CS-584-002 Winter 2024
Algorithm Design and analysis

A Comparative Analysis of TSP Algorithms for Travel Applications

Chethana Muppalam

Abstract

The Traveling Salesman Problem is one of the most prominent problems in combinatorial optimization and is regularly employed in a wide variety of applications. This paper presents a comparative analysis of routing algorithms applicable to travel applications, specifically addressing the Traveling Salesman Problem (TSP) in the context of optimizing curated itineraries. The objective of this paper is to demonstrate the extent of sub-optimality resulting from Traveling Salesman solution procedures when implemented in the design of Travel Applications and to discuss the practical consequences. The TSP algorithms experimented are Minimum Spanning Tree, Nearest Neighbor, Greedy, Christofides, Simulated Annealing and Threshold Accepting. The experiments aim to provide insights into algorithm performance and suitability for real-world applications of travel planning.

1. Introduction

1.1 Problem Statement

Traveling the open road is a quintessential adventure, yet planning an itinerary that balances exciting destinations, manageable travel times, and avoids congested routes can be overwhelming. As a passionate explorer, I often experience this challenge. This research project aims to tackle this road trip planning dilemma. By evaluating and comparing existing routing algorithms used in travel apps, This experimentation will identify the most efficient solutions for navigating between a user-defined list of destinations. This will empower travelers to create itineraries that maximize their enjoyment and minimize travel hassles.

The Traveling Salesman Problem has become so fundamental in modern computing that it is commonly considered the prototypical NP-Hard combinatorial optimization problem, possessing a far-reaching impact on countless applications in science and industry. NP-Hard problems are those for which no known polynomial-time algorithm exists to solve them optimally, meaning that as the problem size increases, the computational resources required to solve them grow exponentially. This means there's no known efficient algorithm guaranteed to find the optimal solution for all problem sizes. However, several approximation algorithms provide good solutions in practical timeframes. A breakdown of some popular approximation algorithms for the problem statement are discussed and put under experimentation in the upcoming sections.

2. Algorithms Explored

2.1. Nearest Neighbor (NN) Algorithm: This algorithm starts at one city and then goes to the nearest unvisited cities until all are visited. It's like choosing the closest neighbor to visit next. Once all the cities are visited, It then returns to the starting city.

- **Approximation Ratio:** The NN algorithm guarantees a worst-case approximation ratio of 2 (twice the optimal tour length).

```
function nearest_neighbor_tour(graph):
    starting_node = randomly_choose_starting_node(graph)
    path = [starting_node]
    visited = {starting_node}
    while len(visited) < number_of_nodes(graph):
        current_node = path[-1]
        min_weight = infinity
        min_neighbor = None
        for neighbor in neighbors(current_node, graph):
            if neighbor not in visited and weight(current_node, neighbor,
graph) < min_weight:
                min_weight = weight(current_node, neighbor, graph)
                min_neighbor = neighbor
        if min_neighbor is not None:
            path.append(min_neighbor)
            visited.add(min_neighbor)
        else:
            break
    path.append(starting_node)
    return path
```

2.2. Greedy Algorithm: Greedy algorithms make decisions based on the best immediate choice without considering the overall consequences. In the context of travel, it would mean always selecting the next destination that seems the most advantageous at the moment.

A greedy algorithm does not always give the best solution. However, it can construct a first feasible solution which can be passed as a parameter to an iterative improvement algorithm such as Simulated Annealing or Threshold Accepting.

- **Approximation Ratio:** Greedy Algorithm for the TSP doesn't have a constant approximation ratio, but its solutions often fall within a factor of 1.5 to 2 times the optimal tour length. However, this ratio can vary widely depending on the problem instance and the specific heuristics employed within the algorithm.

```
function greedy_tour(graph):
    starting_node = randomly_choose_starting_node(graph)
    path = [starting_node]
    unvisited = all_nodes_except_starting_node(graph, starting_node)
    while not empty(unvisited):
        min_weight = infinity
        min_neighbor = None
        for node in path:
            for neighbor in neighbors(node, graph):
                if neighbor in unvisited and weight(node, neighbor, graph) <
min_weight:
                    min_weight = weight(node, neighbor, graph)
                    min_neighbor = neighbor
        path.append(min_neighbor)
        remove min_neighbor from unvisited
    path.append(starting_node)
    return path
```

2.3. Minimum Spanning Tree (MST) based Heuristics: This algorithm finds the shortest route that connects all destinations without forming any loops. It's like drawing the shortest possible lines to connect all the dots on a map. Various heuristics use a Minimum Spanning Tree (MST) as a building block. Prim's or Kruskal's algorithms can be used to find the MST, and then additional steps are taken to convert it into a valid tour (e.g., doubling the edges in the MST).

