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. They were then compared in terms of their solution quality, execution time, and memory usage.*

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. Eles foram então comparados em termos de sua qualidade de solução, tempo de execução e uso de memória.*

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 algorithm complexity 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. One benefits from **picturing the problem as a tree**, especially in Section 2.1.

In this project, three algorithms were implemented: an optimal solution using the *Branch and Bound* technique, and two approximations, the *Twice Around the Tree* and the *Christofides* algorithms.

It must be noted that the approximation algorithms rely on the assumption that the instances of the problem obey the **triangle inequality**: in simple terms it says that no single side of a triangle can be greater than the sum of the two others (which is reasonable for most real-world applications).

2.1. Optimal Solution: Branch and Bound

As a hard problem, the most straightforward approach is the *brute force* method, which involves generating all possible permutations of the cities and calculating the total distance for each permutation. However, evaluating the permutations quickly becomes impractical as the number of cities increases, and this may just be the only way to look for the optimal solution on similar problems.

To overcome this limitation, the *Branch and Bound* technique provides a more efficient alternative. Instead of exploring all possible permutations, it divides the problem into smaller subproblems and makes informed decisions to discard some of these subproblems, effectively pruning the ”search space”.

Simply storing all paths would evidently yield a lot of data, as all possible combinations would end up stored in memory. That is why the algorithm must make smart decisions when choosing which city to visit next, which is called a **lower bound**.

This lower bound parameter is calculated by summing the total distance of the previously calculated partial paths and checking which new path to unvisited cities would be the shortest. By maintaining this metric, the *Branch and Bound* algorithm can discard subproblems that will not lead to the optimal solution, and focus 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

[1] See [Reducible 2021] for an intuitive and visual explanation.

MST (such as *Prim's* and *Kruskal's* algorithms, presented previously during DCC206 classes), 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 these odd edges from the MST. Out of multiple available options, the approach used in this implementation was the minimmn-length matching of the odd-degree nodes, which tries to find a maximum matching in a *bipartite graph*[2]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, which means that the total cost of the path found is at most twice the cost of the optimal solution; [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 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 optimal solution.

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, Dataset 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.

[2]A graph is bipartite if its vertices can be divided into two groups such that no two edges connect vertices from the same group.

The **datasets** were selected by the professor from the TSPLIB 95 [Reinelt 2013]. They are composed of a sequence of 78 instances that gradually increase the number of nodes/cities, from 52 and up to 18512. They are stored in `.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. All instances are considered to respect the `triangle inequality` rule (see Section 2 for more information).

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`[3]were chosen to keep track of time and allocated memory, as they provided the most convenient interface. As mentioned before, the `networkx` library was also used.

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_recipe.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. (Look for the `run_algorithm_name` boolean variables in the `main` function.)
- Run the code in a terminal with the following command: `python main.py` or `python3 main.py`.
- The results are stored as a `.csv` file within the subfolder `results`. Each result file will have a timestamp for convenience.

4. Experiments and Discussion

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 the dataset and libraries used).

4.1. Memory Use

The first and most important aspect when dealing with exponential optimisation problems 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 (Section 3) works by storing the entire graph in memory, so simply storing the dataset is the first memory bottleneck.

[3]The library to monitor memory significantly increases memory use, so the program can be run to just keep track of memory and also to collect just the other metrics.

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

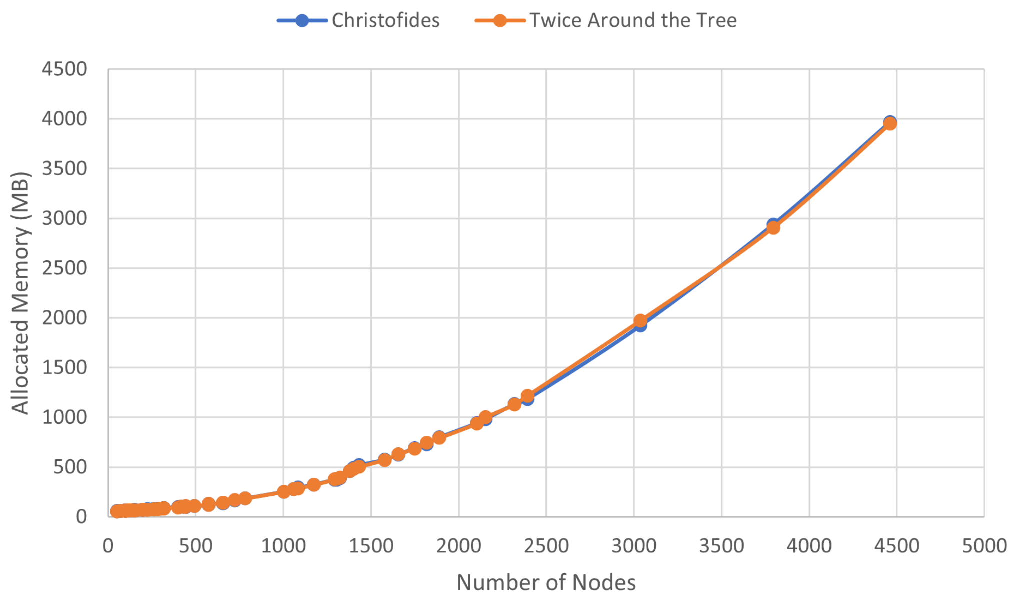

Figure 1. Comparison of the maximum allocated memory by the *Christofides* and the *Twice Around the Tree* algorithms.

