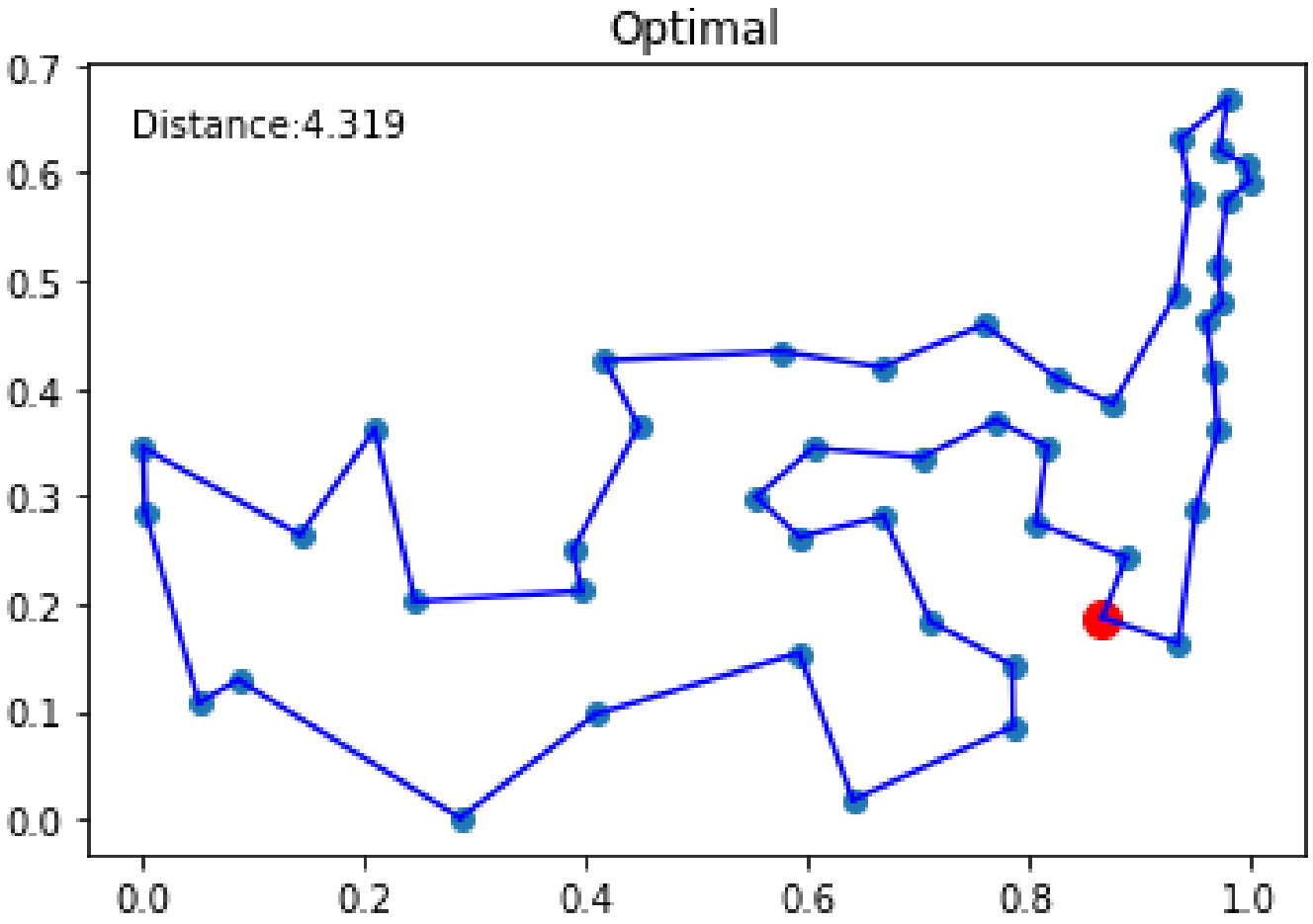
Theoretical and Practical Effectiveness of Greedy Algorithms for the Travelling Salesman Problem

Jonathan Ouwerx
B Block
College



Travelling Salesman Problem

By Jonathan Ouwerx

1 Table of Contents

2 Introduction

The Travelling Salesman Problem is an incredibly well researched problem in computer science. Its study has unbelievable academic and practical value and so it is unsurprising that it is prevalent in many walks of life. The problem had its most notable origin in the early 20th century when lawyers, priests and salesmen would travel around the country participating in circuit courts, preaching, and selling. However it was not until Hassler Whitney discussed the 'Travelling Salesman Problem' in a lecture in 1934 that it gained traction in the mathematical community, although he denies any recollection of such a lecture[1]. The Travelling Salesman Problem (TSP) refers to the problem of finding the optimal tour, the shortest path between a number of points, visiting each point once and then returning to the first point. Every possible tour is a Hamiltonian path: it visits every point just once, except for the first and final one[2]. The problems that I will be focusing on will be geometric. They obey the triangle inequality which states that the distance from node i to node k is shorter than the distance from node i through node j to node k. c_{ij} refers to the distance between two nodes.

$$c_{ik} \leq c_{ij} + c_{jk}$$

Since the 20th century, the TSP's relevance has become so much greater. Its applications in map routing are even more significant with the advent of home delivery, as well as its applications in genome ordering, scheduling, and microprocessor design[3]. As the demand for algorithms has increased, so has the supply. Solutions have been inspired from all walks of life with examples including genetic algorithms, ant colony optimization and simulated annealing, as well as the more abstract algorithms, such as the k-opt method[4].

2.1 Time Complexity

The time complexity of a problem refers to how quickly it can be solved, in what time magnitude. A constant time problem always takes the same amount of time no matter how large the input size, for example finding the last number in a list. We can use Big-O notation to show this, with the constant time problem being denoted by O(1). A linear search requires as many operations as there are items in the input list. If there are n items, then only n operations are necessary. The problem is O(n). Big-O notation refers to the limiting behaviour of an algorithm, the asymptotic result, so any smaller order terms are abstracted away[5].

Basic sorting algorithms are of quadratic time complexity, for example the Insertion Sort. An item is picked from the list one at a time and compared to every other previously selected item. While it is feasible that a list is already fully sorted, it is standard practice to refer to the worst-case scenario, so Insertion Sort is $O(n^2)$[6]. While there are various intermediary tiers,

---

[1] Cook, William(2014). *In Pursuit of the Travelling Salesman*, Chapter 2, Princeton University Press

[2] En.wikipedia.org. 2021. *Hamiltonian path - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Hamiltonian_path>

[3] Cook, William(2014). *In Pursuit of the Travelling Salesman*, Chapter 3, Princeton University Press

[4] E. H. L. Aarts and J. K. Lenstra (eds.), *Local Search in Combinatorial Optimization*, John Wiley and Sons, London, 1997, pp. 215-310

[5] Austinmohr.com. 2021. [online] Available at: <http://www.austinmohr.com/Work_files/complexity.pdf>

[6] En.wikipedia.org. 2021. *Insertion sort - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Insertion_sort#Best,_worst,_and_average_cases>

such as logarithmic and linearithmic time complexity, all algorithms of these types are considered good algorithms; they can solve problems in a reasonable amount of time. Is there a definition for a good algorithm? Yes, according to Cobham's Thesis, any algorithm that runs in polynomial time (by a deterministic Turing machine), mathematically expressed as $O(n^c)$ is good (or tractable)[7]. These are the constituents of the P group, P referring to polynomial time complexity. When problems are beyond P, they can take much, much longer to solve, illustrated in Figure 1. The brute-force algorithm for a TSP is $O(n!)$, one of the most terrifying time complexities for mathematicians and computer scientists (one of the best exact algorithms is the Held-Karp Dynamic Programming algorithm, taking time $O(n^2 2^n)$ [8]). Crucially, Big-O notation refers to the upper bound of an algorithm/problem, so all the algorithms explained above are technically $O(n!)$.

