The Long Road: Comparison of Approximation Heuristics for Traveling Salesperson Problem

Daniel Chen
Carleton College
Northfield, USA
chend2@carleton.edu

Miles Hoene-Langdon
Carleton College
Northfield, USA
hoenem@carleton.edu

Arthur Viegas Eguia
Carleton College
Northfield, USA
viegaseguiaa@carleton.edu

Artem Yushko
Carleton College
Northfield, USA
yushkoa@carleton.edu

***Abstract*—The Traveling Salesperson Problem is a fundamental problem in graph theory. The problem statement says "given a set of cities, what is the shortest possible route visiting each city exactly once before returning to the initial city". As it is a NP-hard problem, there are no known algorithms that solve it in polynomial time. However, different approximation algorithms can generate results to some degree of accuracy and satisfactory computational efficiency. In this study, we implemented and analyzed three algorithms for approximating solutions to the Traveling Salesperson Problem: the Nearest Neighbor heuristic, the Smallest Insertion heuristic, and the Christofides algorithm. The first two are fast greedy approaches, that are conceptually simple and easy to implement but generate solutions which are far from optimal. The third, Christofides algorithm, is close to state of the art, yields a solution within 1.5 times the optimal distance but has a higher computational cost. We evaluated these algorithms on a number of datasets, including country maps, electrical grids, protein interaction networks, and synthetic data. Our analysis compared the solution accuracy and runtime of each heuristic across graphs ranging from 10 to 16,862 nodes. The results offer insight in how each approximation algorithm fares in terms of computational cost and accuracy, helping make more informed decisions in problem solving.**

***Index Terms*—TSP, Christofides Algorithm, Greedy Algorithms**

I. Introduction

The Traveling Salesperson Problem (TSP) poses the question, 'Given a list of cities and the distance between each pair of cities, what is the shortest route that visits each city exactly once and returns to the origin city?' [1]. This problem assumes the existence of a graph where every node is a city. Every edge connects two cities and has as its weight the distance between the cities it is connecting. Moreover, this problem assumes that the graph is fully connected (there is an edge connecting every pair of nodes).

The TSP is a classic optimization problem and one of the most extensively studied problems in the field. It is used in logistics, microchip manufacturing, and DNA sequencing, as an example [2] [3]. These real-world problems can be reduced to the TSP, allowing us to build solid solutions. Moreover, the TSP is used as a benchmark for many optimization methods. [4]

As it is an NP-hard problem [5], there are no known efficient ways to solve it. Nevertheless, many known algorithms and heuristics approximate the problem's solution in polynomial time. A heuristic is an algorithm that does not provide the most optimal solution to a problem but gives one that is "good enough" as an approximate solution [6]. Furthermore, for some approximation algorithms (heuristics), there exists a bound for how far the answer given is to the optimal answer. Additionally, as it is an intensively studied problem, there are instances of large graphs that have been solved entirely or whose solution was approximated within a small fraction of the optimal (such as 1%) [7]. The TSP is different from the Hamiltonian Path Problem, as the Hamiltonian Path asks whether a path that connects every node in a graph exists. Meanwhile, in the TSP, we assume such a path exists; after all, the graph is fully connected. Moreover, there are multiple such paths. Our goal in the TSP is to find the Hamiltonian Cycle of the smallest total weight.

However, while these heuristics provide reasonable solutions in many cases, they also have limitations. For example, the Nearest Neighbor and the Smallest Insertion can theoretically produce significantly longer paths than the optimal solution [8].

The Christofides algorithm, on the other hand, guarantees a solution that is at most 1.5 times the optimal distance for TSP instances that follow the triangle inequality [9], but this comes at the cost of slower runtime [10]. While this trade-off may be acceptable for minor problem instances, it becomes more restrictive as the number of nodes grows. For very large datasets, even approximation algorithms can become expensive in terms of both computation time and memory usage.

In this paper, we presented a study on the accuracy and runtime of different approximation algorithms for the TSP on various datasets. First, we described our implementation of two simple greedy methods, Nearest Neighbor and Smallest Insertion, along with a slower yet mathematically precise Christofides Algorithm. Then, we compared their performance in terms of accuracy and computational cost on various datasets. The primary data used were datasets representing countries with a range of locations from 29 to 16,862, along with yeast protein interaction data ranging between 10 and 1000 proteins. We also ran our algorithms on more specific geographic applications, such as one dataset representing an electrical grid, as well as a synthetic dataset that did not follow the triangular inequality.

Finally, our study analyzed the tradeoffs between solution quality and computational efficiency across different problem

sizes and structures. We hoped to gain a better understanding of the advantages and disadvantages of each heuristic, as well as different situations where a less accurate solution might be preferred, reducing runtime and memory usage.

II. Methods

For this paper, we will use the following as our input and output of the TSP:

Input:

A set of points in the Cartesian plane $S = \{(X_1, Y_1), (X_2, Y_2), ..., (X_{n-1}, Y_{n-1}), (X_n, Y_n)\}$, where each point in the set represents the coordinates of a city.

Output:

An ordered set of cities $G = (P_1, P_2, P_3, ..., P_n, P_1)$ where the total weight of G is minimized. Given each city is a point in the plane, and $d(p_i, p_j)$ returns the distance between points i and j, we want to minimize $(\Sigma_{i=1}^{n-1} d(p_i, p_{i+1})) + d(p_n, p_1)$.

As we are not implementing the algorithm that finds the optimal solution, only algorithms that approximate the solution, we are unlikely to find the minimal solution for G. What is more, we are also loosening up the constraints such as instead of using points which are coordinates of a city/geographic location, we will use points that represent proteins in a dataset.

A. Nearest Neighbor

One of the most intuitive ways of solving the TSP, which is also naïve is the Nearest Neighbor heuristic, which is the first algorithm that we implemented for this project.

Given a list of cities, the first algorithm one might come up with to approximate the solution of the traveling salesman problem is always going to the closest unvisited city from where they are right now. In other words, given a graph, this simple heuristic begins at an arbitrary node in the input graph and then sorts its edges by distance. It then chooses the "closest" node to go to (the one with the smallest edge weight that is not in the visited set). This is repeated until each node has been visited once.

1) Description: The premise of this algorithm is that it always goes to the closest unvisited city from the last visited node. It begins at an arbitrary node from the input graph and then sorts its edges by distance. Then, the algorithm connects the node to the "closest" one, which is defined as the one with the smallest edge weight that has not been visited yet. It then repeats the same process for the newly connected node until each node has been visited once, and finally, it connects back to the start node and returns the approximate solution (both the distance and the tour).

2) Implementation: Our algorithm takes a NetworkX object as an input. To ensure that our solutions are reproducible and deterministic, we always start the tour with the first node of the NetworkX graph. The output of our function is the tour and the total tour distance.

1) Given a NetworkX object
2) Get the first vertex in the NetworkX graph by index (this way we get our results to be deterministic).
3) Add it to a hash set.
4) For each point,
   a) Sort all of its neighbors based on edge weight
   b) Select the closest unvisited neighbor.
   c) Keep track of the path and the total distance.
5) Once we visit all points we calculate the distance back to the origin and add the origin to the tour.
6) Return both the tour length and the tour itself.

