UCC

University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

University College Cork

AM4065

Travelling Salesman Problem

Barry O'Donnell

December 2021

Abstract

The Travelling Salesman Problem (TSP) is a problem focused on computational optimization methods and algorithms. In this paper we will be analyzing algorithms used to find solutions for TSP networks. The algorithms we will be focusing on are; Brute Force Algorithm, Dijkstra's Algorithm, Nearest Neighbour, k-Nearest Neighbour and Ant Colony Optimization. We will also analyse two more 'algorithms' that were a product of my own mind. We will study the cost and order of each method, and ultimately analyse the effectiveness of each algorithm.

Contents

Chapter 1

Introduction

1.1 Background

Consider a salesman who has many clients in many different cities. To be a successful businessman, it is essential that he must visit each city to meet his client. To minimize his own expenses, the client ponders the question; what is the minimum amount of travel time I can partake in and visit each city only once and get home in time for dinner.

The simple question gives rise to an interesting conundrum and what is most commonly referred to as the Travelling Salesman Problem (TSP). We can interpret this problem as; 'given a set of nodes connected by weighted edges and starting from a selected node: what is the path around the network which visits each node only once, minimizes the path cost, and ends at the starting node?'

First analysis of the problem originates from W. R. Hamilton's *Icosian Games*[1]. This was a mathematical game where the objective was to find the path around a network that visited each node exactly once. Such a path would become known as a Hamiltonian path. In many of these analyses, it is assumed that the network is fully connected (each node is connected to every other node by an edge) with no self loops (a node does not have an edge which leaves and connects back to itself). Each edge is also weighted and directionless (the weight of an edge between two nodes can be compared to the 'distance' between two 'cities'). This is referred to as the symmetric TSP.

This problem of finding the shortest Hamiltonian path was first approached by Kantorovich and Koopmans [2] and was inspired by what we now refer to as 'last-mile delivery.' This is a scenario in which goods have reached a sort of final delivery centre and now must be delivered to a set of customers. Solving to find the shortest path to reach each customer and return to the delivery centre would minimize travel expenses.

The advancement in linear programming and Dantzig's [3] development of the simplex method enabled one to obtain a solution for TSP within a reasonable amount of computational time. When the theory of NP-Completeness was developing, TSP would become one of the first problems to be proven NP-Hard by Karp [4], meaning that TSP can be solved in polynomial time.

TSP has been optimized using many different methods since Kantorovich's day. Dantzig's simplex method still works quite well today, but people have sought solutions using migration patterns [5] and by modelling ant colony behaviours [6]. In the present day, TSP is seen more as a benchmark for optimization algorithms, but has applications in last-mile delivery logistics and molecular biology [7].

In this paper we will approach TSP using multiple algorithms. These models are deterministic and will always output the same result for the same network. These algorithms are; Brute Force (BF), Dijkstra's algorithm, Nearest Neighbour (NN) and k-Nearest Neighbour (k-NN). We will also look at one non-deterministic model, Ant Colony Optimization (ACO), which depends on random walks. We will also look at two naive algorithms, resulting from modifications of the previously mentioned algorithms.

First and foremost, we must have a strict definition of what a network is, and how we are interpreting the TSP.

1.2 Construction of a Network

Consider a network $\mathcal{G}$ consisting of a set of nodes v_i connected by edges e_{ij}, where edge e_{ij} connects nodes v_i and v_j. In an unweighted network (all edges leaving a node have equal cost), if nodes v_i and v_j are connected we say they are adjacent or $a_{ij} = 1$. Define the adjacency matrix $\mathbf{A}$ of a fully connected network as,

$$\mathbf{A} = \left\{a_{\mathbf{ij}}\right\}_{\mathbf{i,j=1,\dots,n}} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \tag{1.1}$$

where n is number of nodes present in the network. In an undirected network $a_{ij} = a_{ji}$. In a fully connected, unweighted, undirected network $a_{ij} = 1$, for all values $i \neq j$ and $a_{ii} = 0$ for all values $i = 1, ..., n$ (no self loops, node v_i is not connected to itself).

In this particular scenario, TSP can be solved quite simply as all edges have the same weight, and taking any Hamiltonian path is a viable solution. TSP becomes prominent when we consider the weighted matrix $\mathbf{W}$. In a undirected, weighted network, each edge e_{ij} is assigned some real value that indicates the cost of traversing said edge. Define the weighted matrix $\mathbf{W}$ as,

$$\mathbf{W} = \left\{w_{\mathbf{ij}}\right\}_{\mathbf{i,j=1,\dots,n}} = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nn} \end{pmatrix}, \tag{1.2}$$

where $w_{ij} \in \mathbb{R}_{>0}$ is the weight of e_{ij}. In a network with no self-loops, weights w_{ii} for all values $i = 1, ..., n$ is considered to be null, or infinite. We can visualise this as the edge connecting a node to itself simply costs too much to travel upon in any given scenario.

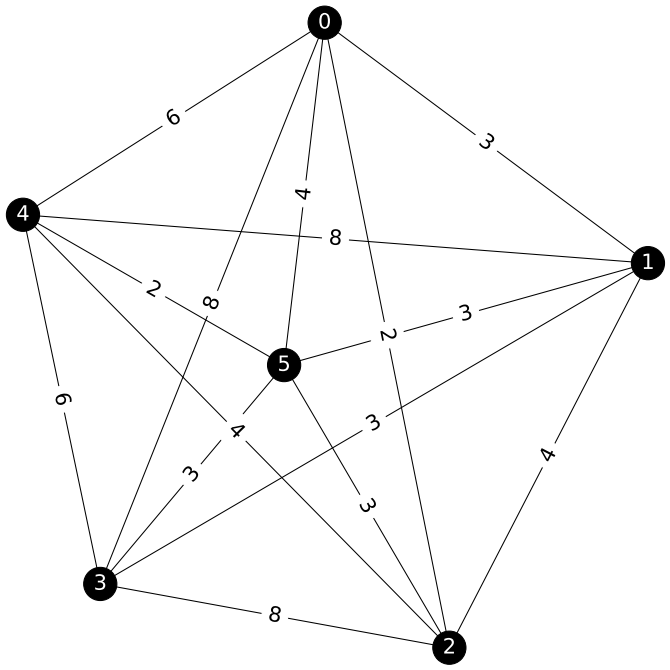


Figure 1.1: A weighted network with $n = 6$ nodes. Here edge $e_{3,4}$ has weight $w_{3,4} = 6$. For simplicity in coding, labelling of nodes starts with 0

Now that we have defined the network as a matrix, we can approach TSP as a way of minimizing a path through a matrix. To analyse our algorithms, we will be using `Python`. In particular the packages we will be using are; `numpy` [8] to accomplish the matrix mathematics, `itertools` [9] to obtain the permutations necessary for BF and k-NN, `matplotlib` [10] to create figures, and `networkx` [11] to visualise networks.

