Travelling Salesman Problem

Kien Quoc Mai

103532920

27/04/2022

COS10009 - Introduction to Programming

Abstract

Travelling salesman problem is a classic problem in combinatorial optimization. This report will state the problem and discuss different approaches to solve it. In addition, some practical applications of it and further research direction will be mentioned.

Contents

1 Introduction

The travelling salesman problem's (TSP) statement is as follows. There is a number of cities and a salesman. The salesman need to visit each city once and return back to the starting city. The problem is that he wants to find the optimal visiting order so that the total length of the tour is minimal.

We can model TSP using graph: Given a complete undirected weighted graph $G = (V, E)$ (see Appendix B) with $n = |V|$ nodes ($n > 2$) and $m = |E|$ edges where c_{uv} ($c_{uv} > 0$) denotes the weight of the edge between vertex u and vertex v. Find a permutation $(p_1, p_2, ..., p_V)$ that minimize the sum $S = \sum\limits_{i=1}^{V-1} c_{p_i p_{i+1}} + c_{p_1 p_n}$

The travelling salesman problem has been proven to be an NP-hard [appendix] problem by Karp in 1972. Generally speaking, there are two approaches to solve TSP: exact algorithms and heuristics algorithms. Then exact algorithms always provide the optimal solution but they are computationally heavy since their worst-case running time increases superpolynomially as the number of cities increases. On the other hand, heuristics algorithms make a trade-off between time complexity and solution's quality. Although they cannot guarantee to find an optimal solution, good approximations can be achieved with a polynomial time complexity.

2 Exact algorithms

2.1 Exhaustive search method

2.1.1 Method

The most straight forward solution to TSP is to apply exhaustive enumeration and evaluation. In other words, an optimal solution can be achieved by generating all possible orders of visit and evaluate the tour's length for each of them. A permutation of the cities can be generate using different algorithms such as Heap's algorithm. Then, the length of the tour can be computed by adding up the edge's length between 2 adjacent cities.

2.1.2 Pseudo-code

```
length = 0
cities = []
minimum = 0
solution = []

function find_solution(index)
    if index > n do
        length = length + distance[cities[1]][cities[n]]

        if length < minimum do
            minimum = length
            solution = cities
        end

        length = length - distance[cities[1]][cities[n]]
        return
    end

    for city in unvisited_cities do
        add city to visited_cities
        cities[index] = city
```

```
            length = length + distance[cities[index - 1]][cities[index]]
            find_solution(index + 1)
            length = length - distance[cities[index - 1]][cities[index]]
            remove city from visited_cities
        end
end

find_solution(1)
```

2.1.3 Full code

See Appendix A for more details.

2.1.4 Analysis

The above implementation of the exhaustive search method iterate through each permutation once and keep track of the total length while generating new permutation. Hence, the overall time complexity is equal to the number of permutations which is $O(n!)$. This factorial time complexity does not scale well with larger inputs. For $n = 20$, the number of tours is 2.4×10^{18}. If a tour can be processed within one millisecond, it would take 77×10^{6} years to produce an optimal solution. On the other hand, the algorithm needs to maintain an array with a size of n which leads to a space complexity of $O(n)$.

2.2 Dynamic programming method (Bellman–Held–Karp algorithm)

2.2.1 Method

In the exhaustive search method, notice that there are some calls to the *find_solution* function that receive the same state and return the same output. For instance, let $n = 6$ and two incomplete states are considered: *cities* $= [1, 2, 3, 4]$ and *cities* $= [1, 3, 2, 4]$. If the optimal solutions for former case is $[1, 2, 3, 4, 5, 6]$, it would imply that $c_{45} + c_{56} + c_{61} < c_{46} + c_{65} + c_{51}$. As a result, the optimal solution for the latter case is $[1, 3, 2, 4, 5, 6]$. It is clear that because the first and current last cities are both 1 and 4 and the remaining unvisited cities are both

5 and 6, the optimal order for the last 2 cities is the same for those two states. Hence, the order for the last two cities can be cached to avoid unnecessary computation. This is the foundation for the Bellman–Held–Karp algorithm.