3) Runtime: The complexity of the code is $O(n^2 \log n)$. The bottleneck of the code is sorting all the neighbors of any given node, which can be done in $O(n \log n)$, but as it is done one time for each of the n nodes the code is in $O(n^2 \log n)$.

4) Practical Example: Given the list of cities in Figure 1, we want to start our tour from Liverpool and visit Hamburg, London and New York City, in which order should we do the tour.

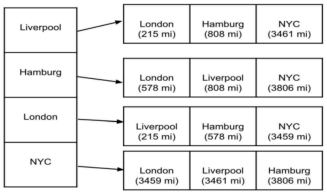

Fig. 1. Example of the tour produced by the Nearest Neighbor heuristic.

For simplicity, we assume that the graph's structure is an adjacency matrix. While this is not exactly how NetworkX stores its data, it is a useful conceptual model for this example.

Starting at Liverpool, we add it to the set of visited cities and to the tour. We then check all of the edges it is connected to (and sort them by distance). Figure 1 assumes that we have sorted the edges (for convenience). In this case, we will go to London, which is the closest city.

We then jump to the column in the adjacency matrix that represents London. We sort the adjacent cities and see that Hamburg is the closest unvisited city. We add Hamburg to the visited set and to the tour. After this, we go to its column on the adjacency matrix.

In Hamburg, we see that the closest unvisited city is New York City, so we add it to the set and the tour. As we have visited all the cities on our tour, we added Liverpool to it again to complete the tour.

At each step, when we add a city to the tour, we also update the distance of the tour by adding the distance from where we are, to the city that we are going to.

This creates the following tour: Liverpool $->$ London $->$ Hamburg $->$ New York City $->$ Liverpool. The distance of the tour is 8060 miles.

B. Smallest Insertion

Another simple heuristic we implemented is the Smallest Insertion algorithm. Which tries to put the next assigned node into the best possible position.

1) Description: Smallest insertion begins by adding two random nodes to the tour, and then iterates through the nodes of the graph. These nodes are in a list with an arbitrary order. Each node is added into the tour in the position that minimizes the total increase in distance of the tour from that addition. Thus, this method runs in $O(n^2)$ time.

2) Implementation: To ensure uniformity, this algorithm was also implemented in regard to working with NetworkX Graph objects. For reproducibility, random seeds were set for system time. Finally, the order of the nodes appended is arbitrary. For the purposes of this research, we set it to the default NetworkX ordering, which appears to be alphabetic [11].

1) Pick two random nodes.
2) Add those nodes to the tour, duplicate the first one, and compute the current distance. This is required as we have to return to the original node by the end of the tour.
3) While there are unvisited edges in the graph:
    a) Set best position and the cost increase to null.
    b) Select the next node we are going to insert.
    c) Try inserting the node into every position on the tour and save the best one.
    d) Actually insert the node into the best position and update the tour.
4) return tour and total distance

3) Example Execution: An example execution could be found in Figures 2-5. We are assuming alphabetic order and AB as the first two randomly chosen nodes. Then, the node C would be inserted in the only available spot, which is between them, creating a triangle ACBA. Then, the node D would be chosen. After inserting it in every possible position within the tour, it would be added second, making the tour ADCBA. The final node to be inserted is E, and the smallest insertion is making it 3rd, resulting in the tour ADECBA. This is a "good enough" tour according to our approximation model, yet it differs from the optimal tour ACBEDA.

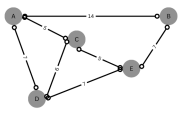

Fig. 2. Given graph.

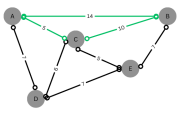

Fig. 3. Triangle ACBA.

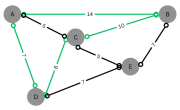

Fig. 4. Penultimate tour ADCBA.

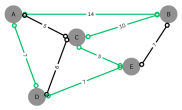

Fig. 5. Final tour ADECBA.

C. The Christofides Algorithm:

The most complex approximation algorithm for the traveling salesperson problem, that we tested, is Christofides Algorithm. This added complexity should, in theory, yield shorter Hamiltonian circuits at the expense of longer run times. Also, the scope of this algorithm's application is slightly more narrow. Unlike our other heuristics, this algorithm requires that the input graph satisfies the triangle inequality for optimal results. We will see why all of this is the case after a brief overview of the algorithm. Then, we'll dive into the finer details of our specific implementation.

The input for Christofides algorithm is a complete weighted graph G. An example input graph G is shown below.

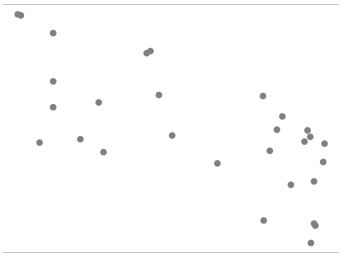

Fig. 6. Input graph for the example of an execution of Christofides that follows. This is our TSP95 dataset for Western Sahara with 29 nodes.

Christofides Algorithm then takes G and performs the seven following steps.

1) Compute the minimum spanning tree T of G.

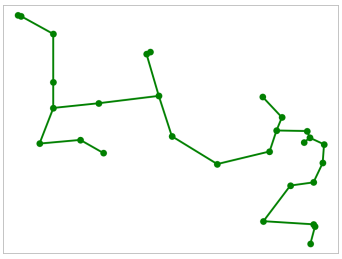

Fig. 7. Minimum spanning tree T

2) Compute the set S of all odd degree vertices in T.

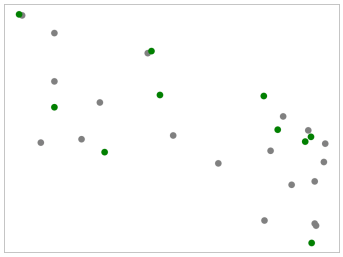

Fig. 8. Odd degree nodes in T (green)

3) Create a sub graph of G consisting of all the vertices in S and the edges between them. Compute the minimum-weight perfect matching M of this sub graph.

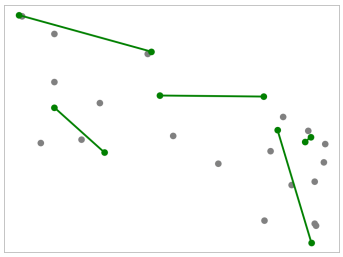

Fig. 9. Minimum weight perfect matching M

4) Combine M with T to form a Eulerian graph H. Find an Eulerian circuit for H.

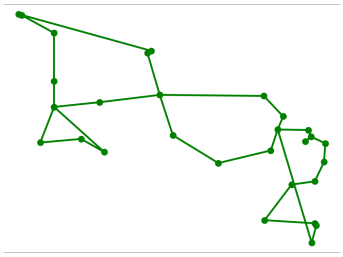

Fig. 10. Eulerian circuit of H.

5) Convert the Eulerian circuit of H into a Hamiltonian circuit and return this Hamiltonian circuit as the tour output.