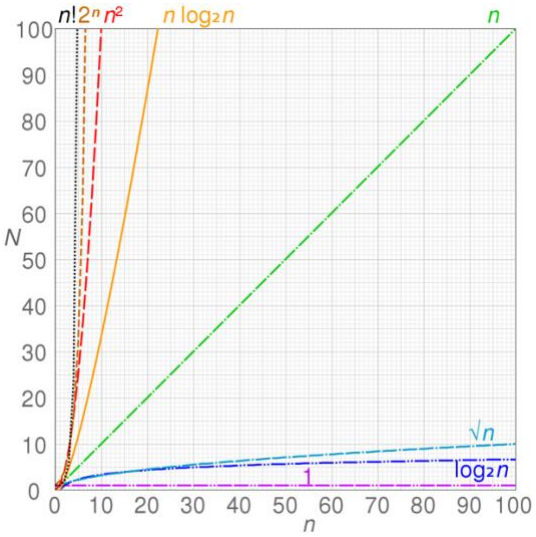


Figure 1; Input size n versus number of operations for given time complexities.[9]

Beyond P problems, there are three other classes: NP, NP-Complete and NP-Hard. NP problems can be verified in polynomial time, so by definition, NP contains P. NP-Complete problems are those which when solved by an algorithm in polynomial time, will be able to solve all other NP problems in polynomial time, at which point we would know that P = NP (I will discuss this shortly). They are effectively the hardest problems in NP and are part of both NP and NP-Hard, by definition. NP-Hard is defined as the set of all problems to which all NP problems can be reduced in polynomial time. The relationships between these three classes are shown in Figure 2 and the current leaning of the mathematical community is that $P \neq NP$ i.e. the left part of the graph is correct, not the right. However, the answer is still unknown and very valuable: P versus NP is one of the Millennium Prize Problems with an attached prize of $1,000,000.

---

[7] Cobham, Alan (1965). "The intrinsic computational difficulty of functions". *Proc. Logic, Methodology, and Philosophy of Science II*. North Holland.

[8] 'A dynamic programming approach to sequencing problems', Michael Held and Richard M. Karp, *Journal for the Society for Industrial and Applied Mathematics* 1:10. 1962

[9] https://upload.wikimedia.org/wikipedia/commons/thumb/7/7e/Comparison_computational_complexity.svg/440px-Comparison_computational_complexity.svg.png

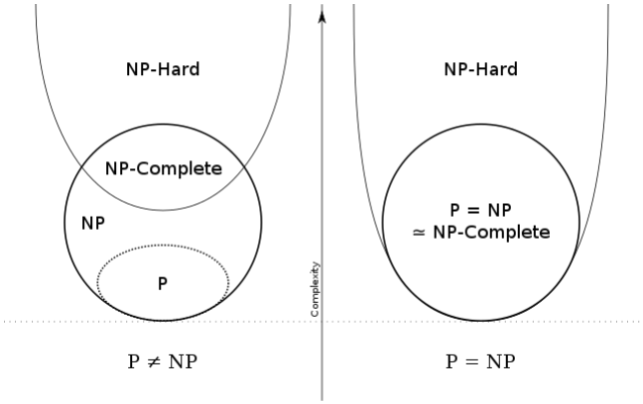


Figure 2; Diagram representing P v NP problem and the structure of NP, NP-Complete and NP-Hard

Figure 2; Diagram representing P v NP problem and the structure of NP, NP-Complete and NP-HardThe optimisation version of the TSP is NP-Hard. By changing it to a decision problem (is there a route shorter than x?), we can turn it into an NP-Complete problem, as it would now be easily verifiable. The overall conclusion is that it is incredibly tough to both find the optimal solution and to prove that it is correct.

3 Practical Implementation

3.1 Apparatus

To compare and visualise various algorithms for the TSP, I developed a framework in python to generate random points, a distance matrix for those points and a graphical representation. The first step in finding the optimal tour of a set of points is generating those points. Using the 'random' module in python, two lists of numbers between 0 and 1 were generated, with the lists corresponding to the x and y axes. Then a distance matrix was created; a matrix where the distance between any pair of points is displayed. In Figure 3, 10 random points are shown, with their coordinates and relative distances described below. The coordinates, distance matrix and the program used are displayed in Appendix 7.1 and Appendix 7.2.

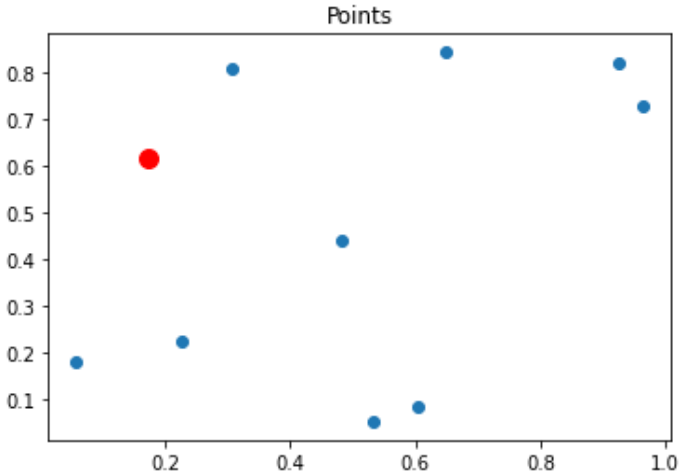


Figure 3; Set of ten random points, with the red point indicating the starting point.

3.2 Algorithms Introduction

There are many different algorithms which have been designed to solve the TSP with over 130 showcased at the DIMACS TSP Challenge[10]. The two metrics that determine the quality of a TSP algorithm are the proximity of the output tour to the optimal tour and the speed with which that tour is found. Proximity is measured simply by the percentage difference between output tour length and optimal tour length, and the speed can be measured either in practice or by calculating the time complexity by algorithmic analysis. These two metrics will be used to compare some of the more common TSP algorithms.

3.3 Brute Force Algorithm

The Brute Force algorithm iterates through every possible tour and records the tour with the shortest length. As it is exhaustive, we can be sure that the output length and point order will be the shortest. It also means that it will take a very large amount of time, running in $O(n!)$ time. An alternative exact algorithm is the dynamic programming (DP) algorithm, running in $O(n^2 2^n)$ time. The main difference between the two is that the DP algorithm records certain parts of the journey in the memory i.e. the shortest distance from the penultimate, antepenultimate or any similar such node to the final node.

3.3.1 Mathematical Model

Both the DP and Brute Force algorithms can be represented by a recursive mathematical formula, shown below[11] (The formula ignores the distance between the final point and the starting point; however this can easily be added). i is some starting point and s is the subset of points not yet reached by the tour. c_{ik} is the cost of travelling from point i to point k, where k is a point in s. g is a cost function for a tour from the starting point i to all the points in s.

$$g(i,s) = \min_{k \in s}\{c_{ik} + g(k, s - \{k\})\}$$

One way to read the formula is as follows:
The minimum cost of travelling from point i to all the points in set s equals the cost of travelling from point i to point k plus the cost of travelling from point k to a next point in the set s, with k now removed, and so on, where at each stage the cost function is the minimum.

We can create a formula with specific i and s:

$$g(1,\{2,3,4\}) = \min_{k \in \{2,3,4\}}\{c_{1k} + g(k, \{2,3,4\} - \{k\})\}$$

This formula can be visualised as a tree branching out at each stage, one branch of which is shown below.

$$g(1,\{2,3,4\}) = \min_{k \in \{2,3,4\}}\{c_{12} + g(2, \{2,3,4\} - \{2\})\}$$

---

[10] Johnson, D. S., L. A. McGeogh. 2002. In: G. Gutin, A. Punnen, eds. *The Traveling Salesman Problem and Its Variations. Kluwer*, Boston, MA. 369-443.

