Algorithms for the Traveling Salesman Problem

Raphael Alves dos Reis

[1]Department of Computer Science
Federal University of Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

`cap497@ufmg.br`

Abstract. *This paper addresses the Traveling Salesman Problem (TSP) by implementing and analyzing four algorithms: two exact (Brute Force and Branch and Bound) and two approximate (Twice-Around-the-Tree and Christofides'). The approximate algorithms aim to provide near-optimal solutions efficiently, while the exact ones guarantee optimal solutions. We evaluate the performance of these methods using execution time, memory usage, and result quality as metrics. The results highlight trade-offs between efficiency and accuracy, demonstrating the suitability of each approach for different instance sizes.*

1. Introduction

The Traveling Salesman Problem (TSP) is a classical combinatorial optimization problem that involves finding the shortest route that visits a set of cities exactly once and returns to the starting city. TSP has applications in logistics, manufacturing, and computer science. Due to its NP-hard nature, solving TSP optimally for large instances is computationally infeasible, necessitating the use of both exact and approximation algorithms to understand its complexity and feasibility. This paper presents implementations and analyses of four algorithms: Brute Force, Branch and Bound, Twice-Around-the-Tree, and Christofides'.

2. Implementations

2.1. Data Structures

A node structure is used to represent a tree node, storing its level, path and bound. It also has a comparison method to prioritize nodes with smaller bounds. For cost estimation, distances between points were precomputed and stored in a matrix to allow fast lookup.

2.2. Algorithms

Algorithm 1 Brute Force

1: Compute distance matrix.
2: Evaluate all permutations.
3: Calculate cost for each path.
4: Track minimum cost and best path.
5: Return min_cost, best_path, timeout_exceeded.

Brute Force (BF) has a time complexity of $O(n!)$ since it evaluates all permutations. Its space complexity is $O(n^2 + n!)$ because it stores the distance matrix and permutations. It is highly accurate but impractical beyond small instances.

Algorithm 2 Branch and Bound
___
1: Compute distance matrix.
2: Initialize stack with root node.
3: best_cost ← ∞, best_path ← ∅.
4: **while** stack is not empty **do**
5:     Pop node from stack.
6:     If time limit exceeded, return.
7:     Calculate bounds and prune suboptimal paths.
8:     Expand promising nodes.
9: **end while**
10: Return best_cost, best_path, timeout_exceeded.
___

Branch and Bound (BB) uses a Depth-First Search (DFS) approach to solve Traveling Salesman Problem (TSP) instances by systematically exploring all possible paths while pruning suboptimal ones based on calculated bounds. Its time complexity is $O(n!)$ due to the need to evaluate all permutations in the worst case, and its space complexity is $O(n^2 + n!)$ to store the distance matrix and recursion stack. The DFS strategy helps prioritize deeper exploration of paths, enabling efficient pruning, and guarantees optimality, though it becomes impractical for larger instances due to exponential growth.

Algorithm 3 Twice Around the Tree
___
1: Construct MST.
2: Perform DFS traversal.
3: Compute cost by visiting nodes twice.
4: Return cost and path.
___

Twice-Around-the-Tree (TAT) has a time complexity of $O(n^2)$ due to MST construction and traversal. Its space complexity is $O(n^2)$ for storing graph structures. It provides approximate solutions efficiently for larger instances.

Algorithm 4 Christofides
___
1: Construct MST.
2: Find odd-degree vertices.
3: Compute minimum matching.
4: Form Eulerian circuit.
5: Shortcut repeated nodes.
6: Return cost and path.
___

Christofides' Algorithm (CA) has a time complexity of $O(n^3)$ due to matching steps and a space complexity of $O(n^2)$. It guarantees solutions within 1.5 times the optimal and balances efficiency with accuracy.