This first graph displays a similar memory footprint for both algorithms. However, to put things in perspective, we should take a look at Figure 2.

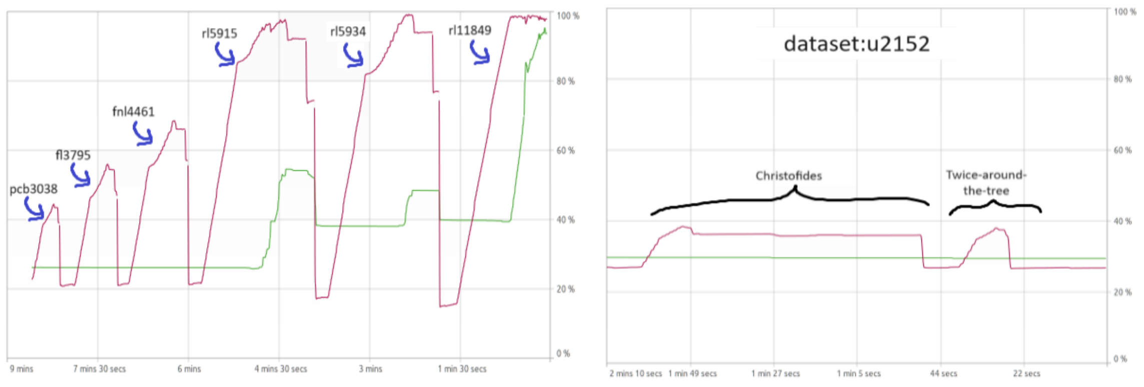

Figure 2. (a) Memory footprint of the system during the execution of the *Twice Around the Tree* algorithm on problems with increasing size. (b) Memory footprint for both *Christofides* and *Twice Around the Tree* algorithms, for the same instance.

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, and storage in the `swap`[4]file was triggered. This can be seen starting from the the dataset `rl5915` execution with the *Twice Around the Tree*, in Figure 2 (a). This behaviour is similar in the *Christofides* algorithm.

[4]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 to the RAM.

Secondly, from the `rl11849` dataset onward the computer did not have enough memory and swap to even build the graph in the executions (See Figure 2 (a)), causing the computer to crash[5]. (As the use of the *memory_profiler* gives us a significantly larger memory footprint, executing the smaller `rl5915` and `rl5934` datasets with it was also not possible.)

The Figure 2 (b) also corroborates the idea of how their memory footprint is similar (using a small dataset to fit the execution time of the *Christofides* algorithm).

Lastly, 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 a somewhat-viable solution. He is, however, assured that choosing the *Depth-First-Search* was the right intuition.

4.2. Solution Quality

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

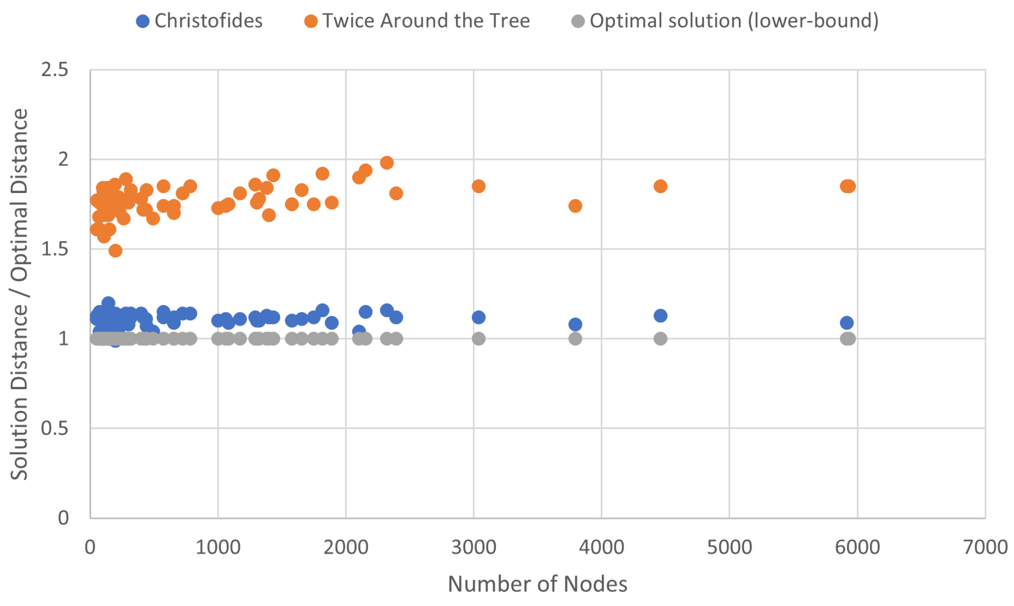

Figure 3. Chart comparing the quality of the approximate and optimal solutions.