[11] Bari, A., 2021. *4.7 [New] Traveling Salesman Problem - Dynamic Programming using Formula*. [video] Available at: <https://www.youtube.com/watch?v=Q4zHb-Swzro>

$$g(2,\{3,4\}) = \min_{k\in\{3,4\}}\{c_{23} + g(3,\{3,4\} - \{3\})\}$$

$$g(3,\{4\}) = \min_{k\in\{4\}}\{c_{34} + g(4,\{4\} - \{4\})\}$$

$$g(4,\{4\} - \{4\}) = 0$$

$$g(3,\{4\}) = c_{34}$$

$$g(2,\{3,4\}) = c_{23} + c_{34}$$

$$g(1,\{2,3,4\}) = c_{12} + c_{23} + c_{34}$$

Thus, we have the cost of one possible journey from point i through all the remaining points. By finding the minimum at each stage, an optimal tour will be created. As mentioned earlier, the DP and Brute force algorithms are differentiated in this model by how many sub tours are recorded. As such, although DP might be of a smaller order of time complexity, it is of a far greater order of space complexity.

3.3.2 Algorithmic Implementation

Using the points generated previously and running the brute force algorithm (with no recorded sub-tours), the code for which is in Appendix 7.3, we can see the optimal tour in Figure 4. We can see that the shortest possible distance is 2.947 units.

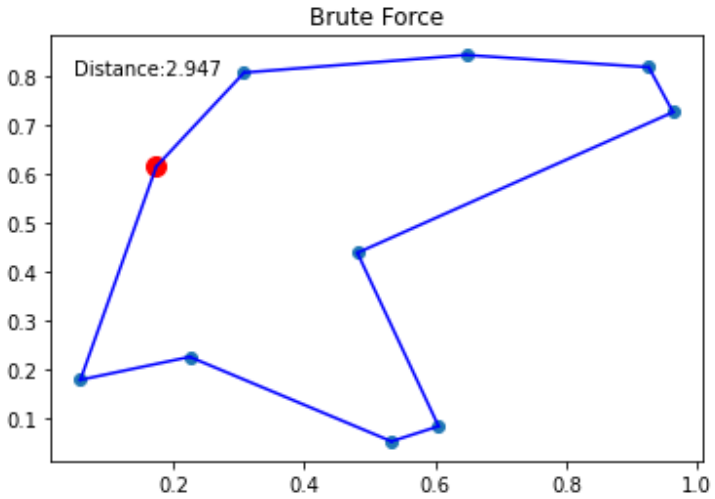


Figure 4; Brute Force solution to the randomly generated points

3.3.3 Brute Force Time Complexity

The time required for this algorithm is O(n!). Given a starting node, there are $n-1$ possible nodes to go to next. From each of those consequent nodes, there are $n-2$ possible nodes to go to next and so on. As the brute force algorithm iterates through every option, we can thus say that it is O(n!).

3.4 Nearest Neighbour Algorithm

The Nearest Neighbour Algorithm repeatedly adds the closest unreached node to the most recently reached node. When every city has been visited, it returns to the starting city. Nearest Neighbour belongs to the family of algorithms called the greedy algorithms. A greedy algorithm makes the locally optimal choice at every stage, using very little computing power, but often resulting in sub-optimal tours[12].

My implementation of the nearest neighbour algorithm is very simple. Given a starting point, the distance matrix is searched for the shortest distance between the starting point and one of the remaining points. This distance is added to the total and the next city is removed from the list of remaining cities. This process is repeated until there are no cities left, at which point the final distance back to the starting city is added. The program is shown in Appendix 7.4, with the output tour shown in Figure 5. The length of the tour is 3.275 units, 11% larger than the optimal tour.

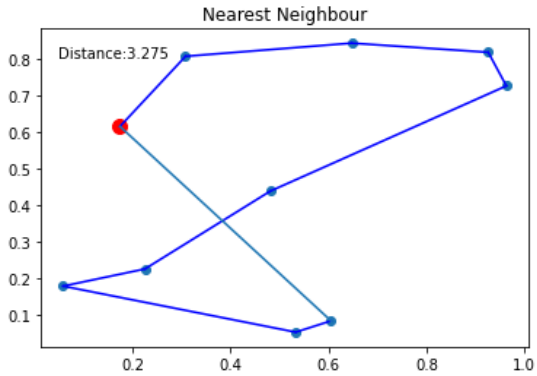


Figure 5; Nearest Neighbour solution to the randomly generated points

3.4.1 Nearest Neighbour Time Complexity

The worst-case time required for this algorithm is $O(n^2)$ shown by the following explanation of steps.

1. *Find the next nearest node by finding the shortest edge connecting to that node. This process takes $O(n)$ time as there are always $(n-1)$ edges connected to every node. Although edges will become unavailable as the tour progresses, that amounts to a subtraction of a constant factor.*
2. *Step 1 is repeated for every consequent node which requires n repeats. n iterations of $O(n)$ results in a time complexity of $O(n^2)$.*

3.4.2 Nearest Neighbour Tour Length Upper Bound

We can also find a certain bound for the worst-case ratio of the nearest neighbour tour to the optimal tour. First, some terms will need to be defined. l_i is the length of the i-th largest edge in the nearest neighbour tour. The line going to the red dot in Figure 6 is the longest line so, even though it is created last, it is l_1. The line labelled l_2 is created 5th, however it is the second longest line.

Optimal

[12] Black, Paul E. (2 February 2005). "greedy algorithm". *Dictionary of Algorithms and Data Structures*. U.S. National Institute of Standards and Technology (NIST).

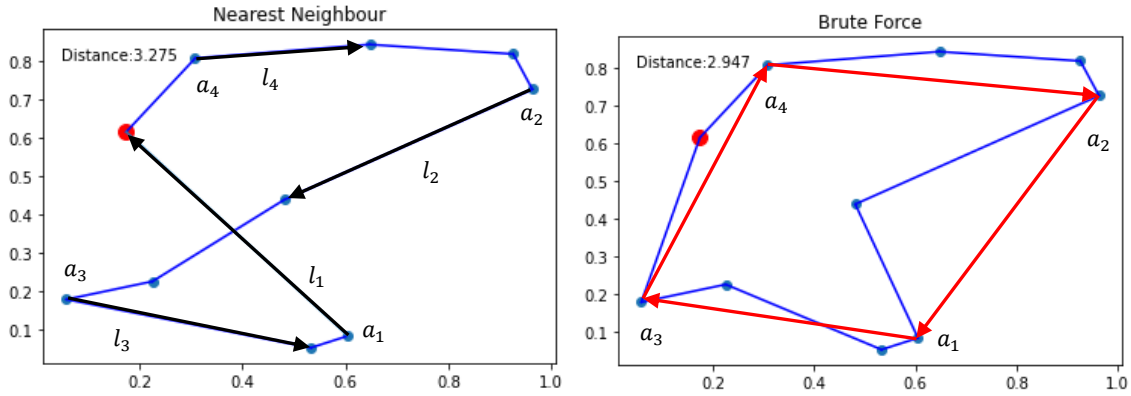


Figure 6; Labelled Nearest Neighbour tour and tour T for upper bound of NN algorithm

Furthermore, for each integer i, where $0 \leq i \leq n$, each point a_i, is the point from which the i-th longest edge was created by the nearest neighbour algorithm. In the nearest neighbour algorithm, each a_i corresponds to the node which the edge of length l_i comes from, so a_i precedes l_i.

We then create a subset of points according to the following rule:

All points a_i are included in the new subgraph for $1 \leq i \leq \min(2k, n)$, with some k that we can decide. The reasons for choosing this subset will be made clear later in the proof. The min() function takes the smallest of the two inputs. We can formally express this subset as shown below:

$$\{a_i \mid 1 \leq i \leq \min(2k, n)\}$$

If we had 10 cities, $n = 10$ and we took $k = 2$, then the subset would include points a_1, a_2, a_3 and a_4, where those points refer to the four edges which connect to the four longest lines in the optimal tour. They are shown on the left side of Figure 6.