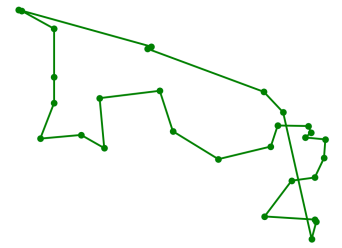

Fig. 11. Output tour

Note that by the handshaking lemma (the sum of degrees of all vertices in an undirected graph must be even), S from step 2 has an even cardinality. This is why a perfect matching is possible in step (3). Also, H, formed in step (4), is a Eulerian graph because all the vertices have an even degree. Step (4) adds one new edge to each odd degree edge of T when combined with the perfect matching of M. Thus, all vertices have an even degree in H, so an Eulerian circuit exists by Euler's Theorem. In step (5), this is where the graph respecting the triangle inequality is necessary. The Hamiltonian cycle is formed by removing repeated vertices in the Eulerian cycle. In order to get a good approximation, we need to know that removing repeated vertices does not drastically increase the cycle weight. This has the possibly of occurring as removing a repeated vertex forces a new edge into the cycle that forms a triangle with the two edges removed that went to the repeated vertex. The triangle inequality ensures that this new edge does not have a greater weight than the sum of the two removed edge weights. So, the path length is ensured to not increase with any removal. Thus, with that we have a solid general overview of Christofides Algorithm, so we will now zero-in on our implementation of some of the more intensive steps of Christofides Algorithm.

1) Implementation of Step (1): Computing minimum spanning trees of undirected graphs is a well studied problem with multiple algorithms that solve it. We used Kruskal's algorithm, which is a greedy algorithm that works by starting with all of the nodes but none of the edges from the input graph and then sequentially adding edges in order of increasing weight, not adding them if they would create a cycle. This algorithm runs in $O(ElogV)$ where E and V are the number of edges and nodes in the input graph respectively. This runtime is found from the amount of time it takes to sort the edges in the input graph before iterating over them.

2) Implementation of Step (3): For step (3) of Christofides Algorithm, we used Edmonds' Blossom Algorithm to find a minimum-weight perfect matching of M [13]. Unlike the other steps of Christofides Algorithm, we were not able to implement Edmonds' Blossom Algorithm from scratch. Instead, we used the NetworkX implementation of this algorithm. However, we were able to grasp some of how this algorithm works by implementing a simpler version of Edmonds' Blossom Algorithm that ignores edge weights. This simpler version takes a complete graph G and returns a maximal matching of G. This algorithm relies on iteratively searching for two features in a graph: augmenting paths and blossoms (Figure 5 provides a visualization of each).

An augmenting path is a series of nodes such that each pair of nodes in the path alternates between being matched or unmatched, and that the first and last node in the path are both unmatched. These paths are important because they can be "augmented." The matchings can be switched to yield one more matching than was in the path.

A blossom is a cycle in the graph that has $2n + 1$ nodes where n is some positive integer. These pose an issue for our algorithm because left untreated they could cause infinite loops or missed nodes. The algorithm deals with this by shrinking these blossoms into single nodes, and then re-expanding them later.

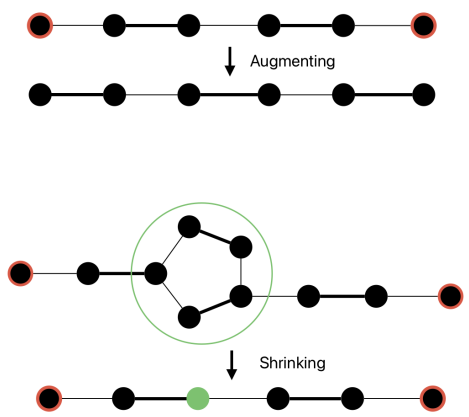

Fig. 12. Example of an augmenting path and the result from augmentation (above). Example of a blossom and the result from shrinking (below).

Here is a simplified pseudoscope for the algorithm:

1) Beginning at an unpaired node, search for an augmenting path
2) If an odd length cycle is found during the search, shrink it into a single node, creating a new contracted graph
3) Once the search finds an unmatched node, an augmenting path has been found.
4) Lift the augmenting path found in the contracted graph back to the original graph. Augment this path and them begin again from step 1.

When used in the Christofides algorithm, Blossom is used with a process called primal-dual [14]. This ensures that the perfect matching yielded is also of minimum weight. Primal-dual is a linear programming term that describes optimizing two sides of a given problem, the primal and the dual. In this case, the primal part of the algorithm is ensuring that the matching is maximal, that is there are the most matchings possible between nodes. The dual part is finding a matching that is of the smallest total weight possible. This process is vital to the algorithm used in Christofides but it is complicated. It involves keeping track of and updating labels on each node of the graph that account for the weight of matching them, and using these labels to route the exact augmenting paths chosen.

The time complexity of the version of Edmonds' Blossom Algorithm that is in the NetworkX library is $O(nm \log n)$ where n is the number of nodes in the graph and m is the number of edges in the graph [15].

3) Implementation of Step (4) and Step (5): For step (4) of Christofides Algorithm, we used Fleury's Algorithm. This algorithm takes a connected graph G as an input and outputs an Eulerian path or circuit if one exists. The steps of this algorithm are as follows.

1) Check whether G has zero or two odd degree edges. If neither is true, then return nothing as there is no Eulerian path or circuit for G.
2) If there are zero odd degree edges, start anywhere. If there are two odd degree edges, start at one of these odd degree edges.
3) Let P be a list of vertices representing a path through G.
4) While there are edges in the graph:
   a) If the current node is degree one, then remove the one edge and go to the vertex connects to adding that vertex to the path.
   b) If the current node is not degree one, then select one edge.
   c) Perform depth first search on the graph counting the number of reachable vertices from the current vertex. Call this count c_1.
   d) Remove the selected edge and perform depth first search from the current vertex, again counting the number of reachable vertices. Call this count c_2
   e) If $c_1 > c_2$, then remove the edge, the current vertex becomes the vertex that this edge goes to, and this new vertex is added to P. If not, then go back to step (c) looking at a different edge connected to the current vertex.
5) Return P.

Let $G = (V, E)$ be an arbitrary input graph. The time complexity of Fleury's algorithm is $O((|V| + |E|)^2)$. This is evident as this algorithm essentially performs depth-first search, which is $O(|V| + |E|)$, but at each step of this depth-first search we perform two more depth-first searches to test whether an edge is a bridge. Now in the case of Christofides, we know that the input Eulerian graph H is formed by combining a tree T and a perfect matching of all the odd degree vertices of this tree. Since $T = (V, E_T)$ is a tree, the number edges $|E_T| < |V|$. Also, the number of of edges added in the perfect matching of odd degree vertices of T is bounded by $\lfloor \frac{|V|}{2} \rfloor$. Hence, the number of edges in $H = (V, E_H)$ is $|E_H| < |V| + \lfloor \frac{|V|}{2} \rfloor < 2|V|$. Therefore, in our implementation of Christofides Algorithm, Fleury's Algorithm on H is $O((|V| + 2|V|)^2) = O(V^2)$ [16].

4) Overall time complexity: The version of Christofides Algorithm that we have implemented is $O(n^3 \log n)$ time, where n is the number of nodes in the input graph. This is due to a bottleneck in one of the steps of the algorithm: computing the minimum weight perfect matching of the odd degree nodes in the minimum spanning tree. As we have covered, this step uses the Blossom algorithm, which runs in $O(nm \log n)$ time where m is the number of edges in the graph. Since the input to Christofides Algorithm is a complete graph we know that $O(m) = O(n^2)$. Also, the subgraph input into the blossom algorithm could potentially be the whole input graph in the case where all nodes in the minimum spanning tree have odd degree. Therefore, we get that the Blossom algorithm here is $O(n^3 \log n)$. The rest of the steps run in $O(n^2)$ time or better, so the worst-case upper bound is equal to Blossom's. Therefore, our implementation of Christofides is $O(n^3 \log n)$. Notably, there are faster minimum weight perfect matching algorithms that exists which can bring Christofides down to $O(n^3)$.

