the-tree algorithm is a good choice for practical applications of the TSP, as it provides a reasonable balance between quality and efficiency. The Christofides algorithm can be used when a high-quality solution is required, but it is important to be aware of its computational limitations.

References

Chumbley, A., Moore, K., Wang, T., and Ross, E. (2013). Traveling salesperson problem.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). The traveling-salesman problem. In *Introduction to Algorithms*, chapter 35.2, pages 1111–1117. MIT Press, 3rd edition.

Developers, N. (2023). Networkx documentation. NetworkX official documentation.

Johnson, D. S. and McGeoch, L. A. (2003). *8. The traveling salesman problem: a case study*, page 215–310. Princeton University Press.

Reducible. The traveling salesman problem: When good enough beats perfect.

Reinelt, G. (2013). Tsplib.