
THE UNIVERSITY *of* EDINBURGH

# Inf2 – IADS 2021-22

## Coursework 3

## The Travelling Salesman Problem

Submitted by: **s2106632**

March 20, 2022

# Contents

# 1 Algorithm

## 1.1 Cheapest Insertion Heuristics

In Part C, I implemented a Heuristic called Cheapest Insertion, implemented and analyzed by Rosenkrantz, Stearns, Lewis, 1977[1]. This kind of idea for solving problem was raised by T. A. J. Nicholson in 1967[2]. This heuristics solves the TSP by starting with one edge (2 vertices), and each time adds an edge that costs the smallest to the whole path. More specifically, after each time one edge is added, it will search every unvisited vertex and tries to add it in the middle of one existed edge and calculate the added cost. Then it will choose the vertex and the edge which is cheapest. However, this heuristic can only work under the **assumption** that the graph fulfills **triangle inequality** ($w(x, y) \leq w(x, z) + w(z, y)$), as the whole method works based on this.
The pseudo code of this Heuristic would be as follows:

*Cheapest Insertion*
**input**
*A weighted graph* $G = (V, E)$.
**begin**
$k, l := \arg\min\{c_{ij} + c_{ji} : i, j \in V, i \neq j\}$.
$T := (k, l, k)$.
**while** *T is a partial tour*
$v := \arg\min\{c(T, k) : k \notin T\}$.
$T := \text{insert}(T, v)$.
**end while**
**end** *Cheapest Insertion*

Figure 1: Cheapest Insertion[3]

With Cheapest Insertion, one graph can generate different results of tours based on different choices of starting points (i, j, in my implementation). If we want to generate the global optimal using this heuristic, we may have to try every combination of values of i and j since different edges have different costs, and adding different vertices give different added cost, which would be $C_2^n$ combinations. And this loop will also be in polynomial time, this will be proved after.

```
globalMin = None
g = newGraph()
for i in range(n):
    for j in range(i, n):
        g.perm = default perm(0 .. n-1)
        cheapestInsertion(Graph, i + 1, j)
        cost = g.tourvalue()
        if globalMin is None or cost < globalMin:
            globalMin = cost
```

Figure 2: Get Global Minimum

## 1.2 Justification of polynomial time

The runtime of my implementation of Cheapest Insertion is slightly different from the past work's, which is $O(n^2 \log n)$, but it's also in polynomial time, which is $O(n^4)$.
Note that the frame of my implementation is made up of a while loop with 2 nested for loops in it. The while loop will run for exactly (n - 2) times, as *currentEdges* starts with one element, after each time, one edge would be deleted, and two new edges would be inserted. Therefore, there would be one edge added in total. The while loop will not end until the *currentEdges*'s length reaches n - 1, which would certainly take $\frac{n-1}{1}$ times for the while loop to reach.
As for the nested for loops, each line in the loops will be executed in O(1), since each step is retrieving values or setting values, we only need to care about how many times the loops will be invoked then. Since in each time the while loop executes, the length of *currentEdges* will be increased by one, and the length of *unvisited* will be decreased by one, the number of loops executed is constantly changing, and the total number of loops will be as follows:

$$
\begin{aligned}
&1 \times (n-2) + 2 \times (n-3) + ... + (n-3) \times 2 + (n-2) \times 1 \\
&= \sum_{i=1}^{n-2} i \cdot (n-i-1) \\
&= \sum_{i=1}^{n-2} ni - \sum_{i=1}^{n-2} i^2 - \sum_{i=1}^{n-2} i \\
&= n \sum_{i=1}^{n-2} i - \sum_{i=1}^{n-2} i^2 - \sum_{i=1}^{n-2} i \\
&= n \cdot \frac{n(n+1)}{2} - \frac{n(n+1)(2n+1)}{6} - \frac{n(n+1)}{2} \\
&= \frac{n^3 - n}{2} - \frac{n(n+1)(2n+1)}{6} \\
&= \frac{n^3 - 3n^2 - 4n}{6}
\end{aligned}
$$

As n is increasing and diverges to $\infty$, the dominant term here would be $n^3$, thus the nested for loops in the while loop will be $O(n^3)$. For the other actions in the while loop, they would certainly not exceed $O(n^3)$, as they are mostly removing and adding element in the list. The most costly execution would be the list comprehension, where the list is in the size of *len(currentEdges)*, as its length increases by one in each outer for loop.
Thus the total time complexity would be:

$$O(n) \times (O(n^3) + O(n) + O(1)) = O(n^4) + O(n^2) + O(n) = O(n^4)$$

And this is certainly a polynomial time complexity to generate result.
When we use this Heuristic to find the most possible optimal solution, we have to try every combination of starting vertices as said above. And this is also in polynomial time. Since the nested for loops will

exactly run $C_2^n = \frac{n(n-1)}{2}$ times, which is $O(n^2)$. For the lines in the nested loops, each time the perm is reset in $O(n)$, and each time the heuristic will run in $O(n^4)$ as suggested. Getting tour value will also run in $O(n)$. Here the dominant term is $O(n^4)$, the whole nested loop will run in $O(n^2) \times O(n^4) = O(n^6)$. We can see that getting the global minimum tour value is still in polynomial time.