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.

[5]Even though the computer crashed when reaching maximum RAM + swap usage, the student noticed that running the same instances in Microsoft Windows did not cause a crash. Sadly, the available Windows machine was underpowered and so this option was ignored.

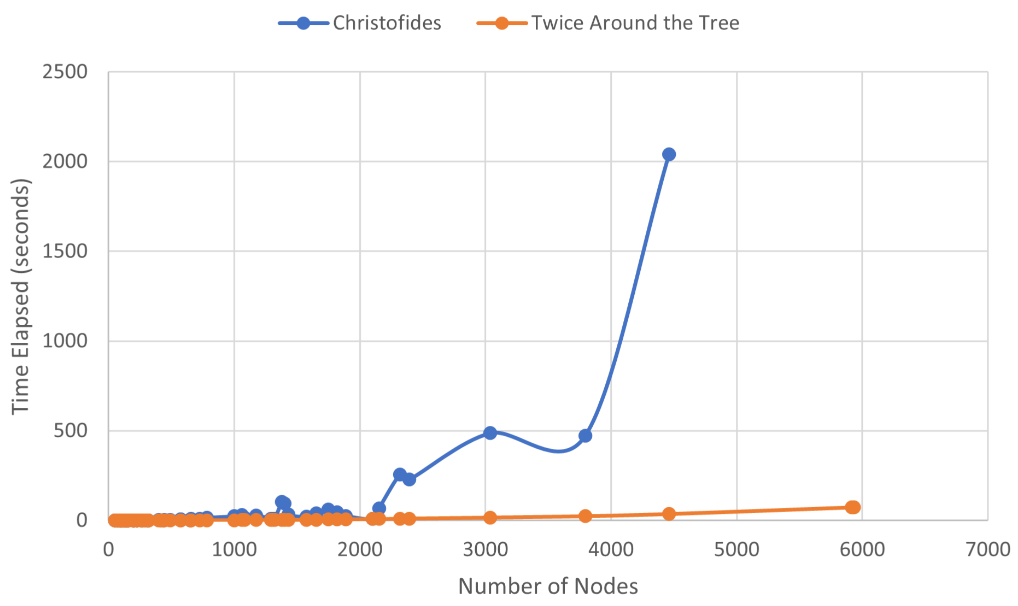

Figure 4. Chart comparing the time elapsed between the *Christofides* and the *Twice Around the Tree* algorithms.

The *Christofides* algorithm takes significantly longer to run when compared to the *Twice Around the Tree* algorithm. In fact, it appears to form an exponential growth. However, as diagnosed in Section 4.1, the algorithms exceeded the available RAM memory, so this claim must be put into perspective. This fact can explain why the `fnl4461` case 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 the *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 and likely other hard problems. It demonstrated the trade-offs between solution quality, execution time, and memory requirements, and highlighted the limitations of exponential algorithms, as the *Branch and Bound* technique quickly becomes intractable for even relatively small instances of the problem.

For the approximations, both had very similar memory requirements, but the Cristofides algorithm consistently outperformed 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

Gerhard Reinelt. (2013). TSPLIB. Retrieved from http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ (Accessed: 2023-12-08).

Chumbley, A., Moore, K., Wang, T., & Ross, E. (2013). Traveling Salesperson Problem. Retrieved from https://brilliant.org/wiki/traveling-salesperson-problem/ (Accessed: 2023-12-08).

Johnson, D. S., & McGeoch, L. A. (2003). 8. The traveling salesman problem: a case study. In Local Search in Combinatorial Optimization (pp. 215–310). Princeton University Press. DOI: 10.1515/9780691187563-011 (Also available at: https://www.cs.ubc.ca/~hutter/previous-earg/EmpAlgReadingGroup/TSP-JohMcg97.pdf).

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). The traveling-salesman problem. In Introduction to Algorithms (3rd ed., pp. 1111-1117). MIT Press. ISBN: 978-0262533058.

Luciana P. Nedel, Rafael H. Bordini, Flávio Rech Wagner, Jomi F. Hübner. (2013). Instructions for Authors of SBC Conferences. Retrieved from https://www.overleaf.com/latex/templates/sbc-conferences-template/blbxwjwzdngr (Accessed: 2023-12-08).

Reducible. (2021). The Traveling Salesman Problem: When Good Enough Beats Perfect. Retrieved from https://www.youtube.com/watch?v=GiDsjIBOVoA (Accessed: 2023-12-09).

SaucelessGiuseppe. (2021). Christofide’s Algorithm. Retrieved from https://www.youtube.com/watch?v=dNCwtFJLsKI (Accessed: 2023-12-09).

NetworkX Developers. (2023). NetworkX Documentation. Retrieved from https://networkx.github.io/documentation/latest/index.html (Accessed: 2023-12-07).

Almeida, Jussara M. (2022). Graphs: Minimum Spanning Tree. PowerPoint slides presented in DCC206 - Algoritmos 1, Universidade Federal de Minas Gerais.