We can create a new tour, called T which goes through the remaining nodes in the same order as they do in the optimal tour. From this point on, it is useful to think of every point a_i, as belonging to the new tour, T. For $k = 2$, in the example we have been using, T is displayed in red on the right side of Figure 6. This specific tour T will be referenced throughout; however, it should be noted that this is only for one specific value of k. For other values, different Ts will be created. Inevitably, as at least 0 nodes will be omitted and otherwise the order of nodes is the same, the distance between any two adjacent points in T, $\{a_i, a_j\}$, will be shorter than the equivalent distance in the optimal tour for the full set of points. As such we can make the following claim:

$$(2) \quad LENGTH(OPTIMAL) \geq LENGTH(T)$$

I will introduce a function here $d()$, where $d(a_i, a_j)$, is the distance between those points in T. Take a given pair of points $\{a_i, a_j\}$ that form an edge (are connected by a line) in the tour T. If a_i appears before a_j in the NN tour, then $d(a_i, a_j) \geq l_i$. If a_j appears before a_i in the NN tour, then $d(a_i, a_j) \geq l_j$. These statements will now be illustrated further. If a_i appears before a_j in the NN tour, there are two possible scenarios.

(1) The following NN node was a_j and so $d(a_i, a_j) = l_i$, as a_i is the distance from that node to the next nearest node.
(2) There were some other nodes between a_i and a_j, and so $d(a_i, a_j) > l_i$, as a_j was not the nearest point.

The same applies if a_j appears first. Take for example $\{a_1, a_3\}$ in Figure 6; a_3 appears before a_1. Therefore, in the NN tour, when a_3 was first added, a_1 was not in the tour. The algorithm then has two choices, either add a_1 next or add a different point first, depending on which is closest. As such, l_3, the line coming out of a_3 in the NN tour, must be equal to or shorter than $d(a_1, a_3)$ in tour T. Given this conclusion, we know that $d(a_1, a_3)$ is always longer or equal to at least one of $\{l_1, l_3\}$. We can now make the following statement:

$$(3) \quad d(a_i, a_j) \geq \min(l_i, l_j)$$

The distance between the points a_i and a_j in T is longer than the shorter of the two edges extending from a_i and a_j in the NN tour, l_i and l_j .

Now, we can add up all the edges in T (the red lines in Figure 6), where each edge is of the form (a_i, a_j).

$$(4) \quad LENGTH(T) \geq \sum_{(a_i, a_j)\ in\ T} \min(l_i, l_j)$$

There are the same number of edges as there are nodes. For every edge, the shortest of l_i and l_j is taken and so we can split up the sum to count by node rather than by edge. More specifically, for every node a_i, as it is connected to two edges, l_i is the possible solution to two different $\min(l_i, l_j)$ functions. We can count the number of times that l_i is the minimum solution by introducing a new value, α_i. α_i must always be between 0 and 2.

Thus, we can rewrite the equation above as shown in equation (5). The sum is now for each point a_i, not edge (a_i, a_j), and refers to the number of times l_i is chosen.

$$(5) \quad \sum_{(a_i, a_j)\ in\ T} \min(l_i, l_j) = \sum_{a_i\ in\ T} \alpha_i l_i$$

Moreover, as there are $\min(2k, n)$ edges and so $\min(2k, n)$ points, we know the following is true:

$$\sum_{a_i\ in\ T} \alpha_i = \min(2k, n)$$

Now as we are still trying to find what value $LENGTH(T)$ is bigger than, we must find the lower bound of the formula we found in (5). To do so, we need to assume that all the shortest lengths are chosen most often, and the longest lengths are chosen least often. As α_i can only ever be 0, 1 or 2, we should assume that $\alpha_i = 2$ for the lengths that are shorter and $\alpha_i = 0$ for the lengths that are the longer. We will as such need to split the array of lengths in half.

Therefore, $\alpha_i = 2$ for $k+1 \leq i \leq \min(2k, n)$ and $\alpha_i = 0$ for $i \leq k$. We can now create the equation below:

$$(6) \quad \sum_{a_i \, in \, T} \alpha_i l_i \geq 2 \sum_{i=k+1}^{\min(2k,n)} l_i$$

In the situation where n is smaller than $2k$, we will merely be decreasing the total size of the sum on the right and as we are trying to show that it is smaller than the sum on the left, this is not a problem. We have also pulled the 2 out in front of the sum as it is the same in every case.

We now have the following series of inequalities and equalities

$$LENGTH(OPTIMAL) \geq LENGTH(T) \geq \sum_{(a_i,a_j) \, in \, T} \min(l_i, l_j) = \sum_{a_i \, in \, T} \alpha_i l_i \geq 2 \sum_{i=k+1}^{\min(2k,n)} l_i$$

$$LENGTH(OPTIMAL) \geq 2 \sum_{i=k+1}^{\min(2k,n)} l_i$$

This now completes the first half of the proof and other than the formula above, we can abstract everything else away.

The next step is to sum each side of the inequality for values of k, when k is equal to powers of 2, $k = 2^j$, where $2^j \leq n$ and j is an integer. Therefore j can take any value between 0 and $\lceil \log_2(n) \rceil - 1$. Substituting $k = 2^j$ and taking the sum for values of j from 0 to $\lceil \log_2(n) \rceil - 1$, we can create the formula shown below.

$$\sum_{j=0}^{\lceil \log_2(n) \rceil - 1} LENGTH(OPTIMAL) \geq \sum_{j=0}^{\lceil \log_2(n) \rceil - 1} \left(2 \left(\sum_{i=2^j+1}^{\min(2^{j+1},n)} l_i \right) \right)$$

As the length of the optimal tour is constant, we can interpret the left term of the equation to be equal to the length of the optimal multiplied by the number of possible values of j, $\lceil \log_2(n) \rceil$. Effectively, we are adding the length of the optimal to the sum once for every integer value between 0 and $\lceil \log_2(n) \rceil$.

$$\sum_{j=0}^{\lceil \log_2(n) \rceil - 1} LENGTH(OPTIMAL) = \lceil \log_2 n \rceil \cdot LENGTH(OPTIMAL)$$

On the right-hand side, we can first pull out the factor of 2. A brief inspection and the testing of a few values of j starting from 0 reveals the following to be true:

$$2\sum_{j=0}^{\lceil \log_2(n) \rceil - 1}\left(\sum_{i=2^j+1}^{\min(2^{j+1},n)} l_i\right) = 2\left(\sum_{i=2}^{2} l_i + \sum_{i=3}^{4} l_i + \sum_{i=5}^{8} l_i + \cdots + \sum_{i=2^{j_{final}}+1}^{n} l_i\right) = 2\sum_{i=2}^{n} l_i$$

As such, we can put the two components back together and create equation (7).

$$(7) \quad \lceil \log_2 n \rceil \cdot LENGTH(OPTIMAL) \geq 2\sum_{i=2}^{n} l_i$$

We also have some other unused information. We know that the length of the optimal tour is at least twice as long as any edge in the graph. This is true because any tour must travel from one side of the edge to the other and then all the way back. According to the Triangle Inequality, travelling via other nodes can never produce a shorter route. Thus, we can create the equation below, which states that the length of the optimal tour is always longer than two times the longest edge in the tour.

$$(8) \quad LENGTH(OPTIMAL) \geq 2l_1$$

We can simply add together equations (7) and (8) as follows:

$$\lceil \log_2 n \rceil \cdot LENGTH(OPTIMAL) + LENGTH(OPTIMAL) \geq 2\sum_{i=2}^{n} l_i + 2l_1$$

Which simplifies to:

$$(9) \quad \frac{1}{2} \cdot (\lceil \log_2 n \rceil + 1) \cdot LENGTH(OPTIMAL) \geq \sum_{i=1}^{n} l_i = LENGTH(NN)$$

The sum of all l_i is by definition equal to the length of the Nearest Neighbour tour and so we have our upper bound. The Nearest Neighbour algorithm will produce a tour that is at most $\frac{1}{2} \cdot (\lceil \log_2 n \rceil + 1)$ times greater than the optimal tour[13].

This upper bound also happens to be true for the greedy algorithm. As the proof follows a similar structure as the above, I will omit it from this essay.

3.5 Greedy Algorithm

The Greedy algorithm is very similar to the nearest neighbour algorithm, being part of the family of greedy algorithms, however it is individually known by that same name. It follows a simple rule with two constraints. If the constraints are violated, then the chosen edge is discarded from the set of available edges.