Chapter 2

Finding the shortest path

2.1 Brute force algorithm (BF)

The most simple approach to this problem is the brute force method. This algorithm is as follows:

Step 1: Draw fully connected weighted network $\mathcal{G}$ with n nodes, such that $\{v_i\}_{i=1,...,n} = \{1, 2, ..., n\}$.

Step 2: Choose one node to be the starting node for all the paths. Since all paths contain every node, which node is chosen as the starting node is a matter of preference. For simplicity, we will choose v_1 as the starting node.

(Aside: Steps 1 and 2 are the same for all algorithms and will be assumed in future algorithm descriptions.)

Step 3: Choose some permutation of the list of numbers $\{2, 3, ..., n\}$. Connect the starting node to the first value in the permutation v_j, where v_j is some node in the network

Step 4: With your current node, connect it to the following value in the permutation.

Step 5: Repeat Step 4 until you have reached the end of the permutation. Once reached, connect your final node to v_1, and calculate the cost of the path by summing the weights of the edges connecting each node.

Step 6: Repeat Steps 3 - 5 until you have successfully attempted each permutation, and find the minimum cost of the values you have obtained

Clearly this method can become computationally expensive as n increases. We notice that for a network $n = 8$ nodes, there are 5040 permutations, while for a network $n = 12$ nodes there are $39,916,800$ permutations. The amount of permutations is $(n-1)!$ if we choose one node to start from and so worst case computation time of BF is of order $\mathcal{O}((n-1)!)$.

Estimating from Figure 2.1, the computational time to find the shortest path for a network with 14 nodes would be $\sim$ 19 hours. Extrapolating further, at 80 nodes, using BF may take upwards of 10^{100} years, meaning that the BF on 80 nodes would not be finished until after the heat death of the universe.

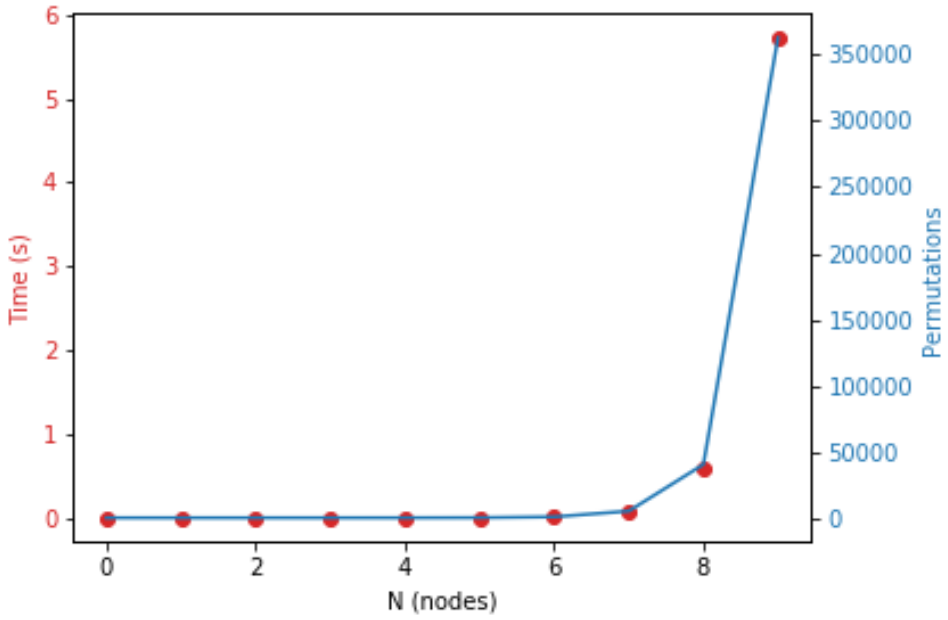


Figure 2.1: Time to compute shortest path for n nodes, overlaid on the factorial of n

It should be clear that this method lacks any form of efficiency, but will always return the correct shortest path.

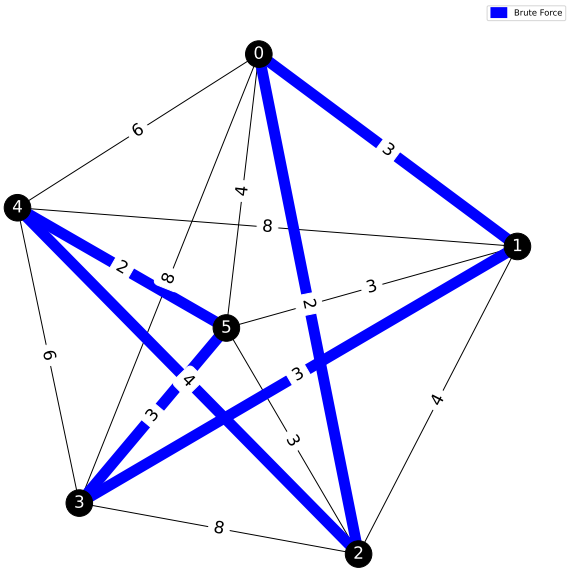


Figure 2.2: The BF path through the network drawn in Figure 1.1. The path is $\{0, 1, 3, 5, 4, 2, 0\}$ and has cost 17

2.2 Dijkstra's Algorithm

Dijkstra's algorithm is a slight improvement on BF by simply negating the need to look any further than the shortest path [12]. BF will search every path, meaning it will search for every path longer than shortest path too. Dijkstra's algorithm can be visualised as a sort of 'radius' search. Consider your starting node as the centre of the circle. As we depart from the centre, our displacement from the centre can be thought as the distance we have travelled. With each step further we can reach more and more nodes along certain paths. This algorithm is completed once we reach a distance where one path is finished.

The algorithm works as follows;

Step 1: 'Step' away from your starting node, increasing your total cost by your step length.

