

Different Heuristic Approaches to Solving the Traveling Salesman Problem

Author: Bryan Kim

Affiliation: Seoul Internation School

E-mail: bryan.kim23@stu.siskorea.org

ABSTRACT

The Traveling Salesman Problem (TSP) is a famous Non-Deterministic Polynomial problem that seeks to figure out, when given a certain set of cities, what the fastest or most efficient route a salesman can take is. Yet, while the premise of the problem is simple, when given a substantial set of locations, it would take an unreasonable amount of time for one to compute the distance of each possible route; hence, mathematicians have been developing algorithms that would produce a reasonably short route within a reasonable amount of time. This paper looks into the various algorithms and approximation methods mathematicians have been able to discover throughout the past few decades, and takes a deeper dive into three drastically different algorithms or methods: the nearest neighbor algorithm, the lin-kernighan heuristic, and the FindShortestTour algorithm (a built-in function in Mathematica's coding base). The objective of this paper is to investigate the various properties that surround these algorithms, to test them out in different scenarios of the TSP, and to explain why we got the results our experiment produced.

1 INTRODUCTION

1.1 Motivation

In a day and age when technological developments have become the norm, human life has significantly

evolved for the better, with elevated convenience and expanded capabilities. Most importantly, advanced technology has enabled humans to perform mathematical calculations, experiments, and research that allow further industrial advancements as we gain insight into the world around us. In fact, throughout the past year alone, mathematicians have seen progress in problems and hypotheses such as the Riemann hypothesis, the Ramsey theory, and the Collatz conjecture.

However, there is still a limit to the capabilities of today's technology. Although artificial intelligence is developing, numerous problems often have too many conditions and cases for today's supercomputers to reach the "optimal" solution within a reasonable amount of time. Therefore, it's necessary to find a reasonable solution that can act as a basis for further research on the problem.

The solvability of a problem is determined through the usage of computational complexity. The application of computational complexity goes back to 1936 when digital computers had just begun to develop: mathematician Alan Turing had developed the Turing machine as a theoretical computational model. [1] Although it was capable of performing computations, it failed to account for the amount of time or memory needed by a computer. Therefore, computer scientists Juris Hartmanis and Richard E. Stearns found the concept of computational complexity, which allows mathematicians to measure time and space as a function of the length of the algorithm's input. Computational complexity proposed a formal criterion for what it meant for a problem to be feasibly "decidable" or "solvable". [2]

As the usage of time polynomials in the input size was applied to computational complexity, the concepts of 'Polynomial Time'(P) and 'Non-Deterministic Polynomial Time'(NP) were created within the idea of computational complexity. An algorithm is to be of Polynomial Time if "its running time is upper bounded by a polynomial expression in the size of the input for the algorithm, i.e., $T(n) = O(n^k)$ for some positive constant k" where n is the complexity of the input. In other words, P indicates that it would be possible to run an algorithm within a reasonable amount of time, making the problem "decidable". NP indicates that it would be impossible for computers to run an algorithm within a reasonable time, making the problem "undecidable". Today there are numerous NP problems that mathematicians have yet to solve. [2]

A famous Non-Deterministic Polynomial problem is the Traveling Salesman Problem(TSP). The TSP aims to find the optimal route for a salesman "who starts from a home location, visits a prescribed set of cities, and returns to the original location in such a way that the total traveling distance will be minimal and each city will have been visited exactly once." A reasonable solution to the TSP requires the usage of heuristics. [3]

Heuristic algorithms were developed in order to yield a solution more efficiently than traditional methods, with the risk of error. Yet, effectively using heuristics can provide a basis for further application of optimization algorithms, allowing more accurate estimations of the "optimal" solution. Some instances of heuristic algorithms are the 'Nearest Neighbor Greedy' algorithm (NN), and the 'Lin-Kernighan' algorithm. The Lin-Kernighan algorithm and NN differ in their approaches to a problem. Lin-Kernighan takes various possibilities into consideration through constant experimentation in an attempt to find a more "optimal" solution. [3] Meanwhile, NN simply makes the locally optimal decision at each stage of the algorithm. Let's put this into the context of the TSP: if there were 4 cities (A, B, C, and D), and the salesman started at A, Lin-Kernighan recognizes that although C may not be the closest to city A from the salesman's perspective, in the bigger picture, it can formulate a shorter route. However, NN ignores this prospect, and if B is closest to A, then the salesman will move to B and find the closest city in terms of B. This research paper uses Mathematica, a computing system, as a basis for extensive exploration dealing with heuristics, comparing the effectiveness of the Nearest