---

[13] D. J. Rosenkrantz, R. E. Stearns and P. M. Lewis, "Approximate algorithms for the traveling salesperson problem," *15th Annual Symposium on Switching and Automata Theory (swat 1974)*, 1974, pp. 33-42, doi: 10.1109/SWAT.1974.4.

Rule 1: Find the shortest edge between any two points and add it to the tour.
Constraint 1: Any implementation of Rule 1 must not increase the number of edges connecting to a point to more than two.
Constraint 2: Any implementation of Rule 1 must not create a completed circuit which goes through fewer points than the full set of points.

As such, after the edges have been sorted by size, my greedy algorithm iterates through each entry in sorted list. The entry is checked to see whether its addition to the current chain would violate the two constraints above. If not, then it is added and the program proceeds to the next shortest edge. The program is shown in Appendix 7.5, with the output tour shown in Figure 7. The length of the tour is 3.11 units, 6% longer than the optimal tour.

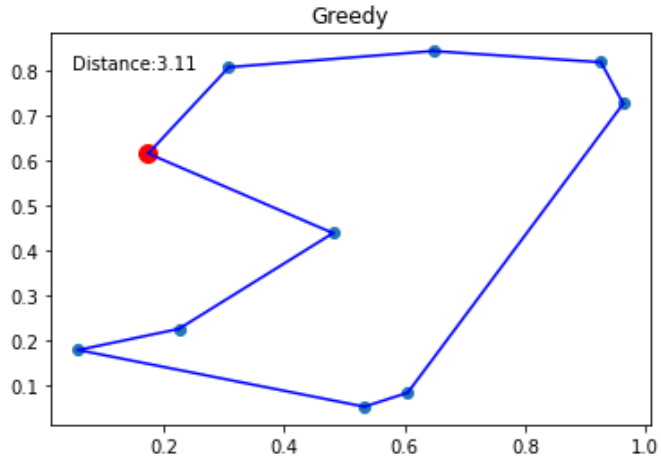


Figure 7; Greedy solution to the randomly generated points.

3.5.1 Greedy Time Complexity

The Time Complexity of the greedy algorithm is $O(n^2 \log_2 n)$. This becomes clear when we break down the algorithm into two steps.

1. *Order the list of distances by size, largest first. This takes $O(n^2 \log_2 n)$ time when using efficient algorithms such as merge sort.*
2. *Add edges 1 to n to the tour from the distance matrix. This takes $O(n)$ as the number of operations is linearly proportional to the number of nodes (all the constraint verification steps are abstracted away as time complexity refers to the worst-case asymptotic time required and, in this scenario, we are merely multiplying n by some constant).*

The second step can be ignored as it is performed in addition to the first step and is of a lesser order.

3.5.2 Greedy Tour Length Upper Bound

The upper bound for this algorithm is the same as the upper bound of the nearest neighbour algorithm as shown below:

$$\frac{1}{2} \cdot (\lceil \log_2 n \rceil + 1) \cdot LENGTH(OPTIMAL) \geq LENGTH(GREEDY)$$

4 Comparison of Algorithms

Now that we have explored some of the theory behind the three algorithms, it is time to discover how well they work in practice. I will use two sources of data for my comparisons, my own randomly generated points and node sets from Heidelberg University's TSPLIB[14]. While randomly generating points allows for a much larger sample size, when the number of nodes increases it becomes practically impossible to calculate the optimal tour. As the TSPLIB contains optimal lengths, it allows for the NN and greedy algorithms to be compared to the optimal for far larger numbers of nodes.

4.1 Randomized Node Sets

My first comparison involved testing the three algorithms on 1000 random repetitions of 10 nodes. My results are shown in Table 1.

Number of Nodes (1000 Repetitions)	Brute force total time /s	Nearest Neighbour total time /s	Greedy total time /s	NN Mean Distance Percentage Difference /%	Greedy Mean Distance Percentage Difference /%
10	563.590	0.008	0.325	11.0	7.8

Table 1; Randomised node set comparison to brute force (10 nodes / 1000 repetitions)

However, I wanted to see how the effectiveness and speed of the Nearest Neighbour and Greedy algorithms varied for different numbers of nodes. So, I compared them for 100 random repetitions of node sets from 10 to 200, with intervals of 10 as shown in Table 2. The important result for each node size was how much longer the Nearest Neighbour tour was than the Greedy tour. The time required is also important however it will likely deviate from expectations as my own implementation is being used instead of the ideal. Specifically, my constraint verification step seems to be of a greater order of time complexity.

Number of Nodes (100 Repetitions)	NN/Greedy distance ratio	Greedy/NN time ratio
10	1.038	39
20	1.055	103
30	1.046	219
40	1.061	378
50	1.057	564
60	1.059	788
70	1.048	1035
80	1.050	1278

[14] G. REINELT, ''TSPLIB—A traveling salesman problem library,'' ORSA J. Comput. 3 (1991), 376-384.

90	1.051	1690
100	1.061	1859
110	1.046	2024
120	1.056	2590
130	1.055	3630
140	1.051	3960
150	1.059	3923
160	1.054	4539
170	1.054	5079
180	1.051	6058
190	1.054	6627
200	1.055	7161

Table 2; Randomised node set comparison between Greedy and NN (10-200 nodes / 100 repetitions)

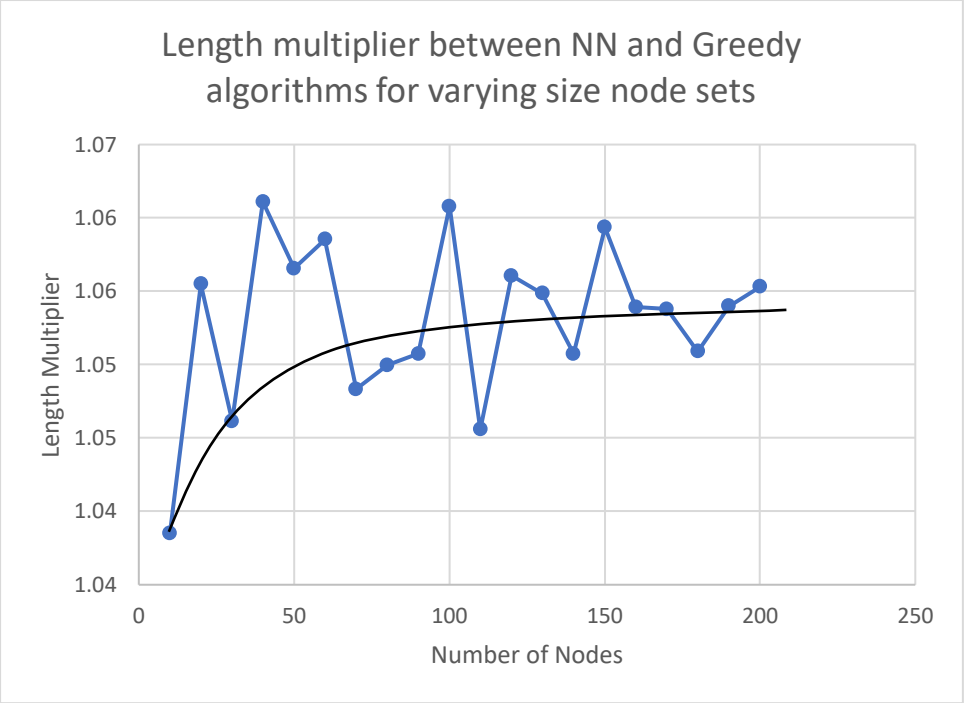


Figure 8; Geometric relationship between number of nodes and difference in length between NN and Greedy tours.

The time ratio between the Greedy and NN algorithm is displayed in Appendix 7.6 in Figure 11.

4.1.1 Results Analysis

The first thing to note is that the Greedy algorithm tends to find shorter tours than the NN algorithm, as shown in Figure 8; this difference increases at a decreasing rate. However, the NN algorithm was many times faster than the Greedy algorithm and the ratio of the times required kept increasing as the number of nodes increased. This is anomalous however can be explained by rationales such as the one given previously.