Step 2: If your cost is greater than the nearest unvisited node along some path, connect that node back to the centre.

Step 3: Repeat Step 1 - 2 until you have obtained one fully connected path.

The worst case computational time of Dijkstra's algorithm is $\mathcal{O}\left(n(n-1)+n\log(n)\right)$ [13].

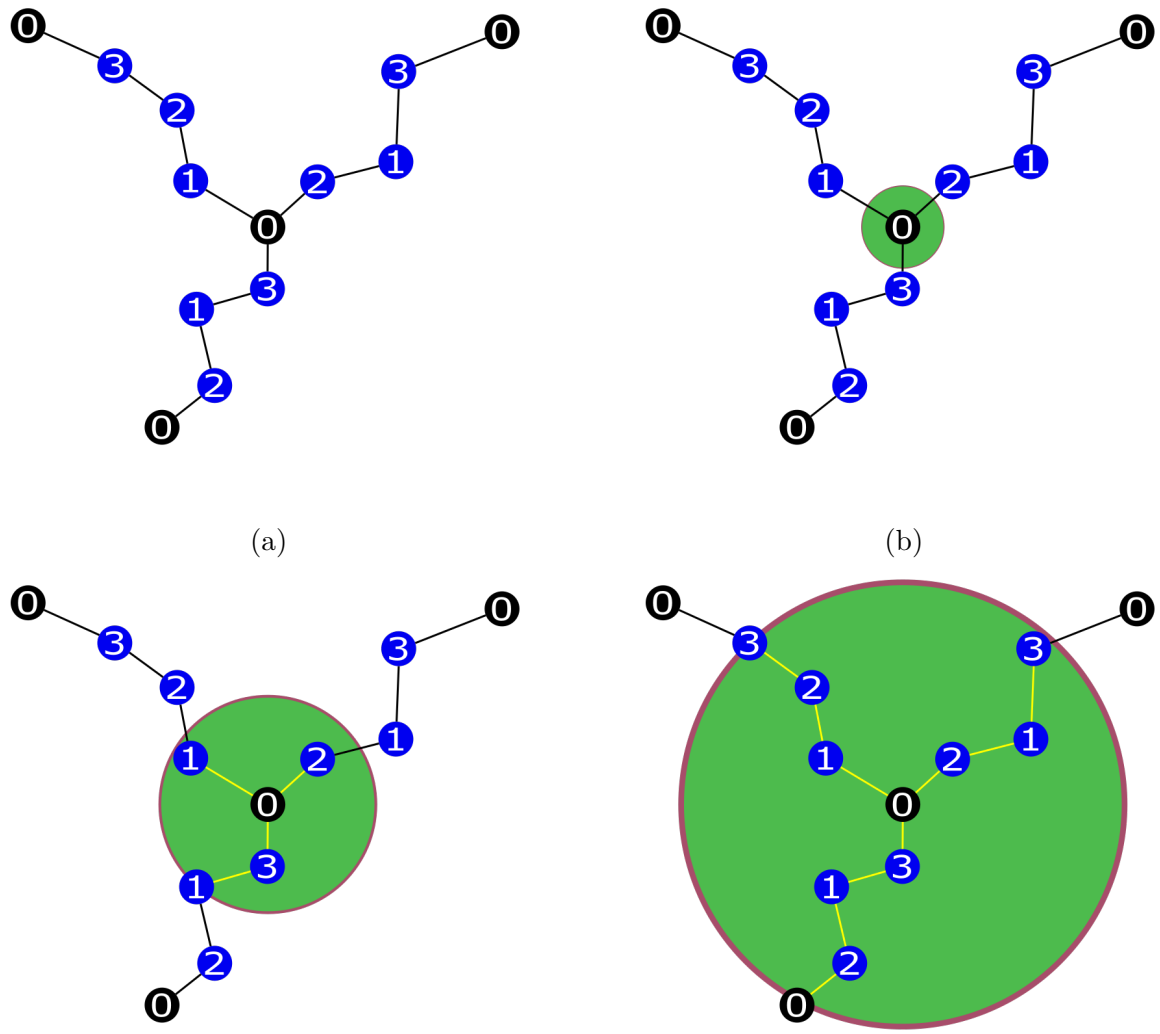


Figure 2.3: A visual example of Dijkstra's method. In this 4 node network, we can visualise it as our starting node with all possible paths around the network sprouting from it. The 'distance' from the node along one path to the centre represents the cost of traversing that path to that node from the starting node. In 2.3b we take our first step out from the starting node. 2.3c we continue our journey until we can reach our first few nodes. We notice that before we even reach the second node for two of the paths, we have reached the second node of the bottom path. Despite travelling the same 'distance' from the starting node, we have reached more nodes in one path compared to others. Continuing on, 2.3d shows that the bottom path $\{0, 3, 1, 2, 0\}$ is the shortest path, and we can stop the algorithm.

2.3 Nearest-Neighbour algorithm (NN)

The next algorithm applies a simple optimization, albeit a naive one. The process is as follows;

Step 1: Choose the minimal $\{w_{1j}\}_{j=2,\dots,n}$ and move to node v_j.

Step 2: Choose the minimal $\left\{w_{jk}\right\}_{k=\{\text{Remaining nodes}\}}$. Move to node v_k.

Step 3: Repeat Step 2 until each node has been visited once. Sum up the weights of the path created to obtain the cost of the path.

This NN algorithm offers a costly compromise. It is a computational cheap algorithm, but it is guessing that the shortest path is the path of the shortest immediate connections with no foresight. This has the added benefit of taking a low amount of time to compute. In comparison to BF, NN needs only to look at it's immediate neighbours, which can be a max of $(n-1)$ nodes (the amount of nodes available to the starting node). The worst case computational time for such an operation will be $\mathcal{O}(n)$ (we include the calculation of returning to the beginning).

We will analyse later in Section 2.7 that this algorithm performs better than random guessing, but perhaps we can go one step further. Consider instead that rather than running this algorithm on just the initial node v_i, we run the algorithm once for each node in the network, and then choose the minimal path found. Since each node will be seeking a path through n amount of nodes, this operation has worst case computation time of $\mathcal{O}(n^2)$. This is a surprisingly effective method as we will see.