Neighbor algorithm, the Lin-Kernighan heuristic, and the FindShortestTour algorithm (one offered by Mathematica's coding base) within various scenarios. [4]

1.2 Design of Research

The paper is organized into three separate sections (excluding the introduction and the conclusion): 1) An Introduction to the TSP 2) Approximation Methods 3) Evaluation. To start off, the paper gives a more formal explanation about what the Traveling Salesman Problem is about and it discusses the progress that mathematicians have already made within the TSP in regards to their various approaches to the problem, and what algorithms or ideas they were able to develop through their research. "Approximation Methods" takes a deep dive into the three algorithms (Nearest Neighbor, Lin-Kernighan heuristic, and FindShortestTour), which is important in providing information and context for the way in which the experiment is carried out, with various different trials and modifications made to the experiment to test different parts of the algorithms out. This leads into the evaluation, which gives a simple explanation as to how the algorithms operated using different codes (this paper uses pseudocodes for its explanation), how the experiment was executed, and the results of the experiment.

2 AN INTRODUCTION TO THE TSP

2.1 Formal Explanation of the TSP

The Traveling Salesman Problem asks to find the shortest route or tour for a salesman starting from a certain city, visiting a given group of cities, and then returning to the starting point of his

tour. [More generally, given an n -by- n symmetric matrix $D=(d_{ij})$, where d_{ij} represents the 'distance' from i to j , arrange the points in a cyclic order in such a way that the sum of the d_{ij} between consecutive points is minimal.] In the grander scheme, we want to arrange the order in which the salesman travels so that the sum of the distances between the consecutive "points" is minimal. Because there is a finite number of possibilities to consider, the problem aims to devise a method or plan that could effectively and efficiently decide on the most optimal arrangement of cities and is also efficient for larger numbers of cities that can complicate the situation and increase the possible arrangements or tours. Acknowledging the complication behind the calculations that may produce an "optimal" or close to the "optimal solution," while this paper doesn't claim to discover new algorithms or methods that may produce a fresh approach to the problem, it aims to record the previous historical progress that has been made regarding the TSP, as well as to compare the various algorithms that have been associated with the problem. [5]

2.2 Historical Methods of the TSP

2.2.1 The Knight's Tour Problem

The Knight's Tour is a problem that was first posed over 1,000 years ago and studied in the realm of graph theory. Its objective was the figure out a route that a knight can follow on an 8 by 8 chessboard and visit every square on the board exactly once. What made the problem challenging at the time, were the 1.305×10^{35} different potential tours which didn't even include any "blind alleys" or "dead ends," making the figure

1.305×10^{35} only an upper bound on the number of tours possible. The significance of the Knight's Tour Problem was that it utilized the "depth-first search with backtracking," a method that is similar to today's Greedy Algorithm. It suggested that one try every possible sequence, backing up when there was a dead end, and trying a different route. [6]

2.2.2 *Hamilton's Icosian*

The Icosian game is a mathematical game invented by William Rowan Hamilton in 1857, with the objective of visiting each and every vertex of a dodecahedron only once, to ultimately arrive back at the starting point. In order to achieve the objective, Hamilton invented the Icosian Calculus, an algebraic tool of non-commutative algebra that was capable of a geometric interpretation, being able to study symmetry properties. Hamilton's Icosian Calculus was important in regard to how it utilized graph theory, something that directly falls under the Traveling Salesman Problem, as the salesman searches for the most optimal and effective tour with given limitations. [7]