5) 1.5 length upper bound: The Christofides Algorithm has been shown to yield tours at worst 1.5 times longer than optimal as long as the graphs satisfy the triangle inequality [10]. A graph that satisfies the triangle inequality is called "metric." In a metric graph, a direct path between two vertices cannot be longer than one that starts and ends with these same two vertices but visits some other vertices in between. Put more formally, for any three vertices a, b, and c, the path from a to c can be no longer than the path from a to b plus the path from b to c.

To prove the 1.5 upper bound, consider an exact solution to TSP on a given input graph G. Removing one edge from this solution becomes a spanning tree S of G with some weight $w(S)$. Our MST T has a weight less than or equal to S since it is minimum. To find our tour from T, we add the minimum weight perfect matching of T's odd degree nodes.

We label our set of odd-degree nodes in T on our exact tour of G. We then create a "pseudo-matching" of our tour by connecting alternating labeled nodes with each edge in the tour that it takes to get to them. For example, if our tour contained a path between nodes 1, 2, 3, and 4, and nodes 1 and 4 were labeled, one "pair" in our pseudo-matching would be the edges (1,2), (2,3), and (3,4). We traverse the whole tour in this way, forming alternating pairs of the labeled nodes, almost like an augmenting path where nodes go paired, unpaired, paired, etc. Because of the alternating nature of our pseudo-matching, we could switch every pair to get a different one. Together, these two pseudo-matchings include all the edges in the exact tour, and one of them will have a total edge weight of at most half that of the exact tour. This pseudo-matching can be converted to a normal matching using edges not necessarily in the exact tour, and doing this forms a (true) maximal matching of the odd degree nodes in T (which has at most the same total edge weight of the pseudo-matching because of the triangle inequality). This maximal matching is greater or equal in weight to our minimum weight perfect matching M by the latter's definition, which ultimately means that M is at most half the weight of the exact tour.

Combining M and T yields an Eulerian graph from which we find an Eulerian circuit which has edge weight equal to that of M plus that of T. This can be no more than 1.5 times that of the exact tour due to the above justification. Our final step is removing repeated vertices from our Eulerian circuit to turn it into our output tour. This removal cannot add any additional edge weight due to the triangle inequality since the node is simply skipped in the circuit. Therefore, our output tour must have total weight no more than 1.5 times that of the optimal tour.

III. Results

A. Datasets

1) Geographical Data: The first half of our research concerned geographical data that followed the triangular inequality. We used 10 datasets from the TSP95 library for comparison to compare our solutions to [18]. The datasets we used were the following: Western Sahara (WI29), Djibouti (DJ38), Qatar (QA194), Uruguay (UY734), Zimbabwe (ZI929), Oman (OA1979), Canada (CA4663), Greece (GR9882), Finland (FI10639), and Italy (IT16862) [7]. Each dataset code should be read as XX123, where XX is the country code, and 123 is the number of nodes in that dataset. The sizes of these datasets vary from 29 to 16,862 points. These are widely cited TSP instances where each dataset represents a country with several locations. There is an edge joining each location to each other location with weight equivalent to the distance between them. One significant advantage of the selected datasets is that the solutions to all of them are known and the total tour length is readily available online, making it trivial to compare our solutions to the optimal. Finally, unlike some of the existing datasets, none of the datasets chosen have duplicated points [7].

2) Electrical grid data: A more real-world application of TSP has been used in routing problems for electrical power distribution companies [12]. We used an electrical grid dataset containing 425 nodes representing grid stations that need service from the team being routed. The edges represent the geographical distance between the stations. Each of our algorithms was run on these data to analyze their efficacy in this application.

3) Protein interaction data: Data was gathered from the STRING database which is a protein-protein interaction (PPI) network that describes the interactions between almost 60 million proteins [19]. This data is accessed by choosing a protein of interest and then constructing an interaction network based on it. We chose to look at interactions with Alcohol Dehydrogenase (ADH) in humans, which is an important enzyme in the metabolism of alcohol. We created interaction datasets of varying size ranging from 10 to 1000 proteins. The proteins are the nodes in the graphs we used for testing, and the edges between them correspond to the amount of interaction they interact according to the database, which gives an interaction score from 0 to 1. We inverted this score meaning a lower number means more interaction. Our heuristics were run on these datasets to yield protein interaction tours of minimum edge weights or maximum interaction. These tours can be analyzed to hypothesize metabolic pathways that these proteins are involved in.

B. Experimental Environment

The algorithms were run on one of the Computational Research Users Group Cluster computers owned by Carleton College [20]. The machine we used was called Ada. It had 56 Intel(R) Xeon(R) CPU E5-2, each with two threads per core, and 377 GB of RAM.

We developed a single reproducible pipeline responsible for loading the datasets, converting them to a network object, and passing them to our heuristic/algorithm. We used Python's time library to track how long each algorithm took to run. We also included a user interface that parses the keyboard input for testing reasons. However, we bypassed it by passing all the parameters through a pipe as *runner.py ¡ data.txt*.

Tour Length	WI29	DJ38	QA194	UY734	ZI929	OA1979	CA4663	GR9882	FI10639	IT16862
Optimal	27,603	6,656	9,352	79,114	95,345	86,891	1,290,319	300,899	520,527	557,315
Smallest Insertion	34,048	7,884	11,457	97,657	118,214	108,749	1,614,506	372,479	643,736	682,857
Nearest Neighbor	36,388	9,745	11,640	99,247	113,926	120,908	1,646,884	388,944	649,600	706,076
Christofides Algorithm	30,640	6,854	10,341	86,597	106,095	95,360	1,414,054	330,945	573,591	616,722

TABLE I
COMPARISON OF TSP95 TOUR LENGTHS USING DIFFERENT HEURISTICS ON VARIOUS TSP INSTANCES.

We ran each algorithm thrice on each dataset to compute the average running time, given possible fluctuations that might have made the code significantly faster or slower. The Smallest Insertion algorithm relies on randomness. To ensure reproducibility, we used a pre-set random seed.

C. Experimental Validation

To validate our algorithms, we set up a number of checks and tests that verified that the tours were complete and that the distances were accurate. They included visualizations and algorithmic comparisons. All these tests were used for each algorithm and dataset before running the code on the datasets we have studied for this project. This way, we were able to ensure that the results were correct and reproducible.

D. Geographical Data

The bulk of our findings concerned geographical data that follows the triangular inequality. As the TSP originates from a geographical setting, this is the most straightforward application of the problem. After averaging the three runs of each algorithm implemented, we got the results displayed in Table 1. In the table, the datasets were ordered by the number of nodes in the increasing order, and the names of the datasets follow the conventions described in this paper.

As we are working with complete graphs, each graph has $\frac{n(n-1)}{2} = \frac{n^2-n}{2}$ edges, so with every new location added the number of edges increases quadratically. Naturally, we observed this theoretical increase in the memory usage and time our algorithms took to run.