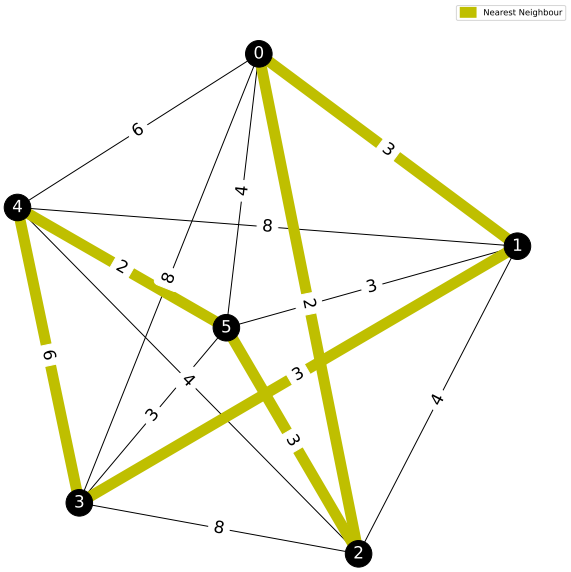


Figure 2.4: The NN path through the network drawn in Figure 1.1. The path is $\{0,2,5,4,3,1,0\}$ and has cost 19

Perhaps a better algorithm could be achieved with another addition; foresight.

2.4 k-Nearest neighbour algorithm (k-NN)

Similar to NN, but now we offer the algorithm to look k amount of steps ahead to choose the shortest immediate path.

Step 1: Choose $\{w_{1\to m}\}_{m=2,\dots,n}$, where m is a node which is k-steps away from node v_1 ($v_1 \to v_j \to \cdots \to v_m$ or $v_{1\to m}$), which minimizes the cost of the all paths k-steps away from your initial node.

Step 2: Move to v_j.

Step 3: Choose $\left\{w_{j\to l}\right\}_{l=\{\text{Remaining nodes}\}}$, where v_l is an unvisited node k-steps away from v_j. The path connecting these two nodes is $(v_j \to v_p \to \cdots \to v_l)$

Step 4: Move to v_p.

Step 5: Repeat Step 3 - 4 until you have constructed a path around the network. Sum up the weights of the path created to obtain the cost of the path.

Note that this algorithm is a sort of an amalgamation of NN and BF. Setting $k = 1$, we see that the algorithm behaves the exact same as NN, taking the immediate shortest path and having no foresight. As k tends to n, it becomes more and more similar to BF, becoming fully BF at $k = n$. Therefore the performance of k-NN depends heavily on the size of k.

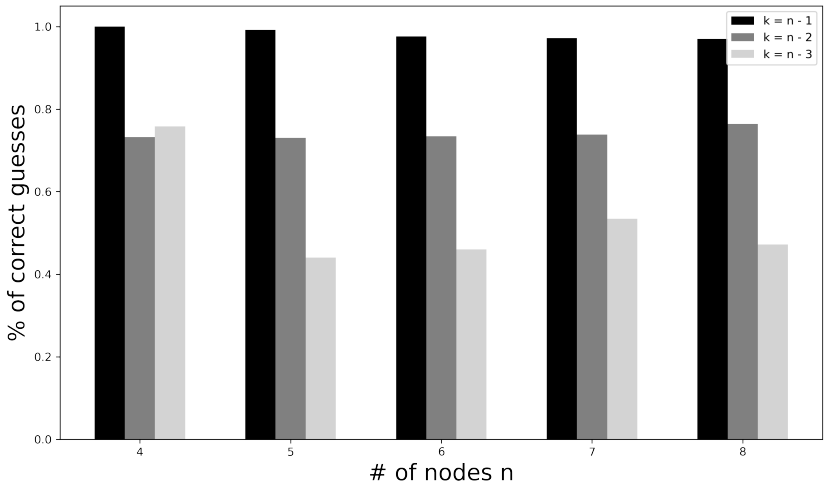


Figure 2.5: Comparing different models of k-NN, varying k

In Figure 2.5, we see a breakdown of the performance of k-NN for various values of k with respect to the number of nodes n. We see that the performance is quite good. We can obtain the correct path over 90% of the time by removing one node from BF, and over 70% when we remove two. It appears that with each decrement away from n there appears to be about a 20% reduction in correct guesses from each algorithm.

Analysing the computational complexity for this algorithm we can see that it must generate k-sized permutations each iteration. Therefore the worst case computational time is $\mathcal{O}(k!)$.

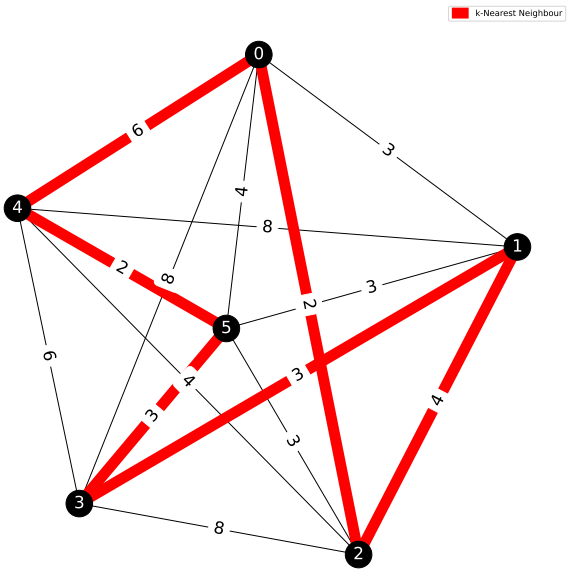


Figure 2.6: The k-NN ($k = 3$) path through the network drawn in Figure 1.1. The path is $\{0, 2, 1, 3, 5, 4, 0\}$ and has cost 20.

Similarly to using multiple NN algorithms, the same can be done with k-NN. This can make this algorithm considerably expensive for values of k close to n. The worst case computation time is $\mathcal{O}(n(k!))$ when using multiple k-NN algorithms.

In Figure 2.7 we have BF, NN, and k-NN ($k = 3$) displayed on the same network. Interestingly, it appears that although k-NN can look 3 steps ahead, NN finds a shorter path through the network.

Now we will discuss a non-deterministic algorithm. The previous few algorithms will always determine the same answer from the same network, but the next algorithm introduces a more natural development of finding the shortest path.