2.3. Algorithms Complexity

Algorithm	Time	Space
Brute Force	$O(n!)$	$O(n^2 + n!)$
Branch and Bound	$O(n!)$	$O(n^2 + n!)$
Twice-Around-the-Tree	$O(n^2)$	$O(n^2)$
Christofides'	$O(n^3)$	$O(n^2)$

Table 1. Complexity Comparison of Algorithms

3. Experiments

3.1. Functions

Following functions compose this work:

Function 1 `load_coordinates`

Require: File with node coordinates in TSPLIB format.
Ensure: Normalized coordinates within a 100x100 square.
 1: Open input file and read all lines.
 2: Initialize an empty list for coordinates.
 3: **for** each line in the file **do**
 4:    Parse index and coordinates (x, y).
 5: **end for**
 6: Normalize x and y values to the range [0, 100].
 7: **return** List of normalized coordinates.

Function 2 `plot_path`

Require: Node coordinates, path, and output type.
Ensure: Plots nodes and connections with execution metrics.
 1: Plot nodes as orange points.
 2: **if** path is final solution **then**
 3:    Draw connections in blue.
 4: **else**
 5:    Draw connections in red.
 6: **end if**
 7: Display cost, execution time, and memory usage.

Function 3 `generate_tsp_instance`

Require: Number of nodes n.
Ensure: Randomly generated TSP instance with n nodes.
 1: Initialize an empty list for nodes.
 2: **for** $i = 1$ to n **do**
 3:    Generate random coordinates (x, y).
 4:    Append node with index and coordinates.
 5: **end for**
 6: **return** List of nodes.

Function 4 `generate_tsp_instances`

Require: Range of nodes, step size, and number of instances per size.
Ensure: Saves generated TSP instances.
 1: **for** each size in range with step size **do**
 2:    **for** each instance count **do**
 3:       Generate TSP instance using `generate_tsp_instance`.
 4:       Save instance to file.
 5:    **end for**
 6: **end for**

Function 5 `warm_up_algorithms`

Require: None.

Ensure: Dummy data fills memory to stabilize measurements.

  1: Generate a small dummy dataset.
  2: Execute simple operations to pre-load memory.
  3: **return** Initialized memory state.

Function 6 `process_files`

Require: List of input files.

Ensure: Sort files, process algorithms, and log results.

  1: Sort files by number of nodes.
  2: **for** each file **do**
  3:    Load instance data.
  4:    Execute algorithms and track metrics.
  5:    Plot maps using `plot_path`.
  6:    Write results to CSV file.
  7: **end for**

Function 7 `plot_metrics`

Require: Metrics data for cost, execution time, and memory usage.

Ensure: Plot and save performance metrics.

  1: **for** each algorithm and instance class **do**
  2:    Compute mean values for metrics.
  3:    Generate plots for each metric.
  4:    Save plots to output directory.
  5: **end for**

3.2. Data

3.2.1. Classification

Two types of data were used: randomly generated coordinates for small instances and instances from TSPLIB95 for larger ones. TSPLIB95 is a well-known library containing samples for TSP and related problems derived from various sources. To assess algorithm performance under increasing problem sizes, data were divided into four classes—small, medium, large, and huge—based on node count.

3.2.2. Selection Criteria

Since the TSPLIB95 library lacks sufficient small and medium-sized instances, random coordinates were generated to analyze performance in these categories. Larger instances were selected directly from the library to represent more realistic and computationally challenging scenarios. Three filtering steps were applied:

  1. Only TSP instances were considered.
  2. Instances were restricted to 2D Euclidean coordinates.
  3. Focus was placed on Padberg and Rinaldi's N-city problems (e.g., prN).

3.2.3. Experimental Setup

For small, medium, and large classes, three instances were generated per node size to compute average performance metrics. The testing approach followed these rules:

- **Small** (5–10 nodes, step 1): All four algorithms—BF, BB, TAT, and CA—were tested.
- **Medium** (12–30 nodes, step 2): BF was excluded due to time infeasibility; remaining algorithms tested were BB, TAT, and CA.
- **Large** (30–100 nodes, step 10): Only TAT and CA were tested, as BF and BB were infeasible.
- **Huge** (76–2392 nodes, TSPLIB): Only TAT and CA were used. Instances larger than 2393 nodes were excluded due to CA memory limitations.

