A Study of the Travelling Salesman Problem

~ Jeet Shah, Suyah Lal, Archit Murali

The Travelling Salesman problem [also called the travelling salesperson problem or TSP] asks the following question -
"*Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?*"

The TSP is a classic example of a combinatorial optimization problem, where the goal is to find the best possible solution from a finite set of possible solutions. In the TSP, the goal is to find the shortest possible tour that visits every city exactly once. It is an NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research (“NP-Hardness”).

> *In computational complexity theory, NP-hardness [non-deterministic polynomial- time hardness] is the defining property of a class of problems that are informally "at least as hard as the hardest problems in NP."*

A more precise specification: A problem H is NP-hard when every problem L in NP can be reduced in polynomial time to H; that is, assuming a solution for H takes 1 unit time, H's solution can be used to solve L in polynomial time (Leeuwen). As a consequence, finding a polynomial time algorithm to solve any NP-hard problem would give polynomial time algorithms for all the problems in NP. As it is suspected that P ≠ NP, it is unlikely that such an algorithm exists (Bovet and Crescenzi).

To solve the TSP, various algorithms have been developed over the years, including exact algorithms, heuristic algorithms, and metaheuristic algorithms. In our assignment, we will be implementing only the heuristic algorithms.

> *Heuristic algorithms are algorithms that use a practical approach (human approach) to solve faster and can handle larger instances of the problem, but they cannot guarantee the best and correct solution.*

Some real-world application of the TSP are as follows:

- TSP solutions can help improve efficiency in **last-mile delivery logistics**. Last-mile delivery is the final link in a supply chain when goods move from a transportation hub to the end customer. TSP can help optimize the route taken by delivery vehicles to minimize travel time and cost.
- **Vehicle routing problem [VRP]** is a generalized variant of TSP that considers several real-world elements such as multiple agents, limited capacity, working hours, time windows and depots. VRP can be used to optimize routes for multiple vehicles with constraints such as delivery time windows and vehicle capacity.
- **Planning, logistics and the manufacture of microchips** are some other areas where TSP has applications. For example, in microchip manufacturing, TSP can be used to optimize the path taken by a laser or electron beam to minimize production time.

There are several algorithms that can be used to solve TSP. Some of these include metaheuristics such as Genetic Algorithm [GA], Tabu Search [TS] and Variable Neighborhood Search [VNS]. Other algorithms include Ant Colony Optimization [ACO], Particle Swarm Optimization [PSO] and Greedy Algorithm (de Castro Pereira et al.)

- **Genetic Algorithm** is a population-based metaheuristic that uses principles of natural selection and genetics to evolve solutions to optimization problems. GA starts with an initial population of candidate solutions and applies genetic operators such as crossover and mutation to generate new solutions. The fitness of each solution is evaluated and the best solutions are selected to form the next generation.
- **Tabu Search** is a single solution-based metaheuristic that uses memory structures to guide the search for an optimal solution. TS starts with an initial solution and iteratively moves to a neighboring solution that improves the objective function value. To avoid getting stuck in local optima, TS maintains a list of recently visited solutions (tabu list) and prohibits moves that would result in revisiting these solutions.
- **Variable Neighborhood Search** is a metaheuristic that systematically explores different neighborhoods of the current solution to escape local optima. VNS starts with an initial solution and iteratively applies neighborhood change and local search operators until no further improvement can be found.

We will be focussing on two algorithms and a baseline. The baseline algorithm is the brute force approach; the other two algorithms include the Held-Karp algorithm and the nearest neighbor algorithm.

- **Brute force:** or the naïve approach which iterates through every single through path in the graph and keeps track of the minimum cost path. This algorithm has a time complexity of $O(n!)$, it has a space complexity of $O(1)$.