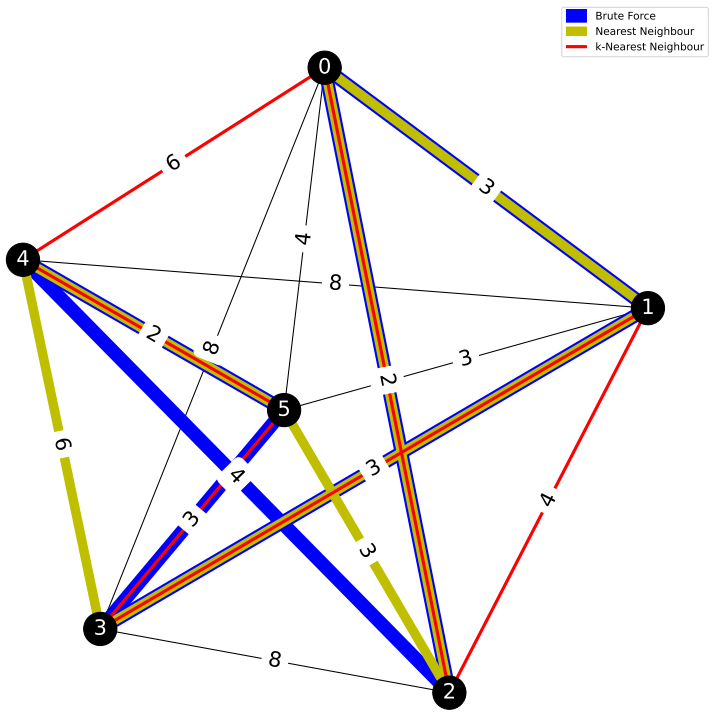


Figure 2.7: A comparison of BF, NN and k-NN. Despite a simple 6 node network, these algorithms produced very different paths.

2.5 Ant Colony Optimization (AC)

Though our problem might be steeped in a realistic setting of a travelling salesman, perhaps a solution could be found by having a change of view. Consider an ant colony. Thousands of ants each hour of each day are sent out on a mission to gather food and resources for the colony. They do this in a hivemind sort of sense with no sort of individuality. This is achieved using a clever little system.

For sake of explanation, imagine this ant colony was newly built, already populated by an army of ants.

Step 1: Ant 'A' departs from the colony in search of food. With each step, it leaves a pheromone trail indicating that 'I am leaving the colony in this direction!' The path will be winding and weaving, trying to cover as much ground as possible, leaving no stone left unturned.

Step 2: There are two outcomes from this;

1. Ant A finds food for the colony! They now return along their original path, following the pheromone while dispersing a stronger pheromone during the return.
2. Ant A is not so fortunate and continues searching until it can no longer search. It's original pheromone trail will disperse and the path will disappear.

Step 3: In the first scenario, once ant A returns to the colony, a strongly scented path will now link the colony to the food.

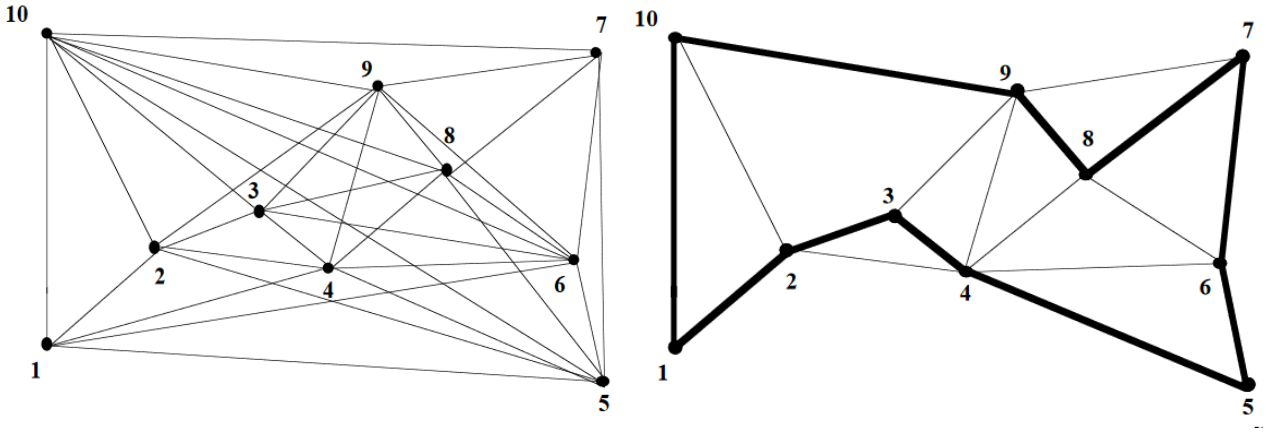
Step 4: Future generations of ants will travel along A's initial path, perhaps choosing randomly to cut and streamline the path, to shorten the distance from the colony to the food source. A path that is shorter will have a pheromone trail that will be stronger.

Step 5: After some time (supposing that the food source is still available) this path will have been optimized, minimalising the distance from colony to food .

This process of minimalisation of paths is where our interest lies.

Consider our original problem. We choose a starting position and send out an ant to travel through the network. When the ant reaches a node, it randomly chooses the next node to travel to. Once the ant has completed its tour about the network it leaves a 'trail' or a measure of how short the path was (we can imagine this by supposing that the pheromone loses its pungency over time). Now consider instead we have done this with ten or a hundred ants all simultaneously keeping track of how long their path was.

Now the next generation of ants departs from the starting position. We now have a map of pheromones over the network, and it appears that some edges in the network have higher pheromone scents on them, incentivising the ants to instead travel down this edge moreso than any other edge currently available. With each generation, the pheromone trail of the shortest path should begin to appear, as the ants will want to minimise the time spent away from the colony.



(a) (b)

Figure 2.8: From [14]. 2.8a; Initial trail distribution on network. 2.8b; Trail distribution after 100 generations. We notice that there are still some edges that are being used, but the shortest path has become apparent around the network.

A problem may arise in this optimization though. 'Stagnation' is possible and can be easy to fall into if the parameters are not selected carefully [6]. This could be because the ants may be reluctant to search for new paths, the pheromone of a trail may be too strong or maybe the pheromone trail doesn't evaporate quick enough. Tweaking these parameters to fit the problem can result in quite a powerful optimization, allowing for solutions to networks with large amount of nodes in polynomial time. (Aside; I believe this video [15] offers a good visual

insight into AC and how it works. It is more grounded in 2-d space rather that a network, but an intuition can be built from the examples examined.)