## References

[1] D.J.Rosenkrantz, R.E.Stearns, P.M.Lewis, *An analysis of several heuristics for the traveling salesman problem.* SIAM J. COMPUT. Vol. 6, No. 3, September 1977.
[2] T. A. J. NICHOLSON, *A sequential methodfor discrete optimization problems and its application to the assignment, traveling salesman, and three machine scheduling problems*, J. Inst. Math. Appl., 3 (1967), pp. 362-375.
[3] Refael Hassin, Ariel Keinan, *Greedy Heuristics with Regret, with Application to the Cheapest Insertion Algorithm for the TSP* Operations Research Letters Volume 36, Issue 2, March 2008, Pages 243-246

# 2 Experiments

## 2.1 Analysis of values

Now that we have 4 heuristics that could generate approximated solution in polynomial time, we then need to evaluate theses four methods.
As we are trying to find the best approximates solution in polynomial time, for 2-opt and swap, the maximum value they can take in as parameters is the value *self.n*. However, for cheapest insertion, it can try every pair of vertices as its parameters and this will run in polynomial time too as proved in part-C. Thus, I will show the global minimum for cheapest insertion.

| Heuristics | six nodes | twelve nodes | cities50 | metric 50 nodes | Euclidean 100 |
|---|---|---|---|---|---|
| Swap | 9 | 12 | 8707 | 2507 | 37985 |
| 2-opt | 9 | 25 | 2781 | 1531 | 8323 |
| Greedy | 8 | 26 | 3011 | 1589 | 9835 |
| Cheapest Insertion | 8 | 24 | 2708 | 1476 | 8402 |
| Default perm | 9 | 34 | 11842 | 3210 | 50079 |

(Metric 50 nodes is a graph with 50 nodes and the upperlimit of distance is 100. For Euclidean 100, there are 100 coordinates, where for $(x, y)$, $x, y \in [50, 1000]$).
The results of the tour values are interesting. In most of the cases, 2-opt and Cheapest give the minimum values among the four heuristics. The reason why Greedy method performs a bit poorer is that Greedy could be easily trapped into local optimal, and because of that, it will go away from global optimal. Swap heuristics is not generally as effective as others, since simply swapping 2 vertices may give better result, but that each time of swapping will cut too many other possible branches, and that is more possible for it to give up the optimal solution. To get a more intuitive feeling about how much a heuristic can do, I will show how many percents it's better than the default one:

| Heuristics | six nodes | twelve nodes | cities50 | metric 50 nodes | Euclidean 100 |
|---|---|---|---|---|---|
| Swap | 0% | 5.9% | 26.5% | 21.9% | 24.1% |
| 2-opt | 0% | 26.5% | 76.5% | 52.3% | 83.4% |
| Greedy | 11.1% | 23.5% | 74.6% | 50.5% | 80.4% |
| Cheapest Insertion | 11.1% | 29.4% | 77.1% | 54.0% | 83.2% |
| Default perm | 9 | 34 | 11842 | 3210 | 50079 |

From this table, we can clearly see that in most of the time, Cheapest Insertion gives the best result. And as the number of nodes and distance increase, the performance for all heuristics is much better except for **Swap**.

## 2.2 Analysis of run time

While comparing heuristics, except for comparing the values, we also care a lot about the actual run time. As obviously, we can find exact solutions for TSP problems, but we don't use them in general, and that is because when the number of n increases, the run time increases a lot faster, which makes it become impossible to find an exact solution in a relatively reasonable time. We have to do some trade-offs between the effect and the run time while solving real problems.

| Heuristics | six nodes | twelve nodes | cities50 | metric 50 nodes | Euclidean 100 |
|---|---|---|---|---|---|
| Swap | 0.00012 | 0.00042 | 0.0080 | 0.012 | 0.040 |
| 2-opt | 0.00016 | 0.0017 | 0.20 | 0.16 | 1.59 |
| Greedy | 0.00010 | 0.00022 | 0.0022 | 0.0034 | 0.0077 |
| Cheapest Insertion | 0.00051 | 0.011 | 11.27 | 11.65 | 350.27 |

From the table above, one really obvious thing is that as the number of vertices increases, the heuristics **Cheapest Insertion** becomes incredibly slow. And that, according to our polynomial time analysis, is because we are finding a global minimum. It's true the four heuristics are all polynomial, but **Cheapest Insertion** is as high as $n^6$, **2-opt** is only $n^2$.

## 2.3 Summarise

Now we collect the results of both the run time and values, we can conclude that, **swap** heuristic is the fastest one, but its result is also the worst. We probably need better solutions, as even the run time is low, it's also more possible that it cannot fulfill the standards on how small the tour values are.

As for **2-opt** and **cheapest insertion**, the two most competing heuristics on values, the optimized values are really close, but comparing their run time, we can clearly say 2-opt is much better.

For **greedy** method, we can tell that it's generally less competing than the two above on values, but it's also performing well, only small number of percentages fewer. Its most competing part is time, as we can see from the table, even with 100 vertices, the run time is still really low, less than 0.01 second. The run time won't change much as the number of vertices increase. This especially suits the situation that there are hundreds of vertices, and we want a relatively good solution. As we have discussed, **cheapest insertion** is not a good choice when *self.n* is large. We can also tell that **2-opt** has a trend that as *self.n* increases, the run time grows faster. From 50 nodes to 100 nodes, *self.n* increases 2 times, but run time increases almost 10 times. Therefore, for some incredibly large number of vertices, we probably will want to use **Greedy** method to solve the problem.