- **Held-Karp algorithm:** Is a dynamic programming algorithm that solves the TSP. It takes advantage of the property that every subpath of a path of minimum distance is itself of minimum distance. Instead of checking all solutions in a naive “top-down” brute force approach, it uses a “bottom-up” approach where all the intermediate information required to solve the problem is developed once and once only. The algorithm has a time complexity of $O(2^n * n^2)$ and a space complexity of $O(n)$. It requires more space to hold all the computed values. A more specific description of the algorithm is presented in the references for (“Held–Karp Algorithm”).

- **Nearest Neighbor Algorithm:** It is a greedy algorithm for finding a sub-optimal solution to the TSP. In this algorithm, one starts with a city as a starting city and repeatedly visits all the cities until all the cities have been visited exactly once. It finds the shortest path but the solution is not necessarily optimal. It differs from VNS, such that VNS is a general-purpose metaheuristic that can be applied to any problem, while the nearest neighbor algorithm is specifically built for TSP. A more specific description of the VNS algorithm is presented in the references for (“Variable Neighborhood Search”). The time complexity of the nearest neighbor algorithm is $O(n^2)$and the space complexity is $O(n)$.

All of the algorithms mentioned above were implemented in Python; they were interpreted by python version 3.11.2 using the package numpy 1.24.1
All of them have been tested using 5 test cases, with test case 4 and 5 being particularly large [10 nodes and 15 nodes, respectively].

All algorithms have been tested using a script written in python that runs all the tests on all the functions three times and calculates the average time taken to run all the tests. The test cases and the script is added to the appendix.

Below is our implementation of the mentioned algorithms:

Brute-force Algorithm:

```python
from itertools import permutations

def brute_force(distances):
    n = len(distances)
    nodes = range(n)
    min_cost = float('inf')
    min_path = None
    for path in permutations(nodes):
        cost = 0
        for i in range(n - 1):
            cost += distances[path[i]][path[i + 1]]
        cost += distances[path[-1]][path[0]]
        if cost < min_cost:
            min_cost = cost
            min_path = path
    return min_cost, min_path
```

The time complexity of the algorithm is super-exponential. This was clearly demonstrated by our testing which could not finish [even after letting it run for over 600 seconds] due to the size of test case 5. To put it in perspective, test case 2 [5 nodes] took 0.016 seconds, while test case 4 [10 nodes] took 30.000 seconds. The algorithm took 1875 times longer for an input with only twice as many nodes.

Held-Karp Algorithm:

```python
from itertools import combinations

def held_karp(distances):
    n = len(distances)
    C = {}
    for k in range(1, n):
        C[(1 << k, k)] = (distances[0][k], 0)
    for subset_size in range(2, n):
        for subset in combinations(range(1, n), subset_size):
            bits = 0
            for bit in subset:
                bits |= 1 << bit
            for k in subset:
                prev = bits & ~(1 << k)
                res = []
                for m in subset:
                    if m == 0 or m == k:
                        continue
                    res.append((C[(prev, m)][0] + distances[m][k], m))
                C[(bits, k)] = min(res)
    bits = (2**n - 1) - 1
    res = []
    for k in range(1, n):
        res.append((C[(bits, k)][0] + distances[k][0], k))
    opt, parent = min(res)
    path = []
    for i in range(n - 1):
        path.append(parent)
        new_bits = bits & ~(1 << parent)
        _, parent = C[(bits, parent)]
        bits = new_bits
    path.append(0)
    return opt, list(reversed(path))
```

The algorithm took 1.200 seconds to run all 5 test cases. The algorithm is about 75 times slower when running test case 5 [15 nodes] in comparison to running test case 4 [10 nodes]. This highlights the exponential time complexity of the algorithm, however it is significantly better than the naïve approach at the cost of some memory which grows linearly with input size.