2.6 More abstract approaches to TSP

These algorithms are more of a result of mishandling code, but I believed that they are worthy of their own space.

2.6.1 Into-the-Fray algorithm (ItF)

Named after the feeling of diving headfirst into a problem with no feeling of regret. It is a mutation of k-NN, wherein rather than moving carefully through the network one step at a time, it leaps right into the fray, believing itself to be on the right track!

Step 1: Choose $\{w_{1\to j}\}_{j=2,\dots,n}$, where j is a node which is k-steps away from node v_1 ($v_1 \to v_x \to \cdots \to v_j$ or $v_{1\to j}$) which minimizes the cost of the all paths k-steps away from your initial node.

Step 2: Jump to v_j along the path $v_{1\to j}$.

Step 3: Choose $\left\{w_{j\to l}\right\}_{l=\{\text{Remaining nodes}\}}$, where v_l is an unvisited node k-steps away from v_j. The path connecting these two nodes is $(v_j \to v_p \to \cdots \to v_l)$

Step 4: Jump to v_p along the path $v_{j\to l}$.

Step 5: Repeat Step 3 - 4 until you have leaped a path around the network.

This algorithm makes not such good use of its foresight power. Similar to k-NN, it is bounded by BF as k tends to n and NN as k tends to 1. Despite this, it's performance is truly stammering. It on occasion barely outperforms NN despite being able to look multiple steps ahead. It seems to mature as k tends to n.

I guess with great power comes great responsibility.

2.6.2 If-it-ain't-broke algorithm (IiaB)

This algorithm utilises something akin to solving hedge mazes. It goes as follows

Step 1: From your initial node, say v_i, move to v_j where $j = i + 1$.

Step 2: From v_j move to v_k, where $k = j + 1$

Step 3: Repeat Step 2 ad nauseum until you reach v_n. Calculate the cost of your path.

This algorithm guarantees a solution, but not necessarily a good one. It has a $2/((n-1)!)$ chance to select the correct path. This is similar to the method of escaping hedge mazes wherein the participant simply places their left hand on the maze wall and holds it there until they have found their way out. It may not be the shortest, but it is a solution.

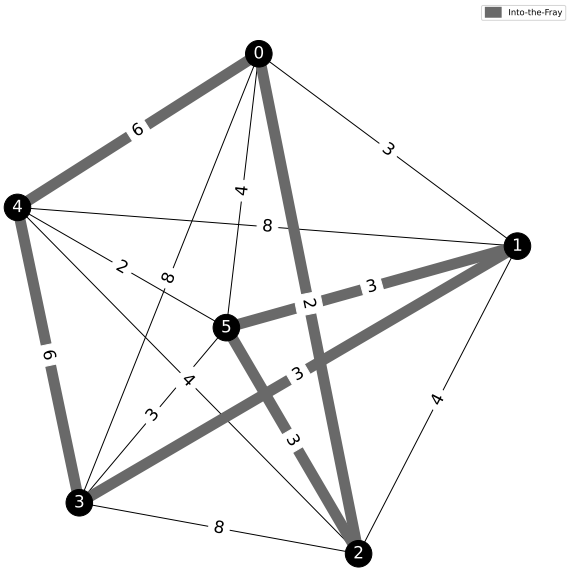


Figure 2.9: The Into-the-Fray algorithm ($k = 2$). The path is $\{0, 2, 5, 1, 3, 4, 0\}$ and has cost 23.

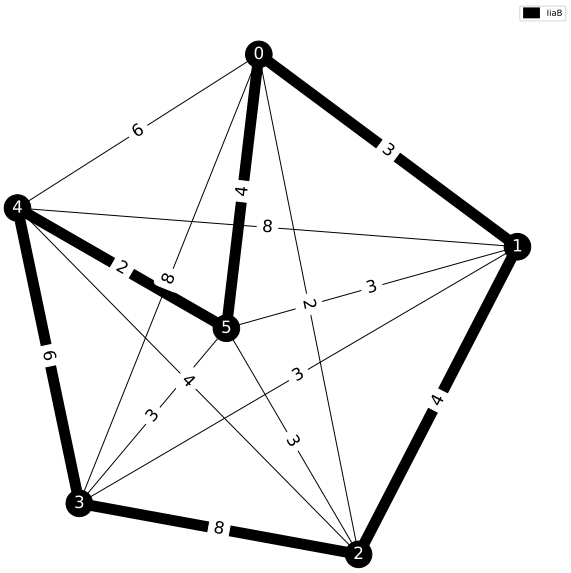


Figure 2.10: The If-it-ain't-Broke algorithm. The path is $\{0, 1, 2, 3, 4, 5, 0\}$ and has cost 27.