2.2.3 *Integer Programming*

Integer Programming is the field of mathematical optimization in which most, if not all, of the variables within the problem are only restricted to be integers. This idea links Integer Programming with combinatorial optimization, as combinatorial optimization looks to find the most optimal subset of items. Items can include, for instance, decisions that collectively lead to a certain result, similarly to how the Traveling Salesman Problem operates: the salesman is

making a series of decisions as to what city to visit next to ultimately produce a set of decisions that yields the shortest route possible. Requiring all problem variables to be integers is what significantly increases the difficulty of a problem. However, mathematicians have been developing different methods to solve such problems, such as the cutting plane method. The Cutting-Planes method was first proposed by George Dantzig, Ray Fulkerson, and Selmer Johnson in 1954 as a means to solve the Traveling Salesman problem, being able to effectively compute accurate lower bounds for the TSP. [8] The work of these three mathematicians also is important in the sense that it was the first time that the Cutting Planes method was utilized to solve a combinatorial problem. Before diving into their work surrounding the Cutting Planes method, let's observe their linear-programming relaxation to the TSP.

2.2.3.1 *Linear-Programming Relaxation*

Linear-programming relaxation is what creates another problem that removes the integer-constraint surrounding the variables of the original problem. The relaxation of an integer linear program is important because it is able to transform an NP-hard optimization problem surrounding linear programming that would be difficult to solve through computations within polynomial time. Instead, we are able to produce a related problem that would be solvable in reasonable time, with the solution of this "alternate problem" being able to give us information about the solution to the original problem. [9]

If we let x be the characteristic vector of a tour, then x satisfies

$$x(\delta(S)) = 2, \text{ for all } v \in V$$

Because every disjoint union of circuits that meet all the nodes or points will appear in the solution set, it is safe to say that tours aren't the only integer solutions to this system. Therefore, we need to add the following inequalities to get rid of such "non-tours" that don't satisfy the TSP,

$$x(\delta(S)) \geq 2, \text{ for all } \emptyset \neq S \neq V$$

as any of the salesman's tours have to enter and leave set S , and will thus contain at least two edges from $\delta(S)$. The inequalities above are our subtour constraints because they enable us to eliminate any subtours to be included in the final solution.

If we combine all of these ideas, we reach Dantzig, Fulkerson, and Johnson's relaxation of the TSP:

$$\text{Minimize } \sum(c_e x_e : e \in E) \quad (1)$$

subject to

$$x(\delta(v)) = 2, \text{ for all } v \in V \quad (2)$$

$$x(\delta(S)) \geq 2, \text{ for all } \emptyset \neq S \neq V \quad (3)$$

$$0 \leq x_e \leq 1, \text{ for all } e \in E. \quad (4)$$

The final line of the mathematicians' relaxation of the TSP is included because it represents the removal of the integer-constraint that is normally present in Linear Programming problems. In other words, it is what makes the inequalities a "relaxation of the TSP." [10]

2.2.3.2 Cutting Planes

If we refer back to the "subtour" linear-program problem

$$\text{Minimize } \sum(c_e x_e : e \in E)$$

subject to

$$x(\delta(v)) = 2, \text{ for all } v \in V$$

$$x(\delta(S)) \geq 2, \text{ for all } S \subseteq V, S \neq V, S \neq \emptyset$$

$$0 \leq x_e \leq 1, \text{ for all } e \in E$$

And tried to solve the problem directly, we would encounter a significant problem: the number of inequalities would be about "the same as the number of distinct subsets of cities, or about $2^{|V|}$." Even if we were to limit ourselves to sets S that satisfied the inequality $|S| \leq |V|/2$, we would still need $2^{|V|-1}$ inequalities, making the overall problem overwhelmingly complex. [10]

Yet, Dantzig, Fulkerson, and Johnson dealt with such inequalities by using the Cutting Planes method. The Cutting Planes method in the TSP begins with the solving of the linear-programming problem (1), (2), and (4). If the "optimal" solution of this problem is a characteristic vector of a tour, then this solution would be the solution to the TSP. However, if it isn't, then we will search for subtour constraints (3) violated by the solution found earlier. Such constraints that are found are added to the initial linear program to then be solved again for another possible "optimal solution." This procedure must continuously be repeated until a solution is obtained that doesn't violate any of the subtour constraints within the linear program. If such a solution is ever obtained, it would also be the solution to the original linear-programming problem. If such a solution continues to not be obtained, then we must continue adding some violated subtour constraints in order to obtain the next linear program.