- **Approximation Ratio:** The approximation ratio for MST-based heuristics can vary depending on the specific implementation. They often achieve a ratio between 1.5 and 2.

2.4. Christofides Algorithm: This algorithm is a bit more complex. It combines a Minimum Spanning Tree (MST) heuristic with a matching algorithm to construct a tour.It first finds a minimum spanning tree, then adds shortcuts to make the route more efficient, ensuring it doesn't visit any city more than once (unless it's the starting and ending point).It's guaranteed to find a tour no longer than 3/2 times the optimal tour length.

- **Approximation Ratio:** Guaranteed worst-case approximation ratio of 3/2 (one and a half times the optimal tour length). This is the best known **approximation ratio for symmetric TSP** (equal distance between two cities A and B is the same as B and A).

2.5. Simulated Annealing and Threshold Accepting:

These both draw inspiration from metallurgical annealing processes, gradually refining a random solution by decreasing "temperature" (randomness) over time. This method mimics the cooling process of a metal, reducing defects and approaching an optimal. These are metaheuristic local search algorithms that can accept even solutions which lead to the increase of the cost in order to escape from low quality local optimal solutions.

SA: Starting from a suboptimal solution, simulated annealing perturbs that solution, occasionally accepting changes that make the solution worse to escape from a locally optimal solution. The chance of accepting such changes decreases over the iterations to encourage an optimal result.

TA: Starting from a suboptimal solution, threshold accepting methods perturb that solution, accepting any changes that make the solution no worse than increasing by a threshold amount. In comparison to the Simulated Annealing algorithm, the Threshold Accepting algorithm does not accept very low-quality solutions (due to the presence of the threshold value).

```
Function simulated_annealing_tour(graph G):
    current_solution = initial_solution
    temperature = initial_temperature
    while temperature > minimum_temperature:
        new_solution = perturb_solution(current_solution)
        delta_cost = cost(new_solution) - cost(current_solution)
        if delta_cost < 0 or random() < exp(-delta_cost / temperature):
            current_solution = new_solution
        temperature = cool(temperature)
    return current_solution
```

- **Approximation Ratio:** No guaranteed approximation ratio, but they can often find near-optimal solutions for complex TSP instances.

3. Experiments

3.1. Procedure

Source Code Repository : https://github.com/chethana613/analysis_tsp_algorithms

This paper addresses the Symmetric Traveling Salesman Problem (TSP), wherein the distance between any two cities remains the same in both directions. Given a set of n nodes and their coordinates, the experiment finds a roundtrip of minimal total length visiting each node exactly once. Euclidean distance was employed to calculate the distances between nodes based on their coordinates. Hence, the distance from node i to node j is the same as the distance from node j to node i.

The goal was to utilize stable and consistent implementations for all algorithms. Consequently, the NetworkX library (nx.approximation.traveling_salesman) was relied upon for most implementations, and Nearest Neighbor was implemented in-house.

Simulated annealing and threshold_accepting both require an initial solution to improve upon. By default, the SA has “greedy” and TA has “SA” as their initial solution choice respectively. Without these initial solutions, these algorithms performed poorly, with the cost being higher than greedy since both these algorithms encourage the solution to go outside the local maxima and degrade for a time.

Datasets are taken from TSPLIB library:
http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

3.2. Setup

3.2.1. Isolation

The algorithms were executed in sequence, ensuring isolation.

3.2.2. Resource Management

The algorithms were implemented and executed in Python’s runtime. Python's memory management is handled by its built-in garbage collector. It automatically deallocates memory for objects that are no longer in use, preventing memory leaks. Certain objects in Python, such as file objects (open()) which were used in the implementation, support automatic resource cleanup via context managers, making it easier to ensure resource cleanup.

3.2.3. Graph construction

Datasets were parsed to construct graphs in-memory and use them for comparisons. Constructing graphs for large datasets, specifically those exceeding 6k nodes, posed computational challenges in Python running on a machine with 32 GB of memory. Efforts were made to leverage the joblib library in Python for parallel construction, aiming to enhance computational efficiency. However, despite these attempts, no significant performance improvement was observed on the local system.

Initially Euclidean distance of the coordinates were calculated by loops but since this was found to be slow, np library was used to improve the overall setup time.

3.3. Experiments Performed

3.3.1. Performance Evaluation on Synthetic Data

Utilized realworld cities dataset from TSPLIB library with different numbers of destinations and their coordinates. Applied each routing algorithm to these datasets, measured the resulting tour lengths, and compared the results with the known optimal solution.

Reason: Synthetic datasets allow for controlled experimentation and provide insights into algorithm performance under different conditions on real world data. This experiment helps assess how well algorithms scale with increasing problem sizes and real world geographical layouts.