3.2.4. Conclusion

Despite TSPLIB95 containing mostly instances infeasible for exact algorithms, it provided diverse examples to assess algorithm scalability. This design ensured a balanced evaluation of algorithm performance across different instance sizes, enabling comparisons between exact and heuristic approaches.

3.3. Results

Overall results can be seen in figure 1. Example instances can be seen in figure 2. More instances and metrics graphs for each class can be seen in available notebook. Each metric graph contains four curves, one for each algorithm, showing how far each algorithm reaches: BF stops at small instances; BB at medium instances; and TAT and CA work for large and huge classes. Despite CA being able to solve an instance with a large number of nodes, it still requires an excessive amount of time and hardware when compared to TAT, so this last one being the only feasible algorithm to be used for huge instances such as the ones used in this work.

3.3.1. Small Instances

- **Cost:** TAT produced the highest costs, reflecting its limited optimization ability. CA provided intermediate results, while BF and BB delivered identical, optimal solutions.
- **Execution Time:** BF exhibited exponential growth, becoming infeasible even for small datasets. BB managed time efficiently but still faced scalability challenges. Approximation algorithms performed better in runtime.
- **Memory Usage:** BB consumed the least memory, BF required the most, and approximation methods balanced performance and resource consumption.

3.3.2. Medium Instances

- **Cost:** TAT produced higher costs compared to CA and BB, while adhering to theoretical bounds of 2x and 1.5x the optimal solution. CA provided closer approximations.
- **Execution Time:** Approximation algorithms performed substantially better in runtime compared to exact methods. Exact methods became infeasible beyond 15-20 nodes, with computation times exceeding 10 minutes. Intermediate solutions (red paths) were visualized for exact algorithms during partial progress.
- **Memory Usage:** Memory usage showed erratic behavior for BB, surpassing approximation algorithms as node sizes increased. Approximation methods remained stable, while exact methods became unstable.
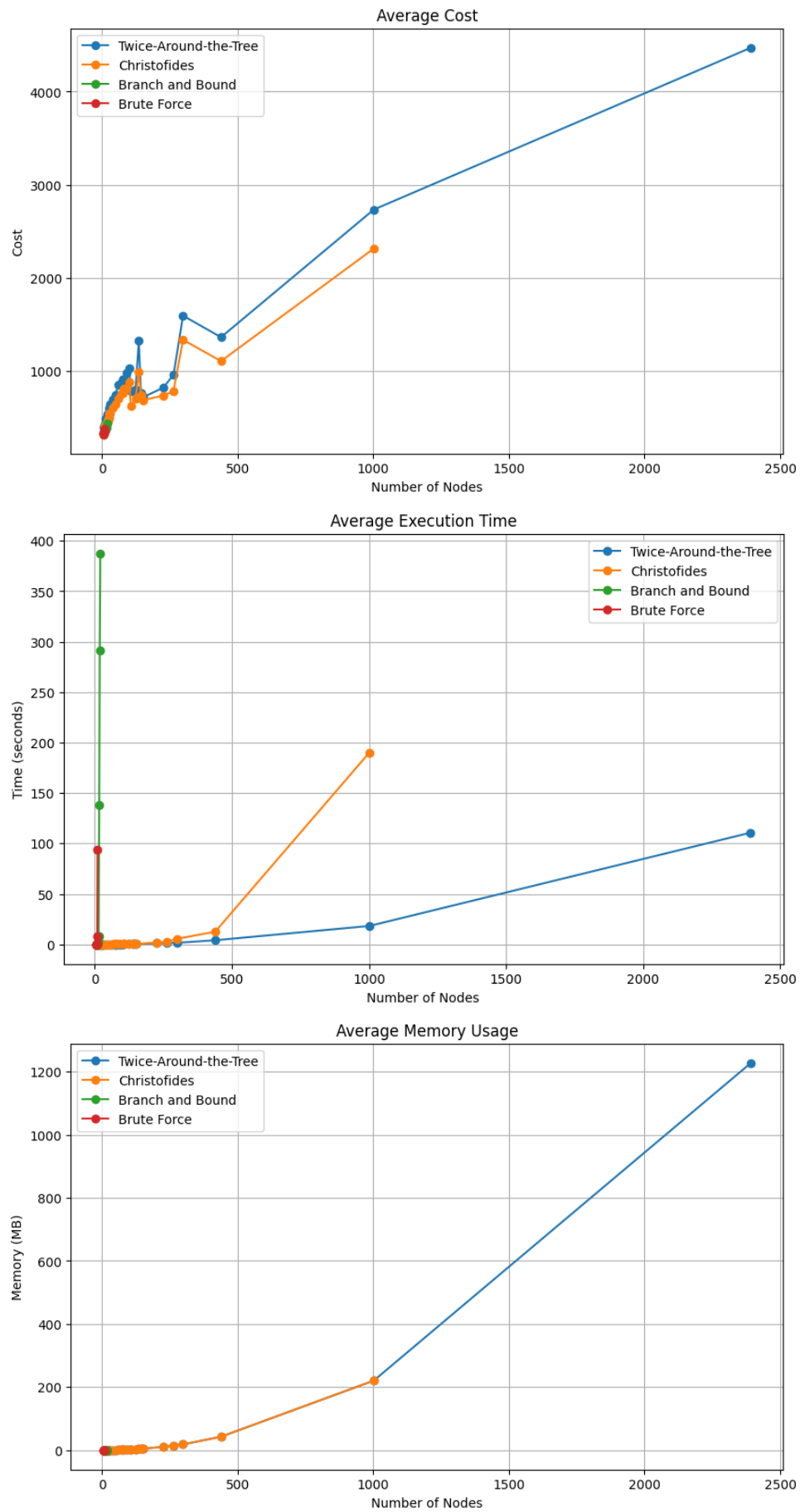
Figure 1. Overall cost, execution time and memory usage results.

