TRAVELLING SALESMAN PROBLEM

Kien Quoc Mai

103532920

31/05/2022

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 directions will be mentioned.

Contents

1 Introduction

The travelling salesman problem's (TSP) statement is as follows. There are several cities and a salesman. The salesman needs to visit each city once and return 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 $S = \sum_{i=1}^{V-1} (c_{p_i p_{i+1}}) + c_{p_1 p_n}$

The travelling salesman problem has been proven to be an NP-hard (see Appendix B) problem by Karp in 1972, which means there is no algorithm that solves TSP in polynomial time. Generally speaking, there are two approaches to solving 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 quality. Although they cannot guarantee finding 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 straightforward 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 visits and evaluating the tour's length for each of them. A permutation of the cities can be generated 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)
write length
write solution
```

2.1.3 Full implementation

See Appendix A for more details.

2.1.4 Analysis

The above implementation of the exhaustive search method iterates through each permutation once and keeps track of the total length while generating new permutations. 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 solution 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 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 $f(S, u)$ is:

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

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

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

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

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

2.2.3 Full implementation

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 an upper bound of $2^n \times n$. Since "calculate_f" contains a for loop that iterates through every element 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 output 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 million times faster than the exhaustive 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 of 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.

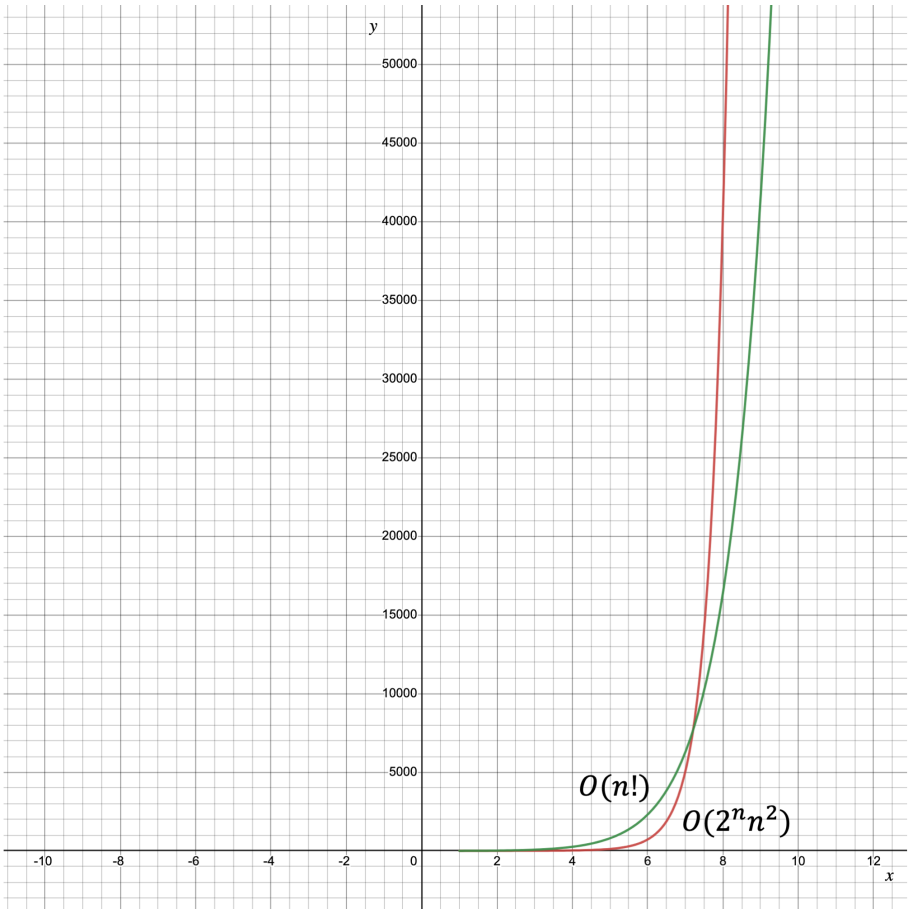

Figure 1: Theoretical time complexity comparison between two algorithms

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 programming approach's time complexity also increases 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 contain more than a thousand graphs

with random edge weights. For each number of cities (n), ranging from 2 up to 21, there are 100 test cases.