The details of the algorithm are as follow. Since a solution to the problem is a cycle, the starting city does not matter. Hence, city 1 can be fixed as the starting city. Let $S = \{v_0, v_1, ..., v_k\}$ be the set of already visited cities in any order. Define a function $f(S, u)$ ($u \in S$) that is the minimum path length traversing through all cities in S starting at city 1 and ending at city u. A recursive formula to calculate that is:

$$f(S, u) = min(f(S \setminus u, v)) + c_{uv}(v \in S \setminus u)$$

The base case is: $f(\phi, u) = c_{1u}$ for $u \in V \setminus \{1\}$

The final solution is: $min(f(V \setminus \{1\}, u)) + c_{1u}(u \in V \setminus \{1\})$

2.2.2 Pseudo-code

```
f = []

function calculate_f(S, u)
    if f[S, u] exist
        return f[S, u]
    end

    if S is empty then
        f[S, u] = c[1, u]
        return f[S, u]
    end

    S = S remove u
    result = Infinity
    for v in S do
        result = min(result, calculate_f(S remove v, v) + c[u, v])
    end

    f[S, u] = result
    return f[S, u]
end

result = Infinity
V = V remove 1
for v in V do
    result = min(result, calculate_f(V remove v, v) + c[1, v])
end
write result
```

2.2.3 Full code

See Appendix A for more details.

2.2.4 Analysis

Let D be the domain of f. The number of all possible values for S is the number of subsets of $V \setminus 1$ which is $2^{|V|-1} = 2^{n-1} < 2^n$. The above implementation of the Bellman–Held–Karp algorithm calculates the $f(S, u)$ for every $S, u \in D$ (the domain of f) by calling the function "calculate_f" exactly once. The number of possible inputs has a upper bound of $2^n \times n$. Since "calculate_f" contains a for loop that iterates through every elements in S, each call to "calculate_f" has a complexity of $O(n)$. Overall, the time complexity of the algorithm is $O(2^n \times n \times n) = O(2^n \times n^2)$. On the other hand, because every outputs of function f is cached, the space complexity is $O(2^n \times n)$. For $n = 20$, the time complexity would be $O(2^n \times n^2) = 419,430,400$, which is approximately 5 millions times faster than the exaustive search approach. However, the space complexity of it for $n = 20$ is $20,971,520$. If each output of f is stored using 4-byte-integer, the total memory usage would be $20,971,520 \times 4 = 83,886,080$ bytes $= 80$ megabytes. This is a clear trade-off between time complexity and space complexity that needs to be considered based on the implementation context such as hardware limitations.

2.3 Comparison

In order to compare the two approaches to TSP, time complexity could be utilized to yield an overall view into the performance of each algorithm. Remind from the previous analysis that the time complexities of the exhaustive search and the Bellman–Held–Karp algorithm are $O(n!)$ and $O(2^n \times n^2)$ respectively.