2.7 Analysis

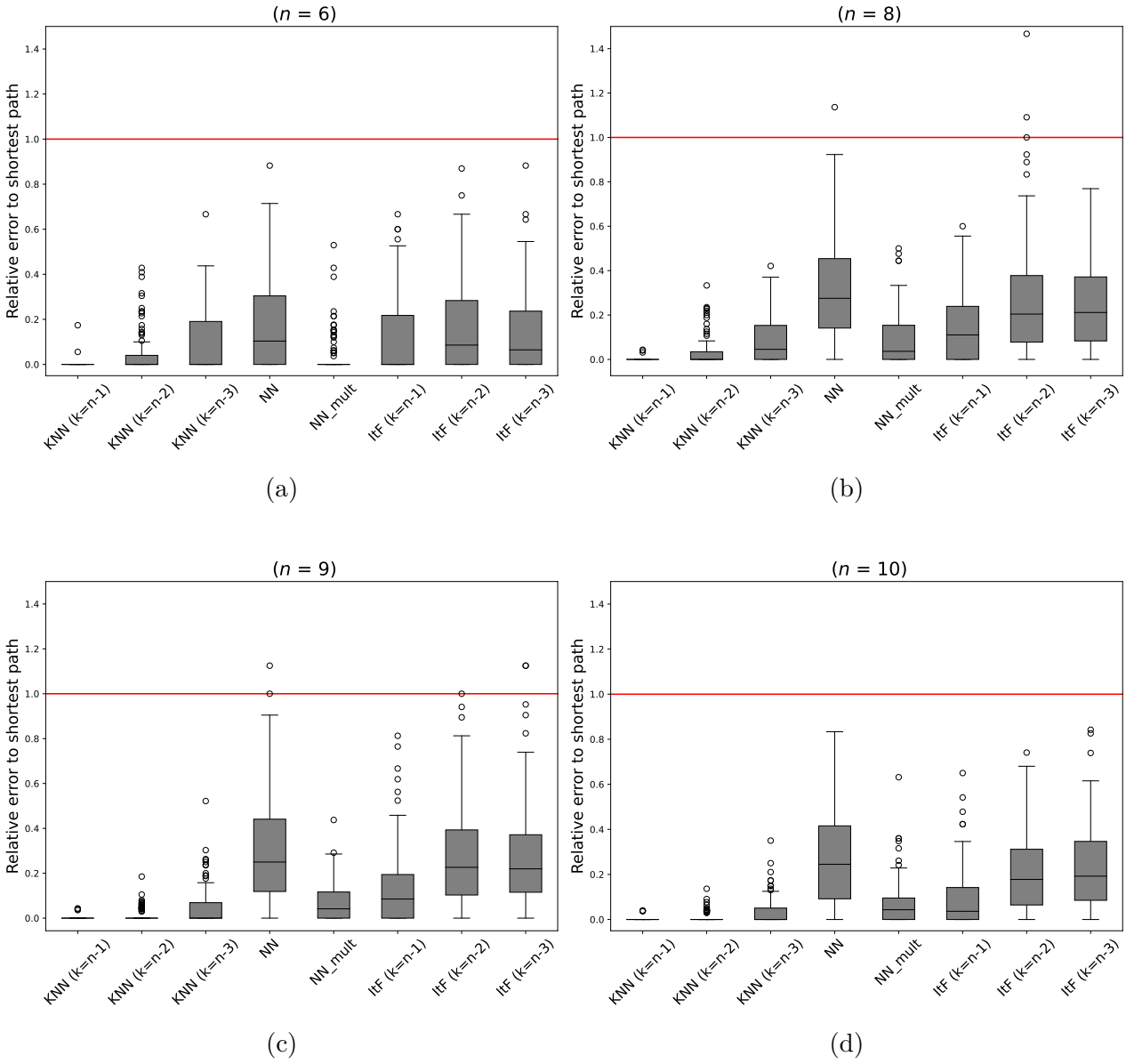


Figure 2.11: Error distribution of various models. This data was obtained from randomly generated networks after 100 trials for each $n = 6, 8, 9, 10$. The red line in each graph indicates where the relative difference is 1, or when the shortest path found by the algorithm was twice the length of the true shortest path.

From Figure 2.11 we see a comparison of error in shortest path length for various models. The analysis was achieved by randomly generating a network with n nodes, applying the BF algorithm to find the ground truth shortest path in the network, and comparing the length of that path to the other algorithms shortest paths. This process was repeated 100 times for each n. n was ranged from 4 up to 10, but only $n = 6, 8, 9, 10$ will be analysed in this section for sake of brevity (figures for $n = 4, 5, 7$ may be found in A.1).

It's clear that the k-NN algorithm with $k = n - 1$ will perform the best in all cases, having

the minimum variance for each n. This does come at a cost of being the most computationally expensive of all the algorithms. For $k = n - 2$, the computational cost is reduced by a factor of $(n - 2)$ and still performs quite well in comparison to its more careful relative. $k = n - 3$ similarly performs quite well, but lags behind the other two k-NN algorithms. The relative error for each of these algorithms is no more than 20% on average.

NN's naive method of locating the shortest path is surprisingly effective. Though it performs the worst overall, when compared to IiaB in Figure 2.12, it appears to do better than just guessing a path. IiaB consistently guesses paths that are significantly longer than the true shortest path. It appears that this small optimization is worth the cost. Interestingly, when we use multiple Nearest Neighbours on the network, it becomes surprisingly effective, holding itself well in comparison to k-NN's $k = n - 3$. This makes it quite effective for a moderate amount of nodes. As the amount of nodes increases, we might expect the effectiveness of this method to decrease. We can counteract this by introducing multiple k-NN with increasing k values as n increase (recall that k-NN when $k = 1$ is NN). For example, when $n = 20$, we would expect multiple NN to become less effective, so we could utilise multiple k-NN where $k = 3$. Further analysis could be done on when these break off points are reached.

ItF has an interesting outcome. It performs better than NN and worse than k-NN, but at the same computational cost as k-NN. It should be understood that this algorithm was a mistake.