3.3.3. Large Instances

- **Cost:** Costs grew steadily, respecting theoretical bounds—2x for TAT and 1.5x for CA.
- **Execution Time:** Execution time showed linear growth for both algorithms, with TAT scaling more efficiently.
- **Memory Usage:** Memory usage increased consistently for both algorithms, with TAT demonstrating slightly lower consumption.

3.3.4. Huge Instances

- **Cost:** Both algorithms produced results within theoretical bounds, with TAT displaying slightly higher costs than CA.
- **Execution Time:** Despite being approximation methods, both algorithms experienced significant runtime growth, and CA became infeasible for very large datasets.
- **Memory Usage:** Memory usage grew significantly for both methods, imposing hardware limitations and leading to the omission of larger datasets from testing.

4. Conclusion

The analysis demonstrates that approximation algorithms, such as Twice-Around-The-Tree (TAT) and Christofides', effectively handle larger datasets while maintaining costs within theoretical bounds of 2x and 1.5x the optimal solution, respectively. Exact methods, including Brute Force and Branch-and-Bound, provide optimal results but quickly become infeasible due to exponential growth in execution time and memory usage as problem size increases. Christofides' consistently outperforms TAT in cost minimization, though both show similar trends in execution time and memory requirements, with TAT exhibiting slightly better scalability. Approximation algorithms remain more practical for larger instances, as exact methods are constrained by computational and hardware limitations. However, even approximations face growing resource demands, making extremely large datasets difficult to handle without significant computational infrastructure.
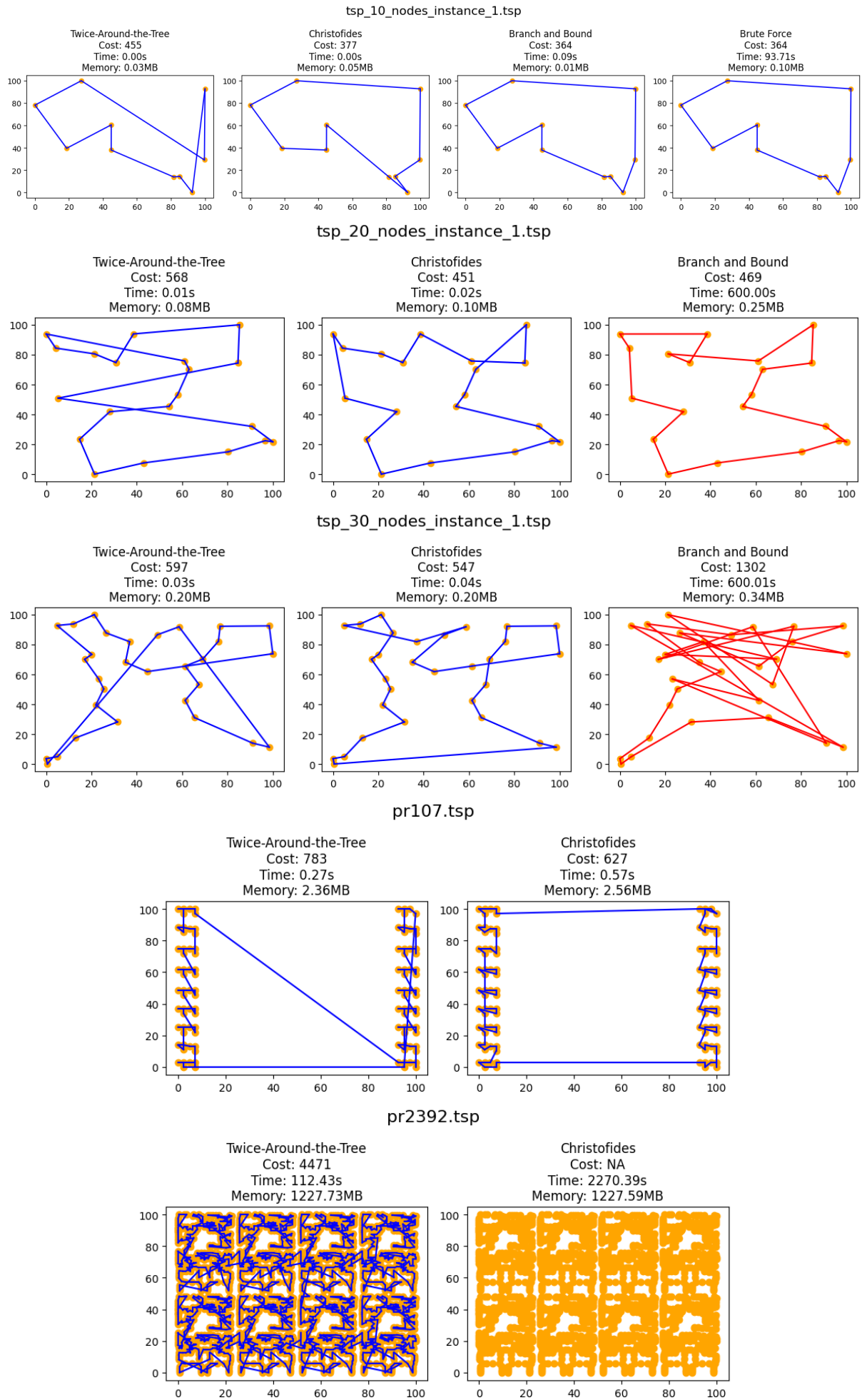
Figure 2. Instances of 10, 20, 30, 107 and 2392 nodes, with its respective metrics.