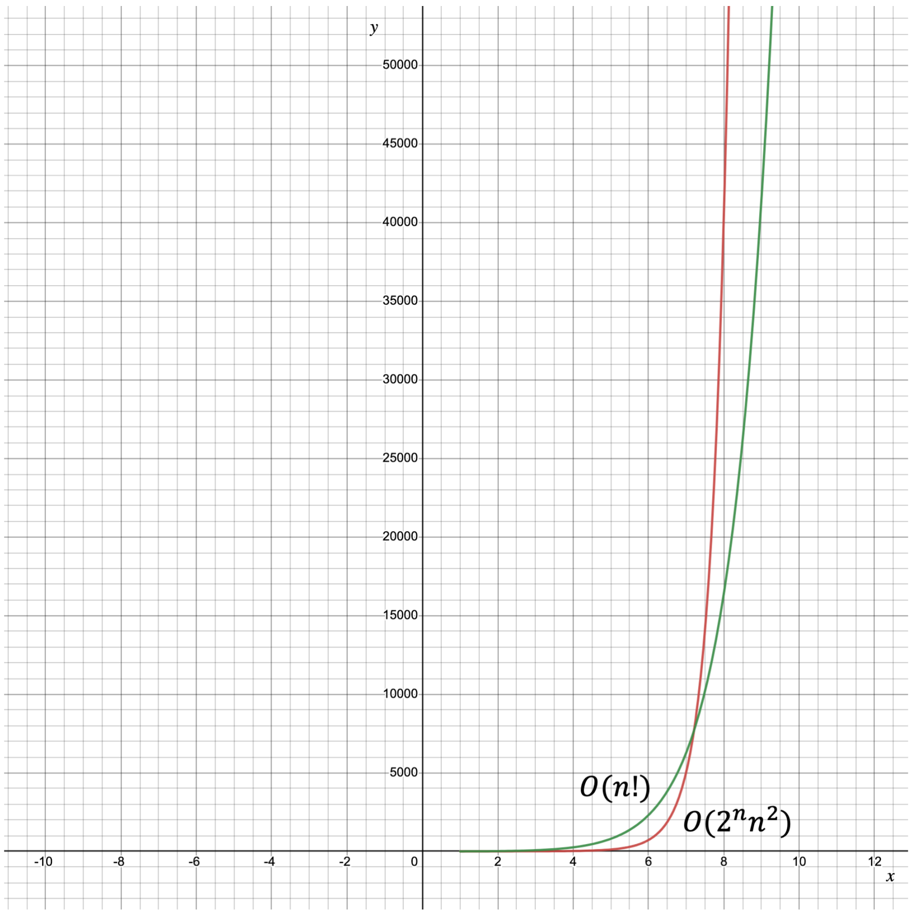

Figure 1: Theoretical time complexity comparison between two algorithms

As can be seen from the graphic, the time complexity of the exhaustive search algorithm grows rapidly as the number of cities increases. Meanwhile, the dynamic approach approach's time complexity also increase greatly with larger n but at a slower pace in comparison to the former approach. For $n < 8$, the exhaustive approach is slightly faster. However, for $n \geq 8$, the time complexity of the Bellman–Held–Karp algorithm is significantly better than its counterpart.

To gain more insights, the implementation of both algorithms has been tested against various datasets to measure actual runtimes. The datasets contains more than a hundred graphs with random edge weight and the number of vertices ranging from 2 up to 12.