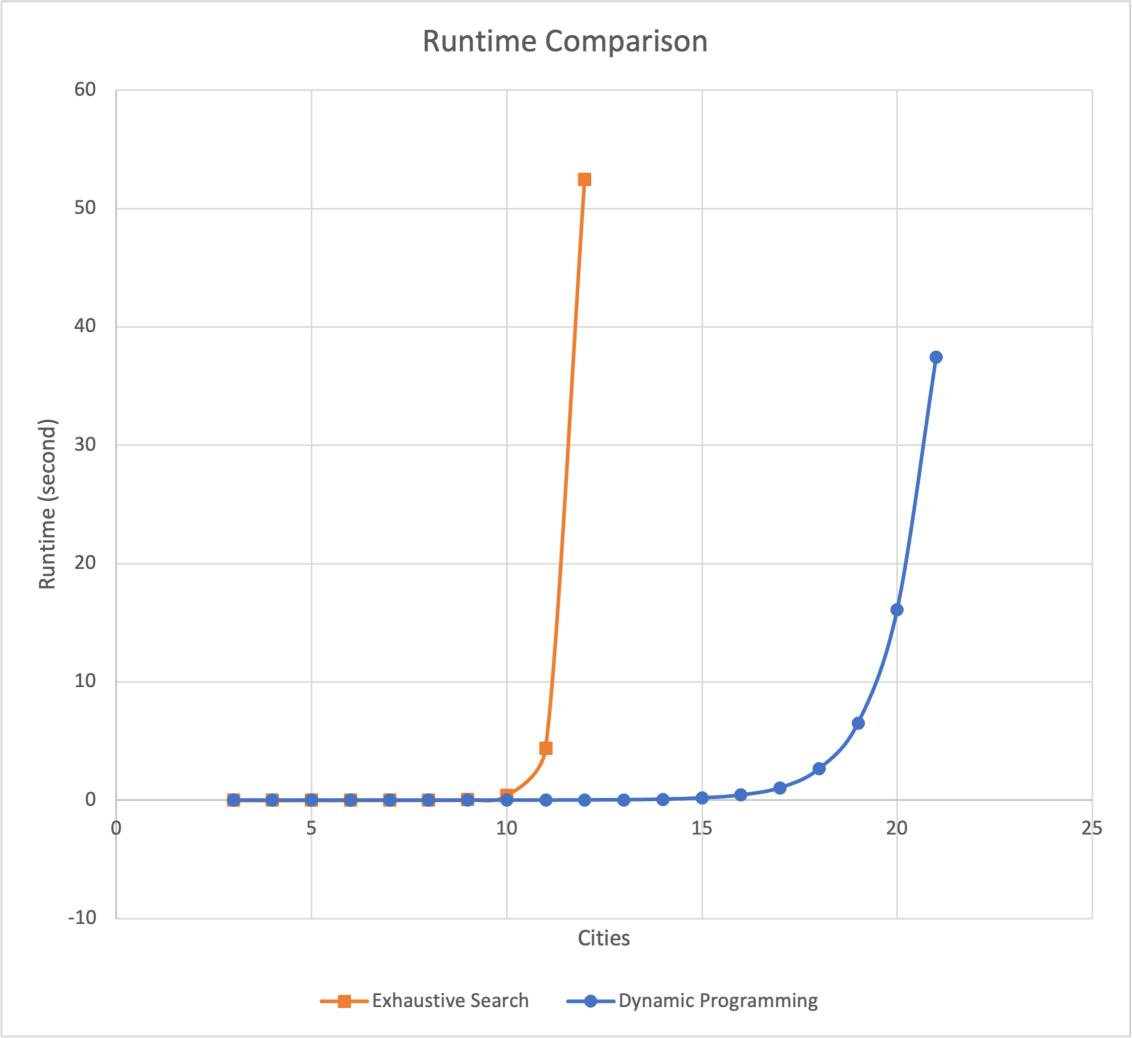

Figure 2: Runtime comparison

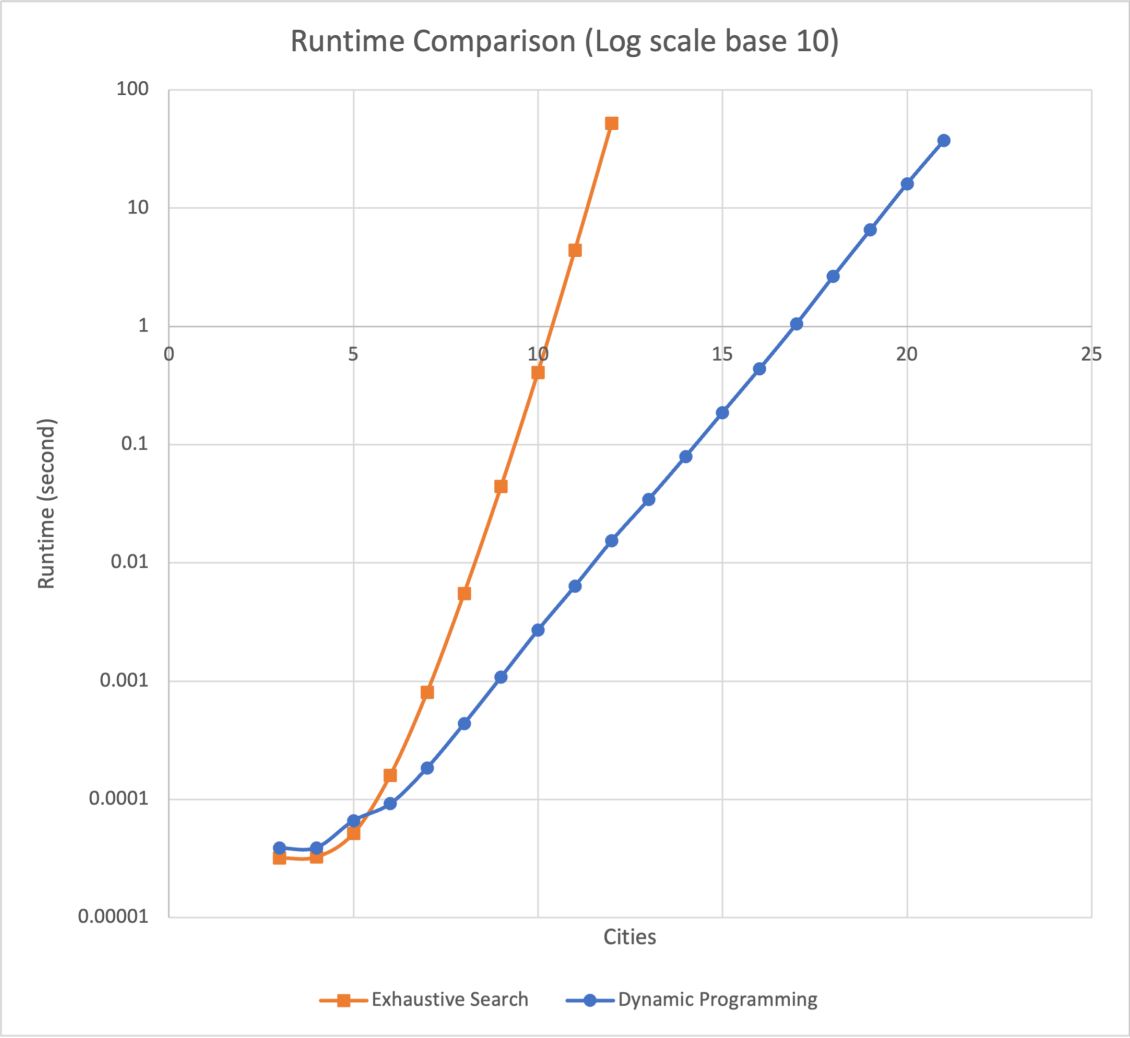

Figure 3: Runtime comparison with log scale plot

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 a 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 differences. 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 terms of runtime at a larger n compared to its counterpart.

3 Heuristic algorithms

3.1 Nearest neighbour algorithm

3.1.1 Method

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

See Appendix A for more details.