To simplify data analysis, we used a scaling factor, setting our optimal tour to 1, and the result for each heuristic to the number of times it fits the optimal tour. Therefore, we used the following equation $\frac{total\ tour\ length\ using\ current\ algorithm}{optimal\ tour\ length}$. As we discovered, the algorithm's optimality does not seem to correlate with the number of nodes in the graph for these specific datasets. However, it is worth pointing out that it is possible to build graphs that favor one heuristic over another. Therefore, these conclusions are specific to our datasets and other instances following similar constraints (i.e. geographical data where the triangle inequality applies).

It is important to outline that Christofides did not reach its theoretical worst possible bound. The algorithm is guaranteed to generate tours at most 1.5 times longer than the optimal one. However, the results remained around 1.1 times worse than optimal, which is significantly better. Moreover, our simple heuristics, Nearest Neighbor and Smallest Insertion, remained between 1.2 and 1.3 times the optimal. Even our simple heuristics yielded better results than the theoretically worst case for Christofides. Therefore, all algorithms fared well in these datasets.

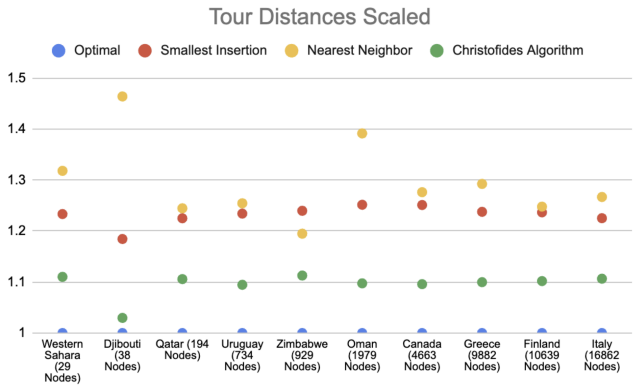

Fig. 13. Scaled comparison visualization of tour optimality.

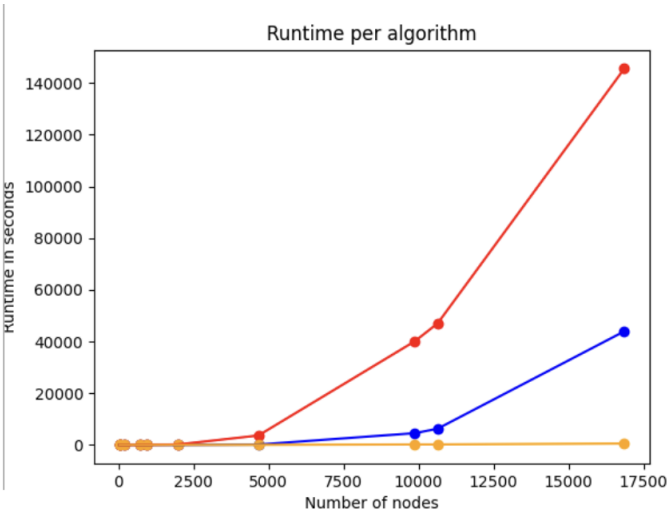

Fig. 14. Comparison of runtime per heuristic and number of nodes in each tour.

The Christofides Algorithm has a theoretical bound of at most 1.5 times worse than optimal. Based on our results, it did not go significantly over 1.1 times worse than optimal. However, such prominent results are offset by a significantly slower running time compared to other heuristics, as presented in Figure 6 and Table II.

All the algorithms had a similar runtime for datasets up until around 200 entries. In that range, they all took less than one second to execute. As we approached 700 entries, the

Time (seconds)	WI29	DJ38	QA194	UY734	ZI929	OA1979	CA4663	GR9882	FI10639	IT16862
Smallest Insertion	0.002	0.003	0.068	0.77	1.25	5.98	40.53	185.46	190.60	556.47
Nearest Neighbor	0.002	0.003	0.058	0.63	1.06	5.13	160.84	4566.48	6283.01	43963.43
Christofides Algorithm	0.008	0.012	0.875	20.67	42.26	195.39	3656.05	40154.20	47028.09	145589.27

TABLE II
COMPARISON OF RUNTIME (IN SECONDS) FOR DIFFERENT HEURISTICS ACROSS TSP INSTANCES.

Christofides Algorithm became significantly slower than the simple heuristics. While both Nearest Neighbor and Smallest Insertion took less than one second, Christofides took around 20 seconds.

Nearest Neighbor and Smallest Insertion performed similarly runtime-wise with the datasets smaller than 4500 points. For the CA4663 dataset, the Smallest Insertion was completed in 40 seconds, while the Nearest Neighbor took 160. Therefore, nearest neighbor started becoming much slower. Meanwhile, the Christofides algorithm started requiring significant computational power for significant periods. Such results brought insight into balancing the needed runtime and results with the available computational resources. Christofides broke the barrier of 10 hours of computation at around 9.8 thousand points (The Greece dataset), while the Nearest Neighbor did that only on our largest dataset, which has 16.8 thousand points. At our largest dataset, Christofides took over 40 hours to finish the computation.

Another finding was that the Smallest Insertion algorithm works significantly faster while producing similar results to the Nearest Neighbor algorithm for datasets larger than approximately 4000 nodes. The difference becomes more striking as the datasets grow: Our largest dataset took 9.5 minutes for the Smallest Insertion algorithm, 12 hours for the Nearest Neighbor algorithm, and 40 hours for the Christofides algorithm.

We observed that the algorithms did not precisely follow their theoretical runtimes, but the general trend remained respectably logarithmic and polynomial for Nearest Neighbor and Smallest Insertion / Christofides. Although Christofides yields the shortest tours, it is substantially slower than the other algorithms. It is also worth mentioning that loading the datasets into memory and converting them into NetworkX Graph objects took a significant chunk of runtime. However, this task was not timed, and we did not include it in our analysis. All the results included was the execution time of the algorithm per se.

Moreover, our largest datasets used up a considerable amount of memory. The ones with more than 10,000 nodes required more than to 15 Gigabytes of RAM, and our largest dataset required 45 Gigabytes of RAM. With other data structures used for the Christofides algorithm, this number jumped to 65 Gigabytes. Therefore, in order to run this program, the usage of a powerful computational cluster is necessary.

E. A comparison between the path chosen by nearest neighbor and the path chosen by Christofides

To compare the accuracy of the tours built by different heuristics, we counted the number of edges that these tours have in common and calculated the longest sub-paths that the both tours have in common. The original TSP95 paper did not include the optimal tour, only its visualization. So, we could not calculate how far our tours were from optimal. Therefore, we used the tour made by the Christofides Algorithm as our standard for accuracy because of its theoretical and practical bound.

Fig. 15. Annotated tour of Uruguay generated by the Nearest Neighbor Algorithm.

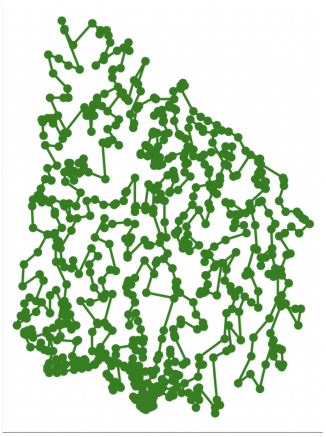

Fig. 16. Tour of Uruguay generated by the Christofides Algorithm.