4.2 TSPLIB Node Sets

4.2.1 Introduction

Heidelberg University has a number of 'solved' node sets in their TSPLIB as a way for researchers to benchmark their efforts. However, the question arises of how optimal solutions are found for node sets in the thousands when the brute force method flounders for any number of nodes above 30. A wide variety of algorithms and approaches are used with one key difference to what I have explored above, specifically improvement algorithms. The Greedy and NN algorithms might provide a good starting point, but options such as the k-opt algorithm and simulated annealing allow for local maxima to be achieved and the tour can be 'kicked' until eventually the global maximum is found. How do we know whether we have found the global maximum? Linear Programming. Its implementation effectively creates a lower bound for the length of the optimal tour. One important stage of linear programming in solving TSPs is establishing linear inequalities/constraints that push up the lower bound. This method can be visualised by viewing the nodes as points on a multi-dimensional graph and by taking many half spaces, we can create a convex hull. In two dimensions this would be like placing a rubber band around a set of pegs; in three dimensions, it would be more like shrink wrapping an object[15]. For the 48 states in the US mainland node set, there are 2,199,023,254,648 required constraints[16] and so it becomes unfeasible to compute them all. However, certain constraints are far more powerful than others and in the 48 states example, 9 constraints can define the entire problem and can be used to find the lower bound. If our solution equals the lower bound, then we know we have the optimal solution.

4.2.2 Algorithmic Implementation

I randomly chose 10 node sets[17] which had sizes smaller than 200 and tested my two algorithms on them, attaining the results in Table 3. One example is shown in Figure 10.

Node Set	Node count	Optimal distance	NN distance	Greedy distance	NN time	Greedy time
ulysses22	22	1.835	2.174	2.169	0.000034	0. 003496
bayg29	29	3.945	4.44	4.298	0.000044	0.00789
att48	48	4.319	5.221	5.174	0.000114	0. 057917
berlin52	52	4.336	5.161	5.721	0.000137	0. 072533
st70	70	6.786	8.055	9.006	0.000196	0. 199118
kroA100	100	5.382	6.79	6.412	0.000405	0.876226
rd100	100	8.056	10.124	8.959	0.000383	0.901403
lin105	105	4.659	6.596	5.593	0.000401	0.622475
ch150	150	9.338	11.714	11.175	0.000795	5.062136
gr202	202	7.205	8.114	7.793	0.001551	15.486101

Table 3; Results of testing Greedy and NN algorithms on TSPLIB node sets.

The mean distance ratio (NN distance : Greedy distance) = 1.032 : 1 (4sf)

[15] Cook, William(2014). *In Pursuit of the Travelling Salesman*, Chapter 5, Princeton University Press
[16] Cook, William(2014). *In Pursuit of the Travelling Salesman*, Chapter 6, Princeton University Press
[17] G. REINELT, ''TSPLIB—A traveling salesman problem library,'' ORSA J. Comput. 3 (1991), 376-384.

Mean Distance Percentage Difference of Greedy Tour from Optimal Tour = 19.0 (3sf)

Mean Distance Percentage Difference of NN Tour from Optimal Tour = 22.1 (3sf)

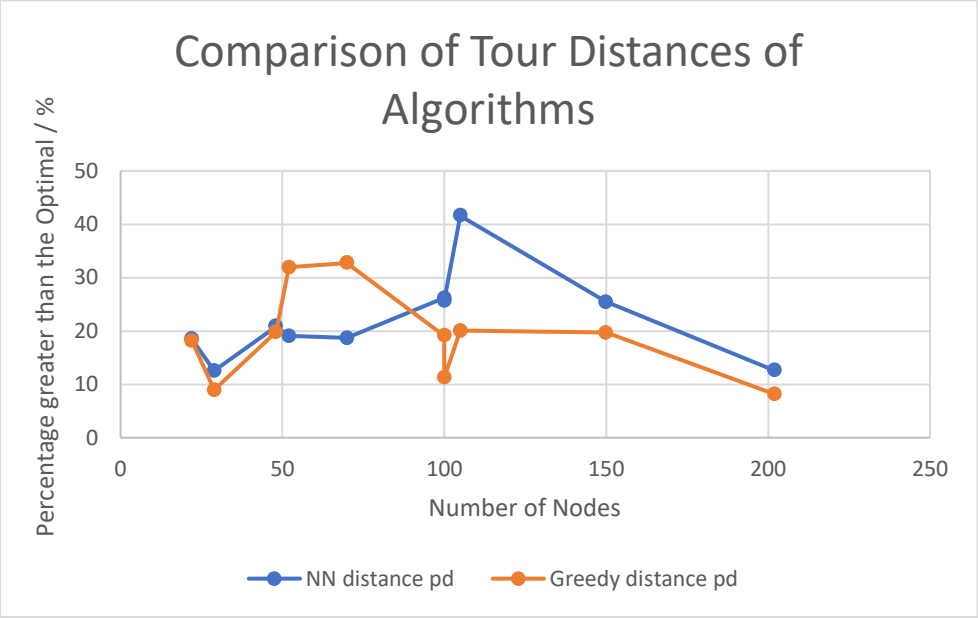


Figure 9; Comparison of tour distances of Greedy and NN algorithms with respect to the optimal for TSPLIB node sets.

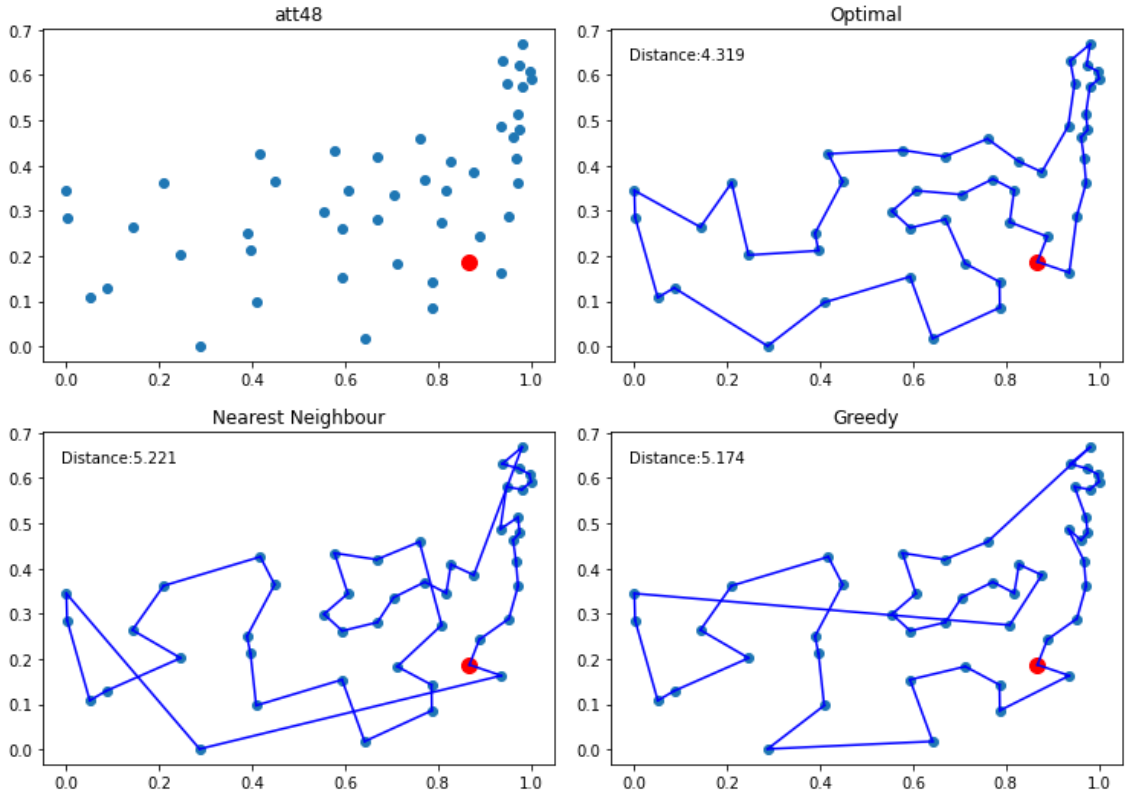


Figure 10; Optimal and algorithmic tours of the 48 states of mainland USA.