3.1.4 Analysis

For each city in the solution, the algorithm iterates 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 produces 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 until a path to an unvisited city is found. Repeat the two steps until returning 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 neighbour of current_node in S
        if neighbour is not visited then
            length = length + c[last node in solution][neighbour]
            append neighbour to solution
            traverse_tree(neighbour)
        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 implementation

See Appendix A for more details.

3.2.4 Analysis

The time complexity for finding a 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 is 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 a 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 benchmark process with actual datasets is conducted to compare the two heuristic algorithms with the exact algorithm. Similar to the previous exact algorithms' benchmark, there are 100 test cases for each number of cities (n). The quality of a solution is measured by the difference from the optimal solution in terms of percentages.

$$Q(n) = \frac{L'(n) - L(n)}{L(n)}$$

Where:

- $Q(n)$: is the quality for test cases with n cities
- $L'(n)$: is the tour's length provided by heuristic algorithms for test cases with n cities
- $L(n)$: is the tour's length provided by exact algorithms for test cases with n cities

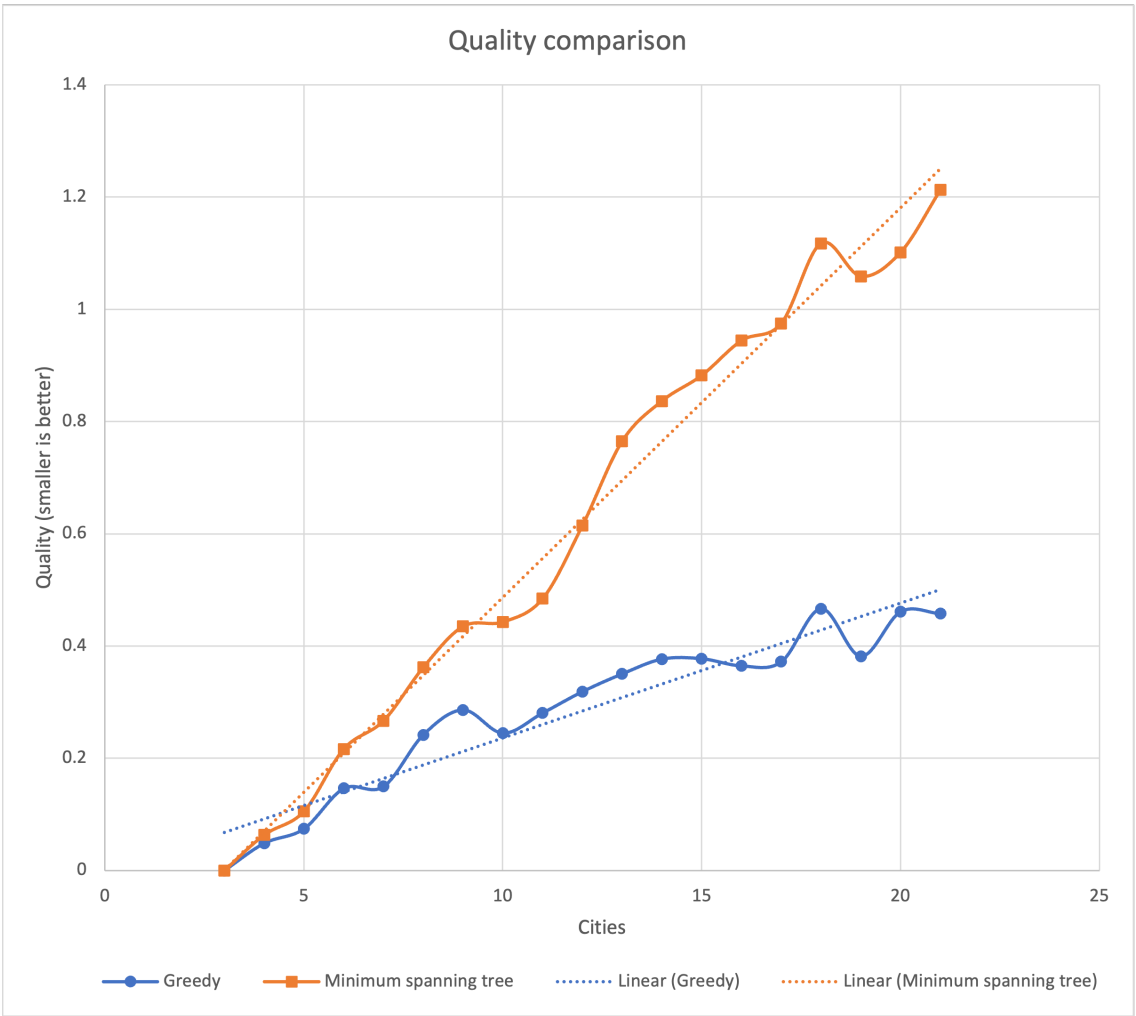

Figure 4: Quality comparison between two heuristic methods (smaller is better)

As demonstrated in Figure 4, despite 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 them more practical for actual applications.