The tours above have 282 edges in common (which represents 38.419% of the tour); what is more, the longest subpath they have in common is 13 edges long. This means the tours are very similar, as almost 40% of the edges are the same. Moreover, the longest subpath they have in common is also large.

However, the tour produced by the Christofides Algorithm is 9.4% worse than optimal, while the Nearest Neighbor's tour is 25.4% worse than optimal. So, the Nearest Neighbor algorithm is 16% further away from the optimal solution than Christofides.

This is explained by the differences between the lengths of the edges chosen by different algorithms. An example can be seen in the red edges red in Figure 17. Those edges are very long and add considerable weight to the graph in the long run. They were chosen because of the greedy nature of the Nearest Neighbor algorithm. When it got to the neighborhood (denoted as the points that are close to an edge, roughly on the circle of the image) of the point that led to the long edge, all the neighbors of the point had already visited. Hence, they had already been added to the tour. So, the closest unvisited node was far away, and the code had no choice but to follow it. On the other hand, the Christofides algorithm is not greedy; it made choices that did not seem optimal right away, allowing it

to choose shorter edges in the future and make the path closer to optimal.

We ran similar tests in other datasets, such as Zimbabwe, Greece, Finland, and Italy, and observed similar results.

The case for nondeterminism: It is worth pointing out that neither nearest neighbor nor smallest insertion are deterministic, and we might get different solutions based on our starting point and other factors. For instance, take a look at figure 9.

Fig. 17. Example of non-determinism in decisions.

If one were to start the tour from the red node, the tour would be different. We would not go to any of the nodes that are currently connected to it. Other nodes are closer to the new starting point, which is clear in the image. Therefore, both the distance and the tour per se depend on the starting position.

It is worth pointing out that the algorithm, as implemented, is deterministic, as it always starts from the node NetworkX classified as 0. We will always find the same tour because we do not use randomness to read from the file or add to NetworkX object. Still, it is worth mentioning that the quality of the paths might change depending on the starting node.

The smallest Insertion algorithm is even more influenced by randomness, as it starts by selecting two random points to start/end the tour. Moreover, it gets the next nodes to add to the tour from a list (which is in the order provided by NetworkX).

1) Electrical Grid: The table below shows the results for the Electrical Grid dataset. In general, it follows exactly the same pattern as the TSP95 data. The smallest insertion is still the fastest algorithm, followed by the nearest neighbor and then Christofides. Similarly, Christofides yields the best tour, followed by the smallest insertion, while the worst tour is from the nearest neighbor.

However, the dataset is relatively small — it only has 425 nodes. Therefore, significant differences in results between heuristics (seen in larger datasets) are not observed.

This dataset also follows geographic data. After all, all the electric grid points are points within a city. Hence, this dataset follows the triangle inequality, and it is very similar to our TSP but on a much smaller scale (a city instead of a country).

The electrical grid dataset is important because it is not part of the TSP95. So, we could not rely on a library to read and parse the data from it. We had to manually process the data and create the graph and all of its edges.

Electrical Grid Results	Distance	Runtime (s)
Smallest Insertion	187.33	0.336
Nearest Neighbor	215.23	0.294
Christofides	188.58	5.14

The dataset had the latitude and longitude of each of the 425 nodes. We saw those as points in the cartesian plane and then developed an $O(n^2)$ algorithm to create the optimal graph. We iterated through every point, and for each of these points, we iterated through the set of points again. We then calculated the distance between each of the points (in miles) and added it to a NetworkX object, creating a complete graph.

Our algorithm operated the following way:

1) For each point i:
2) a) For each point j:
   b) if j != i:
   c) calculate the distance between i and j in miles
   d) add points to the networkx graph
3) return graph

This was a good educational exercise in which we generated a complete graph. We believe the TSP library is doing something similar inside its functions.

F. Synthetic Non-triangular Data

One of the assumptions of the Christofides algorithm is that datasets must follow the triangle inequality. This is essential as the algorithm takes shortcuts when converting from an Eulerian circuit to a Hamiltonian cycle. If the Eulerian path follows the triangle inequality data, the shortcuts will shorten the length of our tour. However, the same shortcuts might make the tour less optimal if the dataset does not follow the triangle inequality.

As an exercise, we generated an undirected graph with 100 nodes. The distance between each node is a random number picked from a uniform distribution ranging from 0 to 10. This gives us confidence that the dataset will not follow the triangle inequality. The results of running our three heuristics in it are outlined in the following table:

Synthetic Non-Triangular Data	Distance	Runtime (s)
Smallest Insertion	137	0.017
Nearest Neighbor	117	0.016
Christofides	286	0.077

In datasets that do not follow the triangle inequality, we can see that the accuracy of the Christofides algorithm is significantly worse than the performance by the Smallest Insertion and the Nearest Neighbor algorithms. The smallest tour is more than two times larger than the one generated by the Nearest Neighbor. What is more, Christofides will still be the slowest, as it has a runtime of $O(n^3)$ compared to $O(n^2 log n)$ and $O(n^2)$ of the Nearest Neighbor and the Smallest Insertion respectively. Therefore, either of the simpler heuristics performs better for non-triangular datasets.

Tour Length	Protein10	Protein20	Protein50	Protein100	Protein250	Protein500	Protein1000
Smallest Insertion	0.62	1.592	5.785	20.886	55.673	119.16	210.927
Nearest Neighbor	0.775	2.822	7.595	25.954	56.93	121.217	201.786
Christofides Algorithm	0.978	1.855	7.268	27.571	70.671	151.141	260.515

TABLE III
COMPARISON OF TOUR LENGTH FOR DIFFERENT HEURISTICS ACROSS PROTEIN TSP INSTANCES.

Execution Time (seconds)	Protein10	Protein20	Protein50	Protein100	Protein250	Protein500	Protein1000
Smallest Insertion	0.0003	0.0009	0.004	0.016	0.099	0.311	1.331
Nearest Neighbor	0.0003	0.0011	0.004	0.0133	0.081	0.245	1.126
Christofides Algorithm	0.0027	0.0069	0.0728	0.370	3.843	26.796	209.029

TABLE IV
COMPARISON OF EXECUTION TIME (IN SECONDS) FOR DIFFERENT HEURISTICS ACROSS PROTEIN TSP INSTANCES.

G. Protein Interaction Data

Solving TSP for protein interaction networks can yield important insights into how the proteins function in vivo. A tour through such a network can represent a metabolic pathway in the body, as it indicates that specific proteins have particularly favorable interactions with each other, or to draw conclusions about theoretical functions of proteins that have not yet been examined experimentally [21].

Given the non-metric nature of the protein-protein interaction data, the graphs we worked with do not satisfy the triangle inequality. This poses an issue for the Christofides algorithm, as its 1.5 upper bound is only guaranteed for graphs that do. This is because shortcutting from an Eulerian circuit to a Hamiltonian circuit may increase tour length when the path between two nodes is not guaranteed to be as short or shorter than that path routed through a third node. So, we cannot be confident that the Christofides algorithm will perform better than our other algorithms in accuracy, as we saw with the TSP95 data.