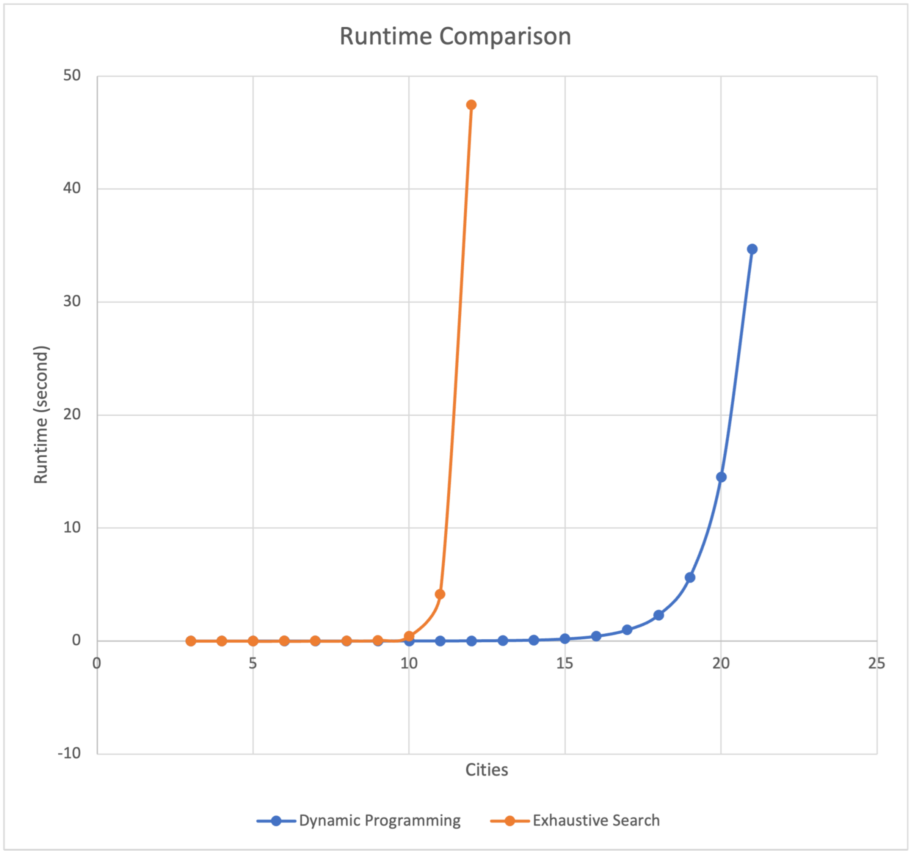

Figure 2: Runtime comparison

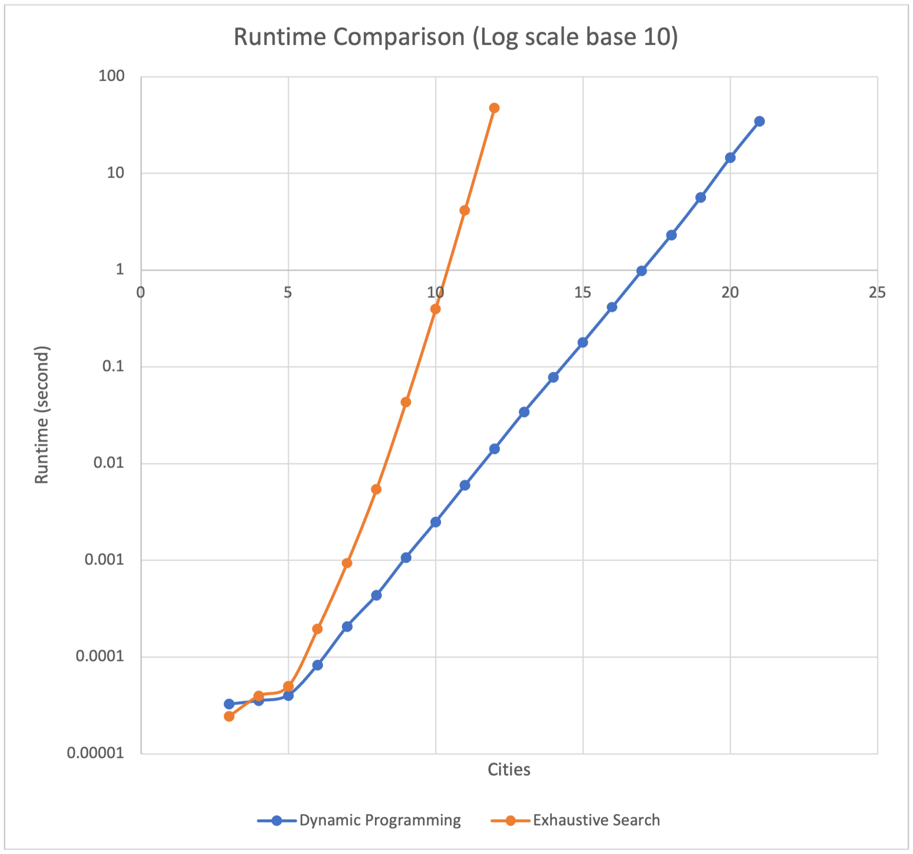

Figure 3: Runtime comparison with log scale plot

Since both of the algorithms have non-polynomial time complexity, which means their runtimes scale up rapidly with larger inputs, a log scale plot is preferred for better understanding of the graph. As demonstrated by the plot, the actual runtimes of those algorithms reflect the time complexities quite accurately. For small inputs, the result is very similar with insignificant difference. However, for large inputs, the difference is prominent - the Bellman–Held–Karp algorithm grows at a slower pace compared to the exhaustive search approach. In addition, the dynamic programming approach starts to grow rapidly in term of runtime at a larger n compared to its counterpart.