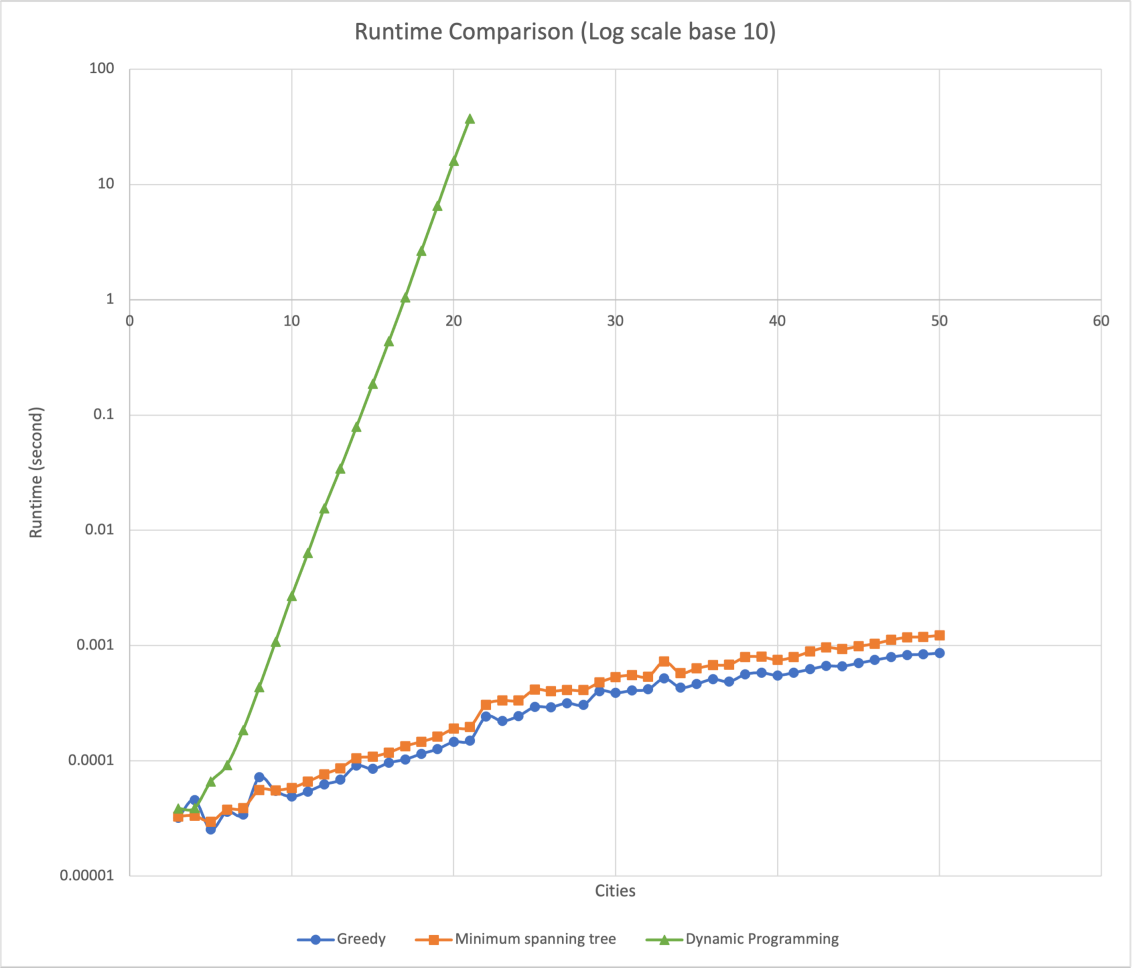

Figure 5: Runtime comparison with log scale plot

4 Practical applications

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

- **Vehicle routing:** Parcel delivery is one of many common real-world problems. Suppose that a delivery truck has a number of parcels whose destinations are scattered across a city. Then, an optimal route needs to be calculated to minimize the total distance of a trip which ultimately optimizes resources. It is clear that TSP algorithms can be utilized to solve this problem if there are not any constraints such as cargo capacity, time and traffic condition. Another variant of this problem is introduced if there are more than one delivery truck (m trucks). That problem can be modeled as an m-salesmen problem.