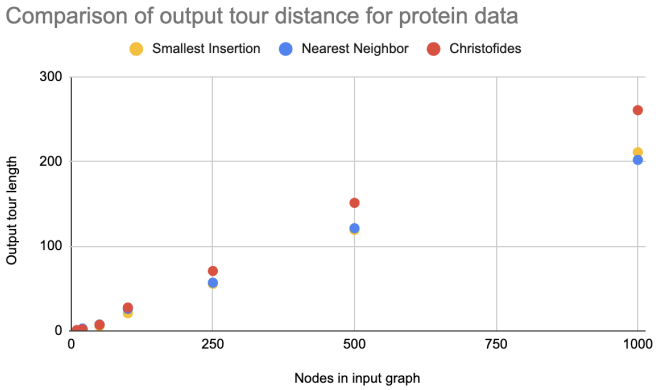

Fig. 18. Comparison of runtime per heuristic and number of nodes in each dataset.

Figure 4 and Table III show the differences in output tour length for each heuristic on protein data ranging in number of nodes from 10 to 1000. Smallest Insertion and Nearest Neighbor yield very similar lengths, but Christofides performs much worse. Most of the tours yielded by Christofides are 20-30% longer than those given by the other heuristics. This, coupled with the fact that Christofides is still significantly slower (Table IV) than Nearest Neighbor and Smallest Insertion on larger datasets confirms that Christofides is flawed on this type of data.

Interpreting the tours we found in a biological context is outside the scope of our training. This testing was mostly done as a demonstration of the benefits and limitations of our three algorithms when used on non-metric datasets. In the future, it would be interesting to examine and compare how well the tours that we found did at predicting biological processes rather than solely looking at the length of tours produced. The latter can give us insight into how these algorithms work and what causes them to break down, but the former would be more rigorous.

IV. DISCUSSION

The results that we have collected provide us with valuable insight into the strengths and weaknesses of each heuristic. With this information, we reflect on what this tells us for real-world applications of these algorithms. Let us first consider the real-world applications of each dataset.

A. Case-by-case Analysis

For the geographic datasets, some real-world applications might be tourism, GPS navigation, and package delivery services such as Amazon and USPS. A tour of all the cities in a country could theoretically be used by tourists traveling in a given country. In this situation, finding an approximate solution to the TSP helps minimize the fuel cost and the driving time. The shorter the tour, the less time and money is spent on transportation between cities. So, we want to use a heuristic that gives us a relatively short cycle through all the cities.

We must also consider the computational power an agent constructing the tour has at their disposal. They will most likely have less computational power than the machines we used in our research, limited to their phones and computers. Also, they will not want to spend much time getting things exactly. Hence, something quick, convenient, and not computationally intensive is desirable for this situation. Our results for the country datasets show that as we increase the number of nodes, the Christofides Algorithm takes much longer than

the nearest-neighbor and smallest insertion heuristics and is much more computationally intense. Thus, if there are many cities, then the Christofides Algorithm is off the table for the tourist situation. It would be better to use the smallest insertion for convenience, given the computational power that a tourist has.

Furthermore, if there is a large number of cities, then the Smallest Insertion is the best bet due to its drastically faster run time compared to both other heuristics. Conversely, if the number of cities is small, the Christofides Algorithm can reasonably used. In our hypothetical tourist situation, we have deduced that the computational resources, the time available, and the scale of the data being considered are all crucial factors in deciding which heuristic is best for real-world applications.

As for package delivery services, a shorter route might bring significant cost reductions in the long run and marginally shorter delivery times (delivering in the morning rather than the afternoon, for example). Shorter tours allow us to reduce expenses such as gas or work time. In order to determine which tour is best, we need to consider some details such as the size of the tour and whether it changes frequently. If the tour does not change, it is worth using an algorithm such as Christofides (or even trying to find the optimal solution). Many delivery companies have the resources to use significant computational power to find the best solution for the TSP problem, even for larger graphs. For tours that change frequently, the best algorithm to use depends on the size of the tour and the computational power available. If the tour is small enough or there is enough computational power, it is worth using Christofides as it is the most precise algorithm. However, if the tour is too large, Christofides might not be feasible as the computation might take longer than the time the tour has to be updated.

Next, for the electrical grid dataset, the real-world application is constructing an electrical grid between different power stations. Unlike the situation for the country datasets with individual tourists, the cost of constructing an electric grid is going to be much higher. Also, the resources of the power company constructing this grid might be willing to invest are much more substantial. A large company can afford or already possesses the necessary computation power to find a better solution to the problem. Also Convenience is not a focus, a company much more concerned about profit over everything. Thus, the power company will want to take their time find the best approximate solution to the traveling salesperson problem, where the nodes are power stations, in order to form a grid that uses the least amount of resources. This will reduce costs of construction, reduce the grid size, reduce maintenance costs, and increase profits once the final product is fully constructed. What is more, it is unlikely that the power grid will be updated very frequently, so a power company can afford an algorithm that takes longer to run but finds a better solution. Therefore, out of all the heuristics that we tested, the power company should use Christofides Algorithm (or depending on its size, even the optimal solution). Finding the shortest possible Hamiltonian cycle is completely prioritized ignoring runtime, within reason. The deciding factor here was the desired high accuracy of the final result.

For our last dataset, the protein interaction data, the real-world application is determining metabolic pathways that proteins are involved in. Notably, this dataset does not follow the triangle equality unlike the other datasets. Based on our experiments a graph not following the triangle inequality completely rules out Christofides as a viable option. Christofides Algorithm generally takes longer than the other two heretics and actually produces consistently worse results. We specified earlier that this is the case due to step (5) of Christofides Algorithm requiring the triangle inequality to adequately bound the length of the returned Hamiltonian cycle. So, we have narrowed down our choice of heuristic to either smallest insertion or nearest neighbor.

B. Determining the Right Heuristic to Apply

Through our case-by-case analysis of the dataset that we used, the real life implications of each, and our results, we have identified five factors that together decide which approximation algorithm should be used in a certain context. The five factors are the computational resources available, the time available, the scale of the data, the nature of the data, and accuracy requirements.

First of all, if the tour needs to be recalculated daily or very frequently, it might be better to use one of our naive heuristics. Those are done executing much faster, and yield results which are acceptable. What is more, for larger datasets it might not be feasible to recalculate the path daily. As they might require a lot of computation power and the calculation might take longer than 244 hours (it might even take months). On the other hand, if the tour needs to be calculated only once (and is unlikely to change), it might be worth it investing in finding a better approximation. After all, this might save money and time in the long run.

What is more, for larger datasets Christofides might not be viable at all. The time it takes to run the algorithm increases really fast. For our largest dataset, for example, it took us more than 40 hours to find the tour with Christofides. For datasets which are much larger, finding the tour with Christofides might take months or even years, which is not always viable.

In order to circumvent the problem described above, one might use lower level programming languages, which tend to be faster, a more powerful cluster, or use some sort of parallelization. These will certainly make the program find the solutions faster or make the problems only show up only for larger datasets.

As we are dealing with a complete graph, and for each new node the size of the graph increases quadratically, for larger datasets it might not be feasible to run any of the algorithms studied here. We need to save the graphs in RAM somehow in order to operate with them. So memory is indeed a bottleneck. Without the required amount of RAM, running the algorithms would not be viable.