3 Heuristic algorithms

3.1 Nearest neighbor algorithm

3.1.1 Method

The nearest neighbor algorithm is one of the most simplest methods to solve TSP. It is a greedy approach which means the it will try to optimize the current state as much as possible without any concern of the overall complete solution. For the nearest neighbor algorithm, the steps are as follow. First, a random city is selected as a starting city. Then, select an unvisited city that is nearest to the last city in term of path cost and add it to the sequence. The previous step is repeated until every city is visited once. Finally, connect the first and the last city together to form a complete tour.

3.1.2 Pseudo-code

```
length = 0
solution = []

last = 1
add 1 to solution

while solution not contains all cities
    min_cost = Infinity
    next_city = -1

    for city not in solution
        if min_cost > distance[last][city]
            min_cost = distance[last][city]
            next_city = city

    if min_cost equal Infinity then
        break

    append next_city to solution
    last = next_city
    length = length + min_cost
end

length = length + distance[last][1]
append 1 to solution

write length
write solution
```

3.1.3 Full code

See Appendix A for more details.

3.1.4 Analysis

For each city in the solution, the algorithm iterate through every possible city to calculate the nearest city to the last city. Hence, the time complexity of this method is $O(n^2)$. This is considered to be a good time complexity since it can be run on a dataset with thousands of cities. However, since it is a greedy approach, some cities may be missed when constructing the visiting sequence and only be inserted later at a much greater cost which heavily affects the final result. Nevertheless, the solution produced by this approach often contains only a few mistakes, which makes it a good starting point for further optimization or a great lower bound for other pruning algorithms.

3.2 Heuristics based on minimum spanning trees

3.2.1 Method

In contrast to the previous greedy approach, which constructs a solution from the ground up, this method produce a solution for TSP based on the graph's minimum spanning tree (see Appendix B). After acquiring a spanning tree (using algorithms such as Kruskal and Prim), a leaf node (see Appendix B) is chosen as a starting city. Then, keep traversing to the next connected and unvisited city in the spanning tree until getting stuck at a node that is not connected to any unvisited cities. Next, traverse backward along the path until a path to unvisited city is found. Repeat the two steps until returning back to the starting city. The cycle obtained from the above process is not yet a valid solution to TSP since some cities are visited more than once. This can be fixed as follow. Whenever it is needed to traverse backward through already visited cities, the unvisited city found by that process is directly added to the sequence with a path from the last city to it. Finally when all cities have been visited, the starting city is directly visited from the last city to complete a tour.

3.2.2 Pseudo-code

```
S = find_minimum_spanning_tree(G)

length = 0
solution = []

function traverse_tree(current_node)
    mark current_node as visited
    for each neighbor of current_node in S
        if neighbor is not visited then
            length = length + c[last node in solution][neighbor]
            append neighbor to solution
            traverse_tree(neighbor)
        end
    end
end

append 1 to solution
traverse_tree(1)

length = length + c[last node in solution][1]
append 1 to solution

write length
write solution
```

3.2.3 Full code

See Appendix A for more details.

3.2.4 Analysis

The time complexity for finding minimum spanning tree varies depending on which algorithm is used. If Kruskal algorithm is used, the time complexity would be $O(|E|log_2|E|) = O(n^2log_2n)$ ($|E|$ has an upper bound of $|V|^2 = n^2$). For the DFS (Depth first search) function, each edge in the spanning tree are traversed twice (forward and backward). Given that the DFS is run on a tree in which $|E| = |V| - 1 = n - 1$, the time complexity of this part is $O(n)$. It is clear that the time complexity of this approach is dominated by that of the minimum spanning tree algorithm. Hence, the overall time complexity of this heuristic approach is $O(n^2log_2n)$. Similar to the greedy method, this algorithm does not always provide an optimal solution but it serves well as an lower bound for further optimization.

3.3 Comparison

Because both algorithms share similar time complexities ($O(n^2)$ and $O(n^2log_2n)$ respectively), they can process relatively large graphs with thousands of cities. However, the main concern with them is the quality of their solutions. A test with actual datasets is conducted to compare the two heuristic algorithms with the exact algorithm. The quality of a solution is measured by the difference to the optimal solution in terms of percentages.