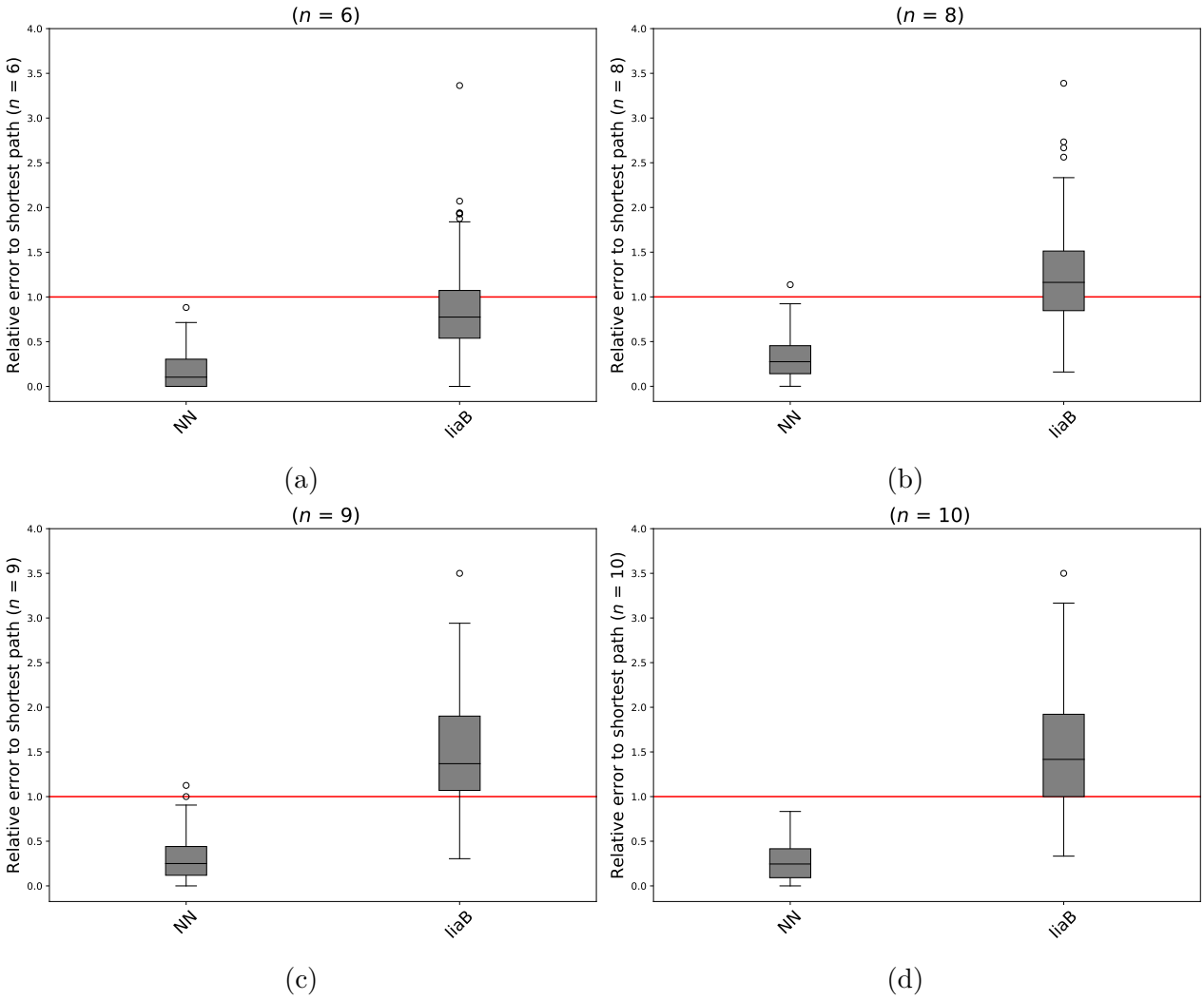


Figure 2.12: Comparison of NN to IiaB

Chapter 3

Conclusion

The Travelling Salesman Problem is a computational optimization problem. In this paper we discussed mostly deterministic algorithms, but there are a wide variety of non-deterministic algorithms similar to that of the Ant Colony Optimization. The scope of this problem is mostly in the eye of the beholder; I personally went in seeking a challenge of my coding skills, and to see the many uses of networks in optimization algorithms. Despite this seemingly naive scope, it seemed to be a never ending tunnel of optimization methods, slowly swallowing me up until I ultimately had to draw the line. I had simply spent too many nights dreaming of networks.

The Travelling Salesman Problem deserves a more thorough examination than the one offered in this paper. Perhaps observing more non-deterministic algorithms or Dantzig's Simplex Method. An interesting analysis would be to observe at what point does multiple k-NN out perform NN in obtaining shortest path length with respect to computational time.

It is fascinating seeing the ingenuity of scientists who create these seemingly abstract solutions to the Travelling Salesman Problem, but it's even more fascinating when they work.

Bibliography

[1] J. Dalgety, *Sir William Hamilton's Icosian Game and Traveller's Dodecahedron Puzzle.* [Online]. Available: https://www.puzzlemuseum.com/month/picm02/200207icosian.htm (visited on 10/18/2021).

[2] A. Schrijver, "On the history of combinatorial optimization (till 1960)," in *Discrete Optimization*, ser. Handbooks in Operations Research and Management Science, K. Aardal, G. Nemhauser, and R. Weismantel, Eds., vol. 12, Elsevier, 2005, pp. 1–68. DOI: https://doi.org/10.1016/S0927-0507(05)12001-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0927050705120015.

[3] G. B. Dantzig, "Reminiscences about the origins of linear programming," *Operations Research Letters*, vol. 1, no. 2, pp. 43–48, Apr. 1982, ISSN: 0167-6377. DOI: 10.1016/0167-6377(82)90043-8.

[4] M. Jünger, G. Reinelt, and G. Rinaldi, "Chapter 4 The traveling salesman problem," *Handbooks in Operations Research and Management Science*, vol. 7, no. C, pp. 225–330, Jan. 1995, ISSN: 0927-0507. DOI: 10.1016/S0927-0507(05)80121-5.

[5] M. Ergezer and D. Simon, "Oppositional biogeography-based optimization for combinatorial problems," *2011 IEEE Congress of Evolutionary Computation, CEC 2011*, pp. 1496–1503, 2011. DOI: 10.1109/CEC.2011.5949792.

[6] M. Dorigo and L. M. Gambardella, "Ant colonies for the travelling salesman problem," *Biosystems*, vol. 43, no. 2, pp. 73–81, Jul. 1997, ISSN: 0303-2647. DOI: 10.1016/S0303-2647(97)01708-5.

[7] J. Jones and A. Adamatzky, "Computation of the travelling salesman problem by a shrinking blob," *Natural Computing*, vol. 13, no. 1, pp. 1–16, Mar. 2014. arXiv: 1303.4969.

[8] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.

[9] G. Van Rossum, *The Python Library Reference, release 3.8.2.* Python Software Foundation, 2020.

[10] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.

[11] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Jan. 2008. [Online]. Available: https://www.osti.gov/biblio/960616.

[12] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik 1959 1:1*, vol. 1, no. 1, pp. 269–271, Dec. 1959, ISSN: 0945-3245. DOI: 10.1007/BF01386390. [Online]. Available: https://link.springer.com/article/10.1007/BF01386390.

[13] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987, ISSN: 0004-5411. DOI: 10.1145/28869.28874. [Online]. Available: https://doi.org/10.1145/28869.28874.

[14] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: Optimization by a colony of cooperating agents. ieee trans syst man cybernetics - part b," *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society*, vol. 26, pp. 29–41, Feb. 1996. DOI: 10.1109/3477.484436.

[15] S. L. Lague, *Coding adventure: Ant and slime simulations*, Mar. 2021. [Online]. Available: https://www.youtube.com/watch?v=X-iSQQgOd1A (visited on 12/06/2021).

Appendix A

Supplementary Code

A.1 GitHub

All code used for this project can be accessed at:
https://github.com/BarryOD4/AM4065-TSP/blob/main/TSP_algorithms.ipynb