3.3.2 Comparisons of Cost and Execution Time

Measured the tour cost produced by each algorithm and the time taken to compute these solutions. Compared the trade-off between solution quality and computational efficiency.

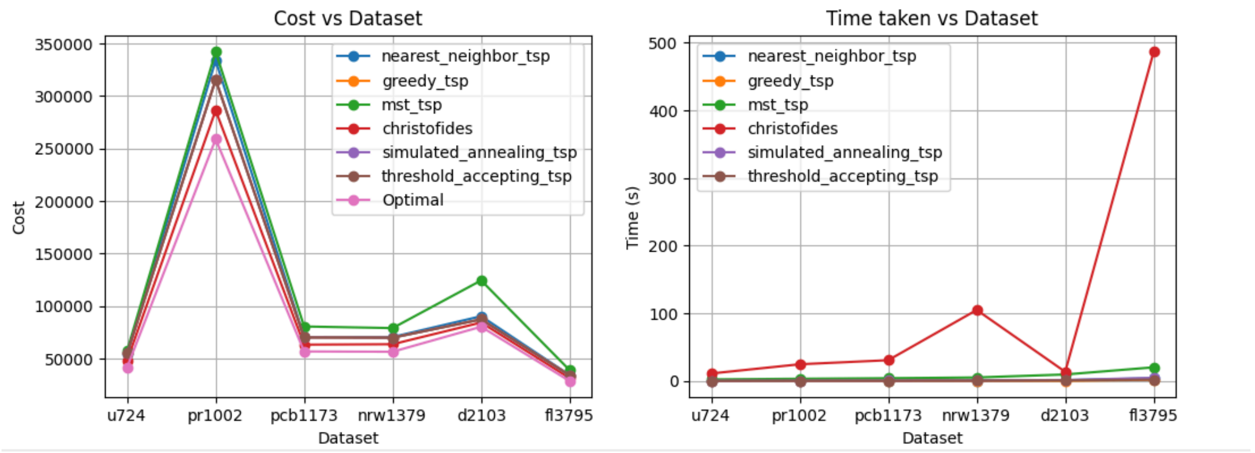
Reason: In practical travel applications, both solution quality and computational efficiency are crucial factors. This experiment helps users understand the trade-offs involved in selecting a particular algorithm based on their priorities.

3.3.2 Comparison with Optimal Solutions

For problem instances where optimal solutions are computationally feasible, compare the solutions produced by approximation algorithms with the optimal solution. In this setting the optimal solutions are compared with all the experimented algorithms.

Reason: While approximation algorithms do not guarantee optimality, comparing their solutions with optimal solutions helps understand the gap in performance and provides context for evaluating their effectiveness.

4. Results and Graphs



Dataset	Optimal	NN		Greedy		MST		Christofides		SA		TA	
	Cost	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time
u**724**	41910	53967.99	0.13s	55223.2	0.06s	57779.67	2.45s	48251.82	11.17s	55223.2	0.40s	55223.2	0.35s
pr**1002**	259045	334063.02	0.25s	315596.59	0.09s	342244.46	3.39s	286424	24.80s	315596.59	0.57s	315596.59	0.49s
pcb**1173**	56892	70608.48	0.31s	70277.94	0.13s	80694.89	4.21s	63391.27	30.81s	70277.94	0.70s	70277.94	0.59s
nrw**1379**	56638	70668.09	0.46s	70015.46	0.25s	79156.27	5.30s	63783.11	105.01s	70015.46	0.95s	70015.46	0.70s
d**2103**	80450	90462.5	1.22s	87468.57	0.41s	124533.87	9.71s	84463.76	13.65s	87468.57	1.48s	87468.57	1.08s
fl**3795**	28772	34388.68	4.23s	34225.56	1.91s	39072.83	20.31s	31537.67	486.57s	34225.56	4.72s	34225.56	2.13s

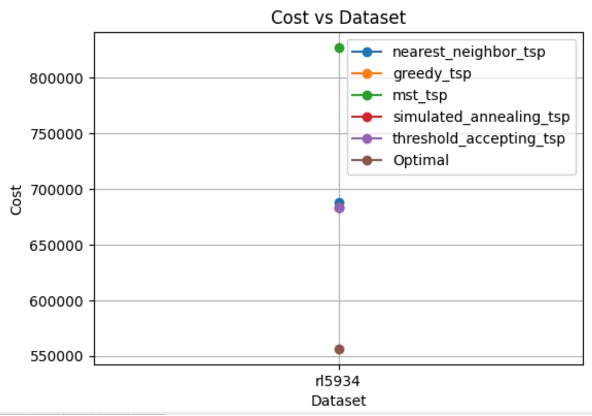
All algorithms closely approximated the optimal solution. MST consistently produced the poorest results across all datasets. NN and Greedy algorithms performed slightly better, with Greedy edging out NN by a mere 1%. SA and TA were the next, and they produced very identical results. Upon experimentation it was noticed that TA failed to converge despite adjusting the threshold and maximum iteration values. Christofides consistently yielded solutions closest to the optimal ones. These results correlate with the theoretical approximation factors.