Nearest Neighbor Algorithm:

```
def nearest_neighbor(distances):
    n = len(distances)
    nodes = set(range(n))
    path = [0]
    nodes.remove(0)
    while nodes:
        current = path[-1]
        nearest = None
        min_distance = float('inf')
        for node in nodes:
            if distances[current][node] < min_distance:
                nearest = node
                min_distance = distances[current][node]
        path.append(nearest)
        nodes.remove(nearest)
    cost = sum(distances[path[i]][path[i + 1]] for i in range(n - 1))
    cost += distances[path[-1]][path[0]]
    return cost, path
```

The algorithm has a quadratic time complexity, this is significantly better than the previous algorithms, however, this results in approximate results instead of the most optimal path. This is made clear by test case 2, 4, 5. The upside is the significant decrease in time taken to compute those paths as it took 0.000 seconds for all 5 test cases to run. This algorithm, like the Held-Karp algorithm, also uses memory which grows linearly with the size of the input.

Conclusion

In conclusion, from the above algorithms, the nearest neighbor algorithm is significantly faster, however, leading to suboptimal paths as a result. The Held-Karp algorithm is a more reliable algorithm, with the most optimal path always being the result. The brute-force algorithm is not preferred as it grows super-exponentially, becoming unwieldy extremely quickly.

Appendix

Script to perform testing and profiling

```
import numpy as np
from time import time
from statistics import mean

functions = [nearest_neighbor, held_karp, brute_force]

np.random.seed(42)
# 1, 2, 3
test_cases = [
    np.array([[0, 10, 15, 20],
              [10, 0, 35, 25],
              [15, 35, 0, 30],
              [20, 25, 30, 0]]),
    np.array([[0, 20, 42, 35, 25],
              [20, 0, 30, 34, 30],
              [42, 30, 0, 12, 10],
              [35, 34, 12, 0, 8],
              [25, 30, 10, 8, 0]]),
    np.array([[0, 10],
              [10, 0]]),
]
# 4
distances = np.random.randint(1, 100, size=(10, 10))
distances = (distances + distances.T) / 2
np.fill_diagonal(distances, 0)
test_cases.append(distances)
# 5
distances = np.random.randint(1, 100, size=(15, 15))
distances = (distances + distances.T) / 2
np.fill_diagonal(distances, 0)
test_cases.append(distances)
```

```
for func in functions:
    times = []
    for _ in range(3):
        start_time = time()
        for test in test_cases:
            func(test)
        times.append(time() - start_time)

    print(f"{func.__name__} algorithm took {mean(times)} seconds to run")
```

All the files are uploaded on Github.

The functions are written with the help of Bing AI (20-03-2023).

References

Alemayehu, Temesgen Seyoum, and Jai-Hoon Kim. “Efficient Nearest Neighbor Heuristic TSP Algorithms for Reducing Data Acquisition Latency of UAV Relay WSN.” *Wireless Personal Communications*, vol. 95, no. 3, Aug. 2017, pp. 3271–85. *Springer Link*, doi:10.1007/s11277-017-3994-9.

Bovet, Daniel P., and Pierluigi Crescenzi. *Introduction to the Theory of Complexity*. Prentice Hall, 1994.

de Castro Pereira, Sílvia, et al. “Genetic and Ant Colony Algorithms to Solve the Multi-TSP.” *Intelligent Data Engineering and Automated Learning – IDEAL 2021*, edited by Hujun Yin et al., Springer International Publishing, 2021, pp. 324–32. *Springer Link*, doi:10.1007/978-3-030-91608-4_32.

“Held–Karp Algorithm.” *Wikipedia*, 5 Dec. 2022. *Wikipedia*, https://en.wikipedia.org/w/index.php?title=Held%E2%80%93Karp_algorithm&oldid=1125702259.

Leeuwen, J. van, editor. *Handbook of Theoretical Computer Science*. Elsevier ; MIT Press, 1990.

“NP-Hardness.” *Wikipedia*, 28 Feb. 2023. *Wikipedia*, https://en.wikipedia.org/w/index.php?title=NP-hardness&oldid=1142170194.

“Travelling Salesman Problem.” *Wikipedia*, 12 Mar. 2023. *Wikipedia*, https://en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=1144174289.

“Variable Neighborhood Search.” *Wikipedia*, 27 June 2021. *Wikipedia*, https://en.wikipedia.org/w/index.php?title=Variable_neighborhood_search&oldid=1030774399.