C. Limitations and Further Recommendations

Now that we have thoroughly examined which of the approximation algorithms should be used in particular cases, it is worth noting that there are many other approximation algorithms for the traveling salesperson problem that are also worth considering. This is due to the limitations of the Christofides Algorithm, the smallest insertion heuristic, and the nearest-neighbor heuristic. For example, we observed through our experiments that the Christofides Algorithm requires the datasets to follow triangle inequality to be a viable option. Hence, it would be worth considering alternative approximation algorithms if high precision is required and sufficient time and computation power are available.

One such alternative is the Lin-Kernighan heuristic. This heuristic has a time complexity of $O(n^2)$. It is used to find optimal and near-optimal solutions with high frequency for the traveling salesperson problem. This heuristic does this by improving upon previously found solutions of this algorithm. It essentially allows whoever runs it to continue to iterating until they obtain a solution that is desirable for their use. This is very convenient for real-world applications as it has flexibility with run time and achieves accuracy. However, the rate at which this algorithm improves solutions and approaches optimal gets slower as the dataset size increases. [22]

Another alternative would be using an exact algorithm for the traveling salesperson problem. These could be desirable if the scale of the dataset is not that large and computation power and time are plentiful. An example of an exact algorithm is the Held-Karp algorithm. The time complexity of this algorithm is $O(n^2 \cdot 2^n)$. Hence, this algorithm would take much longer than any approximation algorithm, but the solution would be exact. The run time of a brute force algorithm for the traveling salesperson problem is $O(n!)$. The Held-Karp utilizes dynamic programming to achieve a run time that is significantly faster than brute force $O(n2^n)$. [23]

Further research to expand our findings should focus on the two mentioned algorithms and how their running time compares to the approximation algorithms we analyzed. This would lead to an even better understanding of how algorithms for the TSP should be chosen for particular real-world scenarios.

V. Conclusion

Given the computationally difficult nature of TSP, approximation algorithms will generally be the method of choice for solving the problem in many cases. Which one to choose, however, depends on multiple factors. Our experiments demonstrated trade-offs between the algorithms we implemented that must be considered when choosing which to use.

Of the three, Christofides should be the algorithm of choice on small datasets, as the runtime difference will be minimal but the accuracy higher. However, as the size of the data increases, the problem becomes more nuanced. If the output needs to be found quickly, Nearest Neighbor or Smallest Insertion should be used despite the decreased accuracy. This could be the case in applications where the main reason for using TSP in the first place is time efficiency. If accuracy is the goal and time is less pressing, Christofides should be run. Such a decision makes sense when minimizing the route saves resources other than time, or if the route can be calculated once and used many times in the future. In the case of data that does not satisfy the triangle inequality, however, Christofides should not be used. One of the other two algorithms would be a better option or entirely different.

Acknowledgment

Thanks are due to our advisor, Layla Oesper, for her patient guidance throughout the research process. We would also like to thank Mike Tie for getting us access to the Ada Computational Cluster for our research.

References

[1] Lawler, E. L. (1985). The Travelling Salesman Problem: A Guided Tour of Combinatorial Optimization (Repr. with corrections. ed.). John Wiley & sons. ISBN 978-0-471-90413-7. https://books.google.com/books?id=qbFlMwEACAAJ

[2] Chan, D. and Mercier, D. (1989). IC insertion: an application of the travelling salesman problem. International Journal of Production Research - INT J PROD RES. 27. 1837-1841. 10.1080/00207548908942657.

[3] Melnikov, B., and Chaikovskii. D. (2024). Some General Heuristics in the Traveling Salesman Problem and the Problem of Reconstructing the DNA Chain Distance Matrix. In Proceedings of the 2023 7th International Conference on Computer Science and Artificial Intelligence (CSAI '23). Association for Computing Machinery, New York, NY, USA, 361–368. https://doi.org/10.1145/3638584.3638607

[4] Johnson, D. S. and McGeoch, L. A. (1997). "The Traveling Salesman Problem: A Case Study in Local Optimization" (PDF). In Aarts, E. H. L.; Lenstra, J. K. (eds.). Local Search in Combinatorial Optimisation. London: John Wiley and Sons Ltd. pp. 215–310.

[5] Papadimitriou, Christos H. (1977), "The Euclidean traveling salesman problem is NP-complete", Theoretical Computer Science, 4 (3): 237–244. doi:10.1016/0304-3975(77)90012-3

[6] Gigerenzer, G.; Gaissmaier, W. (2011). "Heuristic Decision Making". Annual Review of Psychology. 62: 451–482. doi:10.1146/annurev-psych-120709-145346. hdl:11858/00-001M-0000-0024-F16D-5. PMID 21126183.

[7] https://www.math.uwaterloo.ca/tsp/world/

[8] Gutina, G., Yeob, A., Zverovich, A. (2002). "Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP". Discrete Applied Mathematics. 117 (1–3): 81–86. doi:10.1016/S0166-218X(01)00195-0

[9] https://kaminsky.ieor.berkeley.edu/ieor251/notes/2-16-05.pdf

[10] Christofides, N. (1976), Worst-case analysis of a new heuristic for the travelling salesman problem (PDF), Report 388, Graduate School of Industrial Administration, CMU, archived (PDF) from the original on July 21, 2019

[11] https://networkx.org/documentation/stable/index.html

[12] Barbosa, D., Silla, C., and Kashiwabara, A. (2015). Applying a variation of the ant colony optimization algorithm to solve the multiple traveling salesmen problem to route the teams of the electric power distribution companies. In Proceedings of the annual conference on Brazilian Symposium on Information Systems: Information Systems: A Computer Socio-Technical Perspective - Volume 1 (SBSI '15), Vol. 1. Brazilian Computer Society, Porto Alegre, BRA, 23–30.

[13] Edmonds, J. (1973), "Paths, Trees, and Flowers," Canadian Journal of Mathematics, vol. 17, pp. 449–467, 1965. doi:10.4153/CJM-1965-045-4

[14] Vondrak, J. (2017). "Polyhedral techniques in combinatorial optimization," Stanford University.

[15] GALIL, Z. (1986). Efficient algorithms for finding maximum matching in graphs. ACM Computing Surveys, 18(1), 23–38. https://doi.org/10.1145/6462.6502

[16] Journal de Mathematiques Elementaires. (1877). . Librairie Vuibert.

[17] Cook, W., and Andre R. (1999). "Computing minimum-weight perfect matchings." INFORMS journal on computing 11.2: 138-148.

[18] Reinelt, G. (1995) TSPLIB. https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/
[19] https://string-db.org/
[20] https://stolafcarleton.teamdynamix.com/TDClient/3356/Portal/KB/ArticleDet?ID=158103
[21] Johnson, O., Liu, J. (2006). A traveling salesman approach for predicting protein functions. Source Code Biol Med 1, 3. https://doi.org/10.1186/1751-0473-1-3
[22] Lin, S., & Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. Operations research, 21(2), 498–516.
[23] Held, M., & Karp, R. M. (1962). A Dynamic Programming Approach to Sequencing Problems. Journal of the Society for Industrial and Applied Mathematics, 10(1), 196–210. https://doi.org/10.1137/0110015