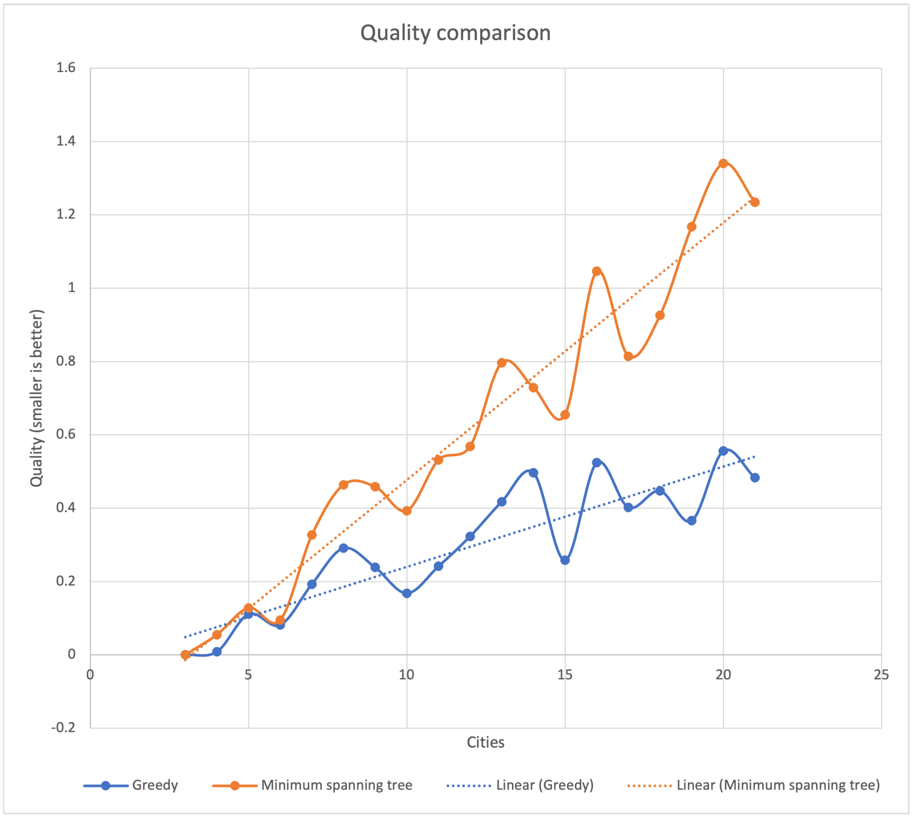

Figure 4: Quality comparison between two heuristic methods (smaller is better)

As demonstrated by Figure 4, desipte using a more complex method to form a tour, the minimum spanning tree method provides worse solutions compared to its simpler counterpart. All in all, the solutions from both algorithms are reasonable given their runtime advantages over exact algorithms (highlighted in Figure 5) which made it more practical for actual applications.