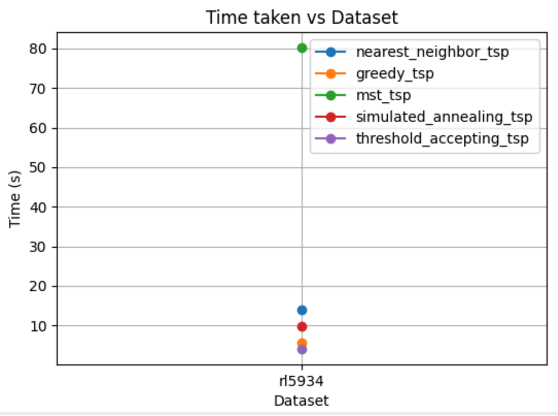
In terms of time complexity, Christofides was the slowest of all, which can be explained by the fact that the algorithm has a complexity of O(N^3 log N). MST was the next slowest due to its time complexity being O(N^2 log N). The execution time differences for the rest of the algorithms were not significant enough to provide any insights, which correlates with all of their time complexities being O(N^2).

Experiments with largest dataset

Below are the results for the largest dataset experimented of ~6k nodes.

Note: This dataset represents the maximum size supported by the experimental system's memory constraints.





Best performing algorithms in the order of Total Cost:

Algorithm	Cost	Approximation Factor
Christofides	607657.04	1.092
Greedy	683806	1.231
Simulated Annealing (SA)	683806	1.231
Threshold Accepting (TA)	683806	1.231
Nearest Neighbor (NN)	688535.68	1.238
Minimum Spanning Tree (MST)	827482.34	1.49

Best performing algorithms in the order of Execution Time:

Algorithm	Time taken	Theoretical TC
Threshold Accepting (TA)	3.96s	O(n^2)
Greedy	5.70s	O(n^2)
Simulated Annealing (SA)	9.85s	O(n^2)
Nearest Neighbor (NN)	13.92s	O(n^2)
Minimum Spanning Tree (MST)	80.27s	O(n^2 log n)
Christofides	406.56s	O(n^3 log n)

Note: Threshold Accepting executed faster than Simulated Annealing because the latter was fed as an input to the former. Therefore, it could be argued that the Greedy algorithm might be the fastest solution of all, Christofides being the slowest.

Based on our experiments, we have determined that for applications with time constraints, any of Greedy, Simulated Annealing, or Threshold Accepting will be practical. For applications that require more accurate results but have access to more computational resources, Christofides should be the obvious choice.

5. Summary

The paper thoroughly compares six common TSP algorithms: Nearest Neighbor, Minimum Spanning Tree-based heuristics, Christofides, Greedy, Simulated Annealing, and Threshold Accepting. It dives into different input sizes and ranges to understand how these algorithms perform. It finds that each algorithm has its unique strengths and weaknesses, depending on the size of nodes. Experiments were conducted on synthetic data from the TSPLIB library, comparing solution quality and execution time across different algorithms. Measures were taken to ensure isolation and resource management during experimentation. Results show that all algorithms closely approximate optimal solutions, with Christofides consistently yielding the best results. Larger datasets revealed Christofides as the best performing algorithm in terms of total cost, while Greedy was the fastest in execution time.

6. References

[1] Comparative study of variations in quantum approximate optimization algorithms for the Traveling Salesman Problem Wenyang Qian,1, 2, ∗ Robert A. M. Basili,3, Mary Eshaghian-Wilner,4, Ashfaq Khokhar,3, Glenn Luecke,4, and James P. Vary2

[2] Comparison of various algorithms based on TSP solving JianChen Zhang 2021 J. Phys.: Conf. Ser. 2083 032007

[3] A Comparative Analysis of Traveling Salesman Solutions from Geographic Information Systems Kevin M. Curtin, Gabriela Voicu, Matthew T. Rice and Anthony Stefanidis* *George Mason University City of McKinney Engineering Department

[4] A Hybrid Optimization Algorithm for Traveling Salesman Problem Based on Geographical Information System for Logistics Distribution Wei Gu1 , Yong Liu2 , Li-Rong Wei3 and Bing-Kun Dong4

[5] Reinelt, G. "TSPLIB--A Traveling Salesman Problem Library." ORSA Journal on Computing, Vol. 3, No. 4, pp. 376-384. Fall 1991

[6] Traveling Salesman Problem: Exact Solutions vs. Heuristic vs. Approximation Algorithms | Baeldung on Computer Science.