- **Scheduling with sequence-dependent 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 becomes a TSP where an optimal sequence of jobs, in terms of total processing time, needs 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 a minimum total length, it is required to minimize to longest edge's length in the cycle. It can be expressed in mathematical form as: find a sequence $v_0, v_1, ..., v_n$ so that $\max\left(\max_{i=0\to n-1}(c_{v_i v_{i+1}}), c_{v_n v_0}\right)$ is minimum. This is especially useful in the case that there is a constraint on how far a salesman could travel before he needs 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 in 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 multiple trucks as previously mentioned.

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 a 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 solve it. Two approaches to this problem (exact and heuristic algorithms) have been analysed to identify the merits of each method. In addition, some practical applications and further research directions are mentioned along with brief descriptions for each of them. To summarize, travelling salesman problem is a classic and challenging problem. Although there is no algorithm to solve it in polynomial time, different heuristic methods are utilized to make it more practical for real-world applications.

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 implementation
- Dynamic programming method's implementation
- Exact algorithms' benchmarking code
- Greedy method's implementation
- Minimum spanning tree method's implementation
- 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 problem whose algorithm for solving it can be translated into one for solving any NP-problem.
- **Weighted graph**: A 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 (second)

n	Exhaustive Search	Dynamic Programming	Greedy	Minimum Spanning Tree
3	3.20E-05	3.87E-05	3.20E-05	3.30E-05
4	3.26E-05	3.90E-05	4.58E-05	3.34E-05
5	5.15E-05	6.60E-05	2.54E-05	2.97E-05
6	0.00016029	9.18E-05	3.63E-05	3.78E-05
7	0.00080889	0.000185013	3.42E-05	3.88E-05
8	0.005536434	0.0004367	7.19E-05	5.59E-05
9	0.044285878	0.001078948	5.49E-05	5.55E-05
10	0.409388905	0.002698818	4.90E-05	5.83E-05
11	4.409797075	0.006342875	5.42E-05	6.60E-05
12	52.4641149	0.015447837	6.23E-05	7.66E-05
13	-	0.034396915	6.86E-05	8.64E-05
14	-	0.07934279	9.07E-05	0.00010549
15	-	0.186839729	8.51E-05	0.00010863
16	-	0.436792193	9.63E-05	0.0001182
17	-	1.04754431	0.000102887	0.00013414
18	-	2.648630186	0.000115071	0.00014657
19	-	6.528463197	0.000126489	0.00016175
20	-	16.09617029	0.000146323	0.00018983
21	-	37.42620619	0.000150178	0.00019683
22	-	-	0.000242441	0.00030692
23	-	-	0.000221559	0.0003335
24	-	-	0.000245053	0.00033468
25	-	-	0.00029299	0.00041423
26	-	-	0.000292535	0.00040173
27	-	-	0.000317175	0.00041275
28	-	-	0.000305667	0.00041027
29	-	-	0.000402944	0.00047917

Table 1: Runtime data for different algorithms (second)