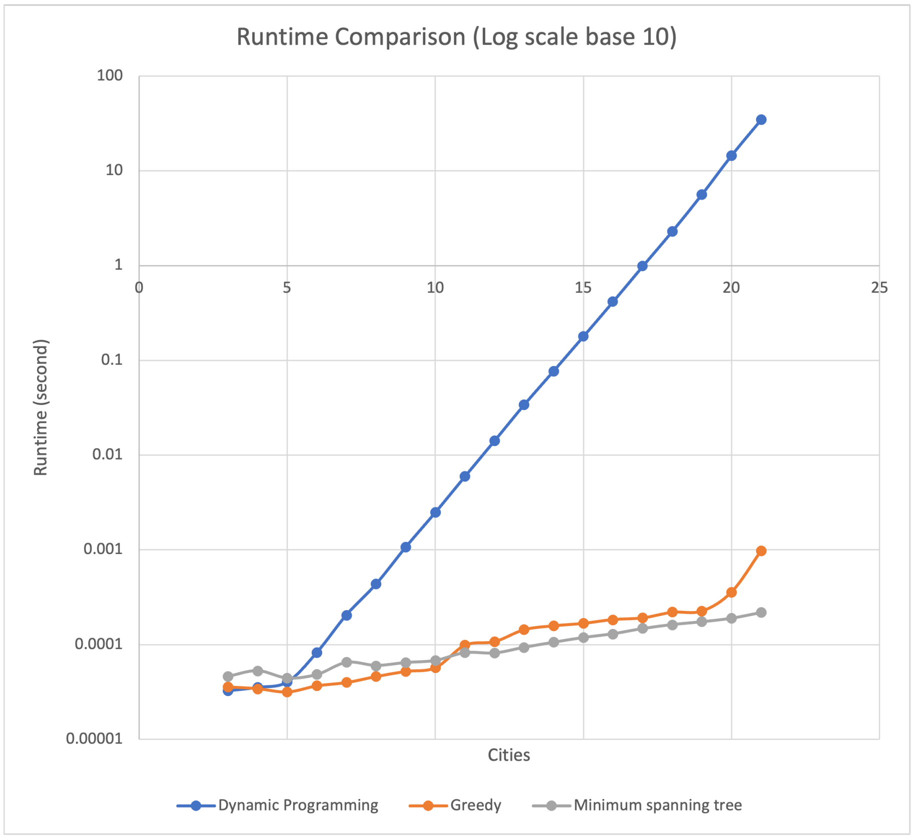

Figure 5: Runtime comparison with log scale plot

4 Practical applications

There are several possible applications that can be solved by modeling the problem as a travelling salesman problem or one of its variants.

- **Vehicle routing:** Parcel delivery is one of a common real-world problem. Suppose that a delivery truck has a number of parcels whose destinations are scattered across a city. Then, an optimal route need to be calculated to minimize to total distance of a trip which ultimately optimizes resources. It clear that TSP algorithms can be utilized to solve this problem if there are no any constraints such as cargo capacity , time and traffic condition. Another variant of this problem is introduced if there are more than one delivery trucks (m trucks). That problem can be model as an m-salesmen problem.
- **Scheduling with sequence dependant process time:** There are a number of jobs that need to be completed. Let the transition time between job i and job j be t_{ij}. Then the problem become a TSP where an optimal sequence of jobs in terms of total processing time need to be solved.

5 Future research

Apart from the symmetric travelling salesman problem discussed in this report, there are a number of TSP's variants and related problems. Two of them are the bottleneck travelling salesman problem and the multisalesmen problem.

First, in the bottleneck TSP, instead of finding a cycle with minimum total length, it is required to minimized to longest edge's length in the cycle. It can be expressed in mathematical form as: find a sequence $v_0, v_1, ..., v_n$ such that $\max(\max_{i=0 \to n-1}(c_{v_i v_{i+1}}), c_{v_n v_0})$ is minimum. This is especially useful in the case that there is a constrain on how far a salesman could travel before he need to rest.

Second, the multisalesmen problem is an extension of TSP. For this variant, instead of only one salesman, there are multiple salesmen (say m salesmen), all starting at the same city. Each city is visited once by one and only one salesman except for the starting city. Then, the problem becomes minimizing the length of all m tours (all salesmen must travel). The solution to this can be utilized to solve the vehicle routing problem for multiples trucks as mentioned earlier.

To summarize, there are many variants and extensions of TSP that have not been covered by this report. The study on those topics would yield greater understanding of TSP and its practical applications.

6 Conclusion

This report has discussed the travelling salesman problem regarding the problem itself and different methods to solves it. Two approaches to this problems (exact and heuristic algorithms) has been analyzed to identify the merits of each method. Lastly some practical applications and further research directions are mentioned along with brief descriptions for each of them.

7 References

Bellman R. (1962). “Dynamic Programming Treatment of the Travelling Salesman Problem”. *Journal of the ACM* 9.1, pp. 61–63. DOI: 10.1145/321105.321111. URL: https://dl.acm.org/doi/10.1145/321105.321111.

Bryant K. (2000). “Genetic Algorithms and the Travelling Salesman Problem”. *HMC Senior Theses*. URL: https://scholarship.claremont.edu/hmc_theses/126.

Jünger M., Reinelt G., and Rinaldi G. (1995). “Chapter 4 The traveling salesman problem”. *Handbooks in Operations Research and Management Science*. Vol. 7. Elsevier, pp. 225–330. DOI: 10.1016/S0927-0507(05)80121-5. URL: https://linkinghub.elsevier.com/retrieve/pii/S0927050705801215.