4.2.3 Results Analysis

The distances produced by, and the times required for the algorithms matches the pattern from the randomised node sets. So, their relative performance is corroborated. In comparison to the optimal tour, the algorithms produce relatively similar lengths with mean PDs at 19.0% and 22.1%. As such, the Greedy algorithm will likely be marginally better for smaller node sets as the tour it produces will probably be shorter, however, for larger node sets, the NN algorithm is superior as the time required for the Greedy algorithm becomes inhibitive.

We can input the number of cities and the length of the tour to find the upper bound for the Greedy and NN algorithms:

$$\frac{1}{2} \cdot (\lceil \log_2 n \rceil + 1) \cdot 4.319 = 14.22$$

The tours produced by both the Greedy and NN algorithms are substantially below the upper bound given above.

5 Conclusion

The TSP can teach everyone a very important lesson; sometimes finding the optimal solution is not the best. As we have seen in the examples above, any optimal finding algorithm is completely unfeasible for large numbers of nodes. Seeing as the TSP has many applications such as finding the fastest route for a laser to follow when travelling between points on a circuit board, of which there can be tens of thousands, those methods are simply not very valuable. As such, the question changes to whether we can find a solution that is very close to optimal. The answer is a resounding yes, with thousands of studies examining the different algorithms that can be used. This essay focused on the Greedy and Nearest Neighbour Algorithms, determining that the solutions they provide can never be worse than $\frac{1}{2} \cdot (\lceil \log_2 n \rceil + 1)$ times the length of the optimal tour. Some random sampling of node sets unveiled a certain insight into the two algorithms: the NN algorithm is faster but less accurate. The speed difference becomes emphasised the greater the number of nodes with the greedy algorithm becoming untenably slow. These algorithms however are both undominated: no competing heuristic algorithm finds better tours and runs more quickly[18]. Overall, these two algorithms are relatively simple to implement and form a strong starting basis for the use of any improvement algorithms if a much more accurate result is required.

[18] E. H. L. Aarts and J. K. Lenstra (eds.), *Local Search in Combinatorial Optimization*, John Wiley and Sons, London, 1997, pp. 215-310

6 Bibliography

- Cook, William(2014). *In Pursuit of the Travelling Salesman*, Princeton University Press
- En.wikipedia.org. 2021. *Hamiltonian path - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Hamiltonian_path>
- Austinmohr.com. 2021. [online] Available at: <http://www.austinmohr.com/Work_files/complexity.pdf>
- En.wikipedia.org. 2021. *Insertion sort - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Insertion_sort#Best,_worst,_and_average_cases>
- Cobham, Alan (1965). "The intrinsic computational difficulty of functions". *Proc. Logic, Methodology, and Philosophy of Science II*. North Holland.
- 'A dynamic programming approach to sequencing problems', Michael Held and Richard M. Karp, *Journal for the Society for Industrial and Applied Mathematics* 1:10. 1962
- Johnson, D. S., L. A. McGeogh. 2002. In: G. Gutin, A. Punnen, eds. *The Traveling Salesman Problem and Its Variations. Kluwer*, Boston, MA. 369-443.
- Bari, A., 2021. *4.7 [New] Traveling Salesman Problem - Dynamic Programming using Formula*. [video] Available at: https://www.youtube.com/watch?v=Q4zHb-Swzro
- Black, Paul E. (2 February 2005). "greedy algorithm". *Dictionary of Algorithms and Data Structures*. U.S. National Institute of Standards and Technology (NIST).
- D. J. Rosenkrantz, R. E. Stearns and P. M. Lewis, "Approximate algorithms for the traveling salesperson problem," *15th Annual Symposium on Switching and Automata Theory (swat 1974)*, 1974, pp. 33-42, doi: 10.1109/SWAT.1974.4.
- G. REINELT, ''TSPLIB—A traveling salesman problem library,'' ORSA J. Comput. 3 (1991), 376-384.
- E. H. L. Aarts and J. K. Lenstra (eds.), *Local Search in Combinatorial Optimization*, John Wiley and Sons, London, 1997, pp. 215-310

(225 words)

7 Appendix

7.1 Appendix 1: Randomized node set

```
Random_x =      [0.174, 0.647, 0.058, 0.533, 0.963, 0.225, 0.308, 0.926, 0.605, 0.481]
Random_y =      [0.615, 0.843, 0.178, 0.052, 0.726, 0.225, 0.807, 0.818, 0.083, 0.438]
```

```
Distance_matrix =  [0.0, 0.525, 0.452, 0.668, 0.797, 0.393, 0.234, 0.779, 0.685, 0.354]
                   [0.525, 0.0, 0.888, 0.799, 0.337, 0.748, 0.341, 0.28, 0.761, 0.438]
                   [0.452, 0.888, 0.0, 0.491, 1.058, 0.173, 0.677, 1.078, 0.555, 0.497]
                   [0.668, 0.799, 0.491, 0.0, 0.799, 0.353, 0.788, 0.861, 0.078, 0.389]
                   [0.797, 0.337, 1.058, 0.799, 0.0, 0.892, 0.66, 0.099, 0.736, 0.561]
                   [0.393, 0.748, 0.173, 0.353, 0.892, 0.0, 0.588, 0.918, 0.406, 0.333]
                   [0.234, 0.341, 0.677, 0.788, 0.66, 0.588, 0.0, 0.618, 0.783, 0.408]
                   [0.779, 0.28, 1.078, 0.861, 0.099, 0.918, 0.618, 0.0, 0.802, 0.585]
                   [0.685, 0.761, 0.555, 0.078, 0.736, 0.406, 0.783, 0.802, 0.0, 0.376]
                   [0.354, 0.438, 0.497, 0.389, 0.561, 0.333, 0.408, 0.585, 0.376, 0.0]
```

(126 words)

7.2 Appendix 2: Auxiliary Functions

```
import random
import matplotlib.pyplot as plt

# Random Point Generator
def RPG(num_points):
    # initializes lists of point coordinates
    random_x = []
    random_y = []

    # adds a random number between 0 and 1 to each list until they are big enough
    for i in range(0, num_points):
        random_x.append(round(random.random(),5))
        random_y.append(round(random.random(),5))

    return random_x, random_y

# Create Distance Matrix
def CDM(x,y):
    # initializes distance matrix list
    distance_matrix = []

    # finds the distance between every pair of points, adding it to the matrix
    for i in range(0,len(x)):
        row = []
        for j in range(0,len(x)):
            distance = ( (x[i]-x[j])**2 + (y[i]-y[j])**2)**(0.5)
            row.append(round(distance,5))
        distance_matrix.append(row)

    return distance_matrix

# Draw Circuit
def DC(order, total, x, y, title):
    # create a figure with just the points
    plt.figure()
    plt.scatter(x,y)
    plt.scatter(x[0],y[0], c = 'red', s = 100)
    plt.title(title)

    if (title == 'Greedy') or (title == "Nearest Neighbour") or (title
                                == "Brute Force") or (title == "Optimal"):
        # for every point in the order
        for i in range(0, len(order)-1):
            # draw a line between that point and the next one
            plt.plot([x[order[i]], x[order[i+1]]],[y[order[i]], y[order[i+1]]],
                                                        color = 'blue')
        plt.plot([x[order[-1]], x[order[0]]], [y[order[-1]], y[order[0]]])
        plt.annotate('Distance:' + str(round(total, 3)), xy=(0, 1), xytext=(12, -12),
                     va='top',xycoords='axes fraction', textcoords='offset points')
```