n	Exhaustive Search	Dynamic Programming	Greedy	Minimum Spanning Tree
30	-	-	0.000388638	0.00053329
31	-	-	0.000407318	0.00055316
32	-	-	0.000417912	0.00053785
33	-	-	0.000517876	0.00072795
34	-	-	0.000432757	0.00057776
35	-	-	0.000464486	0.00063409
36	-	-	0.000511101	0.00067547
37	-	-	0.000486732	0.0006832
38	-	-	0.000563257	0.00079427
39	-	-	0.00058113	0.00079887
40	-	-	0.000551677	0.00075031
41	-	-	0.000582617	0.00079576
42	-	-	0.000622852	0.00088708
43	-	-	0.000665025	0.00096447
44	-	-	0.000663279	0.00093414
45	-	-	0.000704175	0.00098734
46	-	-	0.000749287	0.00103953
47	-	-	0.00079405	0.00111978
48	-	-	0.000829941	0.00118176
49	-	-	0.000839869	0.00119177
50	-	-	0.000862333	0.00122854
51	-	-	0.000954714	0.00129644
52	-	-	0.000929491	0.00136683
53	-	-	0.001012238	0.00150382
54	-	-	0.001035772	0.00158919
55	-	-	0.001055875	0.00162725
56	-	-	0.001058467	0.00165286
57	-	-	0.001180776	0.00168393
58	-	-	0.001195885	0.00177184
59	-	-	0.001185154	0.00176879
60	-	-	0.001268166	0.00187739

Table 1: Runtime data for different algorithms (second)

n	Exhaustive Search	Dynamic Programming	Greedy	Minimum Spanning Tree
61	-	-	0.001265423	0.00190643
62	-	-	0.001272704	0.00192692
63	-	-	0.001383263	0.0019957
64	-	-	0.001398047	0.00212818
65	-	-	0.001403449	0.00217676
66	-	-	0.001534086	0.00221613
67	-	-	0.001542611	0.00228385
68	-	-	0.001734008	0.00244817
69	-	-	0.001684037	0.00255723
70	-	-	0.001666841	0.00250425
71	-	-	0.001746566	0.002576
72	-	-	0.001728454	0.00263326
73	-	-	0.001823995	0.00273359
74	-	-	0.001846299	0.00277479
75	-	-	0.001914631	0.00290564
76	-	-	0.002046453	0.00306483
77	-	-	0.002017704	0.00309652
78	-	-	0.002000932	0.00313462
79	-	-	0.002121675	0.00328526
80	-	-	0.002121509	0.00328628
81	-	-	0.002168084	0.00333616
82	-	-	0.002257213	0.0035315
83	-	-	0.002305034	0.00348768
84	-	-	0.002373268	0.00365833
85	-	-	0.00240029	0.00370327
86	-	-	0.002428881	0.00369138
87	-	-	0.002584143	0.00396798
88	-	-	0.002593448	0.00391388
89	-	-	0.002618243	0.00409043
90	-	-	0.002767014	0.00421322
91	-	-	0.002695329	0.00428417

Table 1: Runtime data for different algorithms (second)

n	Exhaustive Search	Dynamic Programming	Greedy	Minimum Spanning Tree
92	-	-	0.002898873	0.00437533
93	-	-	0.002971953	0.00464328
94	-	-	0.002980004	0.00460544
95	-	-	0.002940619	0.00465288
96	-	-	0.003046868	0.00477333
97	-	-	0.003143457	0.00487525
98	-	-	0.00325369	0.00500448
99	-	-	0.003120419	0.00483338

Table 2: Solution's quality data for different algorithms

n	Dynamic Programming	Greedy	Minimum Spanning Tree
3	0	0	0
4	0	0.048530389	0.06342403
5	0	0.07434006	0.10538183
6	0	0.146247191	0.2165839
7	0	0.149566195	0.26679899
8	0	0.241765986	0.36194666
9	0	0.286223001	0.43502506
10	0	0.2447654	0.44287034
11	0	0.280671586	0.48435827
12	0	0.318777877	0.61463786
13	0	0.350375772	0.76492082
14	0	0.376800272	0.83656684
15	0	0.377516463	0.88213387
16	0	0.364688917	0.94427919
17	0	0.371998542	0.97507249
18	0	0.466522251	1.11772021
19	0	0.381738648	1.05861191
20	0	0.461267497	1.10159958
21	0	0.457649963	1.2126893