Weisstein E. W. (2022a). *Graph*. Text. URL: https://mathworld.wolfram.com/.

– (2022b). *Minimum Spanning Tree*. Text. URL: https://mathworld.wolfram.com/.

– (2022c). *NP-Hard Problem*. Text. URL: https://mathworld.wolfram.com/.

– (2022d). *Spanning Tree*. Text. URL: https://mathworld.wolfram.com/.

– (2022e). *Tree*. Text. URL: https://mathworld.wolfram.com/.

– (2022f). *Weighted Graph*. Text. URL: https://mathworld.wolfram.com/.

8 Appendix

8.1 Appendix A

All the code done for this report can be found in this Github repository:

- Exhaustive search method's full code
- Dynamic programming method's full code
- Exact algorithms' benchmarking code
- Greedy method's full code
- Minimum spanning tree method's full code
- Heuristic algorithms' benchmarking code

8.2 Appendix B

Some concepts mentioned in this report but have not been covered:

- **NP-hard problem**: An NP-hard problem is a problems whose algorithm for solving it can be translated into one for solving any NP-problem.
- **Weighted graph**: A weighted graph is a collection of points (vertices) and lines (edges) connecting some (possibly empty) subset of them. Each edge in a weighted graph is assigned a numeric value called weight.
- **Tree**: A connected graph consisting of n vertices and $n-1$ edges is considered a tree.
- **Minimum spanning tree**: A spanning tree of a graph $G(V,E)$ is a tree formed by a subset of $|V|-1$ edges from E. Given that, a minimum spanning tree of a weighted graph $G(V,E)$ is a spanning tree with minimum total edges' weight.

8.3 Appendix C

Table of collected data for Figure 2, 3, 4, and 5.

Table 1: Runtime data for different algorithms

n	Exhaustive Search	Dynamic Programming	Greedy	Minimum Spanning Tree
3	2.44E-05	3.26E-05	3.58E-05	4.62E-05
4	3.97E-05	3.54E-05	3.44E-05	5.31E-05
5	5.01E-05	4.02E-05	3.17E-05	4.44E-05
6	0.000193704	8.27E-05	3.68E-05	4.83E-05
7	0.000939171	0.000206658	3.99E-05	6.50E-05
8	0.005429388	0.000436892	4.61E-05	6.03E-05
9	0.042910471	0.001071142	5.22E-05	6.47E-05
10	0.394868292	0.002484488	5.71E-05	6.81E-05
11	4.137521942	0.005957496	9.87E-05	8.23E-05
12	47.45031898	0.014177412	0.000107712	8.17E-05
13	-	0.0340078	0.000144037	9.38E-05
14	-	0.077598813	0.000158263	0.0001061
15	-	1.79E-01	0.000167962	0.000119025
16	-	4.15E-01	0.000184221	0.000130367
17	-	9.82E-01	0.000191808	0.000147987
18	-	2.30E+00	0.000219787	0.000162996
19	-	5.632617708	0.000226558	0.000174604
20	-	14.51834118	0.000361079	0.0001903
21	-	34.6804868	0.000985962	0.000218696

Table 2: Solution's quality data for different algorithms

n	Dynamic Programming	Greedy	Minimum Spanning Tree
3	0	0	0
4	0	0.008754764	0.054685986
5	0	0.111204659	0.127845771
6	0	0.081447316	0.094737496
7	0	0.192942727	0.326746133
8	0	0.290970378	0.463851425
9	0	0.238383815	0.458400096
10	0	0.168056563	0.393004789
11	0	0.241531244	0.53164546
12	0	0.323197966	0.568503738
13	0	0.418027073	0.797008742
14	0	0.496811901	0.729141915
15	0	0.257905317	0.655277894
16	0	0.523949127	1.046272263
17	0	0.402076874	0.81460755
18	0	0.447251375	0.926137246
19	0	0.365698067	1.167502497
20	0	0.556107519	1.340385245
21	0	0.483138672	1.234636736