7.3 Appendix 3: Brute Force Function

```
def bf_algorithm(distance_matrix):

    # adds the nodes 1 to (n-1) to a list
    remaining_nodes = []
    for i in range(1,len(distance_matrix)):
        remaining_nodes.append(i)

    # initialisation of variables
    current_node = 0
    current_shortest = 1000000
    distance = 0
    current_order = [0]
    best_order = []

    # iterative function which iteratively adds and removes nodes
    def find_shortest_node(current_node, remaining_nodes, distance_matrix,
                    current_shortest, distance, current_order, best_order):
        # when we descend to the next tier, the "current" node becomes the previous
        previous_node = current_node

        # condition in the function currently has a full tour
        if len(remaining_nodes) == 0:
            # adds the final distance from the end node to the initial node
            distance += distance_matrix[current_node][0]
            current_order.append(0)
            # checks to see if this tour is shortest than all previous tours
            if distance < current_shortest:
                current_shortest = distance
                best_order = current_order.copy()
            # removes the last 0 from the order as the function will now move on
            current_order.pop(-1)
            return current_shortest, current_order, best_order

        # calls the function again for every next possible node
        for current_node in remaining_nodes:
            new_list = remaining_nodes.copy()
            current_order.append(current_node)
            distance += distance_matrix[previous_node][current_node]
            # removes the next selected node from list of remaining nodes
            new_list.remove(current_node)

            # iterates the function again "descends a tier"
            current_shortest, current_order, best_order = find_shortest_node(current_node,
            new_list, distance_matrix, current_shortest, distance, current_order, best_order)

            # removes the last node from the order so the next can be checked instead
            current_order.pop(-1)
            # removes the associated distance
            distance -= distance_matrix[previous_node][current_node]

        return current_shortest, current_order, best_order

    # calls the main function
    total, ignore, node_order = find_shortest_node(current_node, remaining_nodes,
            distance_matrix, current_shortest, distance, current_order, best_order)
    return node_order, total
```

7.4 Appendix 4: Nearest Neighbour Function

```
def nn_algorithm(distance_matrix):

    # adds the numbers 1 to (n-1) to a list
    remaining_nodes = []
    for i in range(1,len(distance_matrix)):
        remaining_nodes.append(i)

    # initialisation of variables
    current_node = 0
    total = 0
    node_order = [0]

    # finds the next nearest node
    def find_nearest_neighbour(current_node, remaining_nodes, distance_matrix,
                               total, node_order):
        minimum = 2
        # for each node which has not been connected
        for next_node in remaining_nodes:
            # find the distance between this node and that node
            distance = distance_matrix[current_node][next_node]
            # if the distance is the smallest so far, record it as the smallest
            if distance < minimum:
                minimum = distance
                min_node = next_node
        # establushes the next node in the order
        node_order.append(min_node)
        remaining_nodes.remove(min_node)
        current_node = min_node
        total += minimum
        return total, remaining_nodes, node_order, current_node

    # finds and adds the next nearest node until there are no nodes remaining
    while len(remaining_nodes) >= 1:
        total, remaining_nodes, node_order, current_node = find_nearest_neighbour(
            current_node, remaining_nodes, distance_matrix, total, node_order)

    # adds the final distance back to the starting city
    total += distance_matrix[current_node][0]
    node_order.append(0)

    return node_order, total
```

7.5 Appendix 5: Greedy Function

```
from TSP_Auxilliary import RPG, CDM, DC

def greedy_algorithm(distance_matrix):
    # changes the distances from node i to node i to 2 instead of 1
    def remove_zeros(distance_matrix):
        for row in range(0,len(distance_matrix)):
            for item in range(0,len(distance_matrix)):
                if row == item:
                    distance_matrix[row][item] = 2
        return distance_matrix

    # lists the edges by size order, shortest first
    def order_distances(matrix):
        ordered_list = []
        for row in range(len(matrix)):
            for item in range(len(matrix)):
                ordered_list.append([matrix[row][item],row,item])
        ordered_list.sort()
        for i in range(len(ordered_list)):
            ordered_list[i] = [ordered_list[i][1],ordered_list[i][2],
                               ordered_list[i][0]]
        return ordered_list

    distance_matrix = remove_zeros(distance_matrix)
    ordered_list = order_distances(distance_matrix)

    total = 0
    pairs = []
    # appearance count refers to how often each node appears in the current edges
    appearance_count = []
    for i in range(len(distance_matrix)):
        appearance_count.append(0)

    # function that adds edges until a full tour is created
    chain_filled = False
    while chain_filled == False:
        # ordered list items are of form [node i, node j, size]
        edge_stats = ordered_list[0]
        appearance_count[edge_stats[0]] += 1
        appearance_count[edge_stats[1]] += 1
        ordered_list.pop(0)
        ordered_list.pop(0)
        dummy_pairs = pairs.copy()
        current_pair = edge_stats[0:2].copy()
        dummy_pairs.append(current_pair)
        current_node = current_pair[1]
```

```
        # Start of Constraint Validation Section

        # given the current node in the chain, this function finds the next node
        def find_next_pair(dummy_pairs, current_node, current_pair):
            finished = False
            for item in dummy_pairs:
                if finished == False:
                    if current_node == item[0]:
                        current_node = item[1]
                        current_pair = item.copy()
                        finished = True
                    elif current_node == item[1]:
                        current_node = item[0]
                        current_pair = item.copy()
                        finished = True

            return current_pair, finished, current_node

        chain = [edge_stats[0], edge_stats[1]]
        count = 1

        # creates the chain of edges connected to the latest added edge
        while count < len(chain):
            dummy_pairs.remove(current_pair)
            current_pair, finished, current_node = find_next_pair(dummy_pairs,
                                                   current_node, current_pair)
            if finished == True:
                chain.append(current_node)
            count += 1

        # checks if the chain forms a complete tour and whether it is a full tour
        valid = True
        if chain[-1] == chain[0]:
            if len(chain) < len(distance_matrix)+1:
                valid = False

        # checks that every node has more than 2 edges
        chain_filled = True
        for item in appearance_count:
            if item == 3:
                valid = False
            if item != 2:
                chain_filled = False

        # End of Constraint Validation Section

        # if the new edge was valid, it is implemented
        if valid == True:
            pairs.append(edge_stats[0:2])
            total += edge_stats[2]

        # otherwise it undoes the latest addition
        else:
            appearance_count[edge_stats[0]] -= 1
            appearance_count[edge_stats[1]] -= 1

    return chain, total
```

7.6 Appendix 6: Time Ratio Chart for Greedy and NN

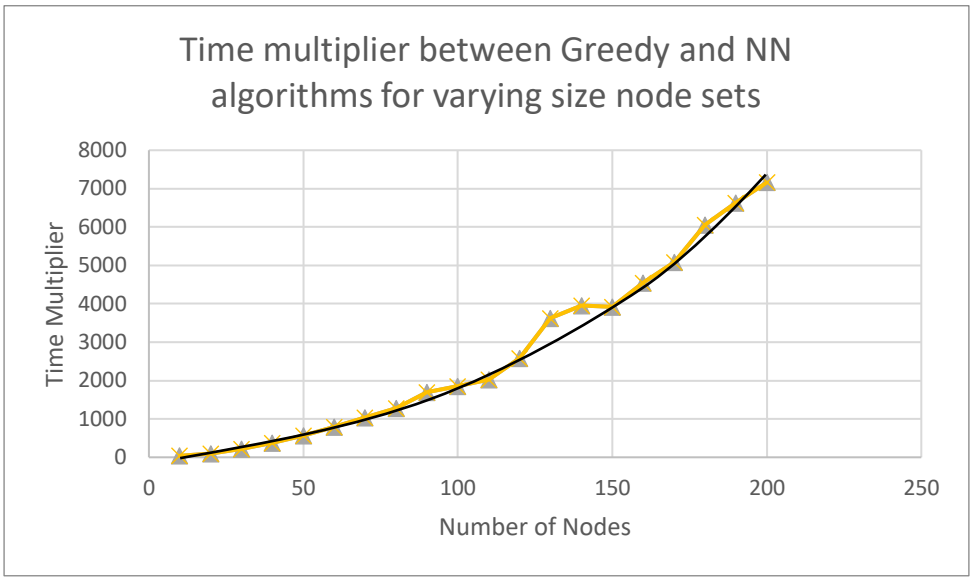


Figure 11; Geometric relationship between number of nodes and difference in time between NN and Greedy tours.