Overall, the Cutting Planes method requires the finding and utilization of not only an efficient means of checking an optimal solution to a relaxed problem (to test whether it violates any of the subtour constraints from (3)), but also an

efficient way to solve the linear-programming problems that are created through the addition of subtour constraints. The method's reliance on a continuous method to an already relaxed problem is what made it overly complex for mathematicians to rely on to produce an accurate result.

2.2.4 Branch and Bound Method

The Branch and Bound Algorithm was first proposed by A. H. Land and A. G. Doig in 1960 for discrete programming. Its premise is to find the optimal solution within a complete space full of potential solutions by taking more of a "process of elimination" approach to the solutions. The algorithm creates two new nodes, dividing the solution space into smaller subsets, with relative upper and lower bounds for each node, or "subset." It uses the bounds for the function to be optimized with the current best solution to the problem in order to gain a better understanding as to the range of the solution. This allows mathematicians to eliminate certain nodes that fall out of this "spectrum" of possible answers. [9] The Branch and Bound Algorithm can be simply represented by the following graph:

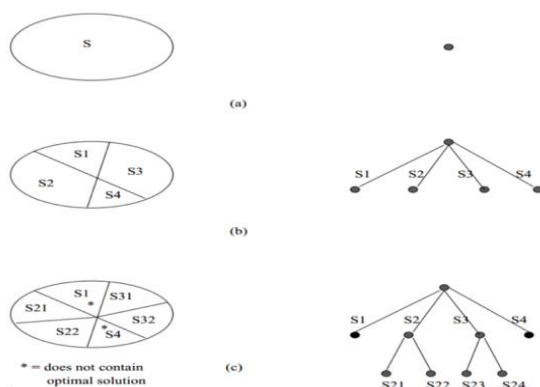


FIGURE 1. Visualization of how the Branch-and-Bound method gradually divides the solution space more in order to ultimately create a range in which the solution is in.

3 APPROXIMATE METHODS

3.1 Nearest Neighbor

The Nearest Neighbor Algorithm (NN) fits in a broader collection of greedy algorithms. Greedy algorithms are algorithms that find a solution by honing in on a few scenarios instead of looking at the grand scheme of the problem. This idea is analogous to running a kilometer. Instead of pacing oneself for the longer kilometer, the algorithm warrants that one pace oneself as if he or she were running 5 sprints of 200 meters.

3.1.1 Explanation of Greedy Algorithms

Greedy algorithms are algorithms that incrementally build up a solution to P by making a sequence of choices. Such choices are not independent of one another, meaning that these choices can restrict the range of choices that would be able to be made later into the sequence. This restriction of the range of choices that can be made is significant in the grander scheme, as it opens the possibility for the "most optimal" sequence to be blocked out from being achieved. While it can be said that we can be more careful to avoid making a choice that is incompatible with optimality, as problems get more complex with more possible "sequences," it becomes more difficult to predict the effect of each choice; this would pull us into a trial-and-error search that would be highly inefficient. At the end of the day, the ultimate objective of such algorithms is to compute and find the "most optimal sequence" effectively, regardless of the complexity of the problem. Greedy algorithms focus on the idea of reaching a solution with great efficiency, though it may not be the "optimal" solution that is being solved for.

Greedy algorithms are applicable to problems that incorporate a choice criterion: a rule for making a choice at each step that is guaranteed to eventually produce an “optimal solution;” it is such choices that collectively form the sequence mentioned earlier. More specifically, problems that revolve around choice criterion usually involves the idea of making a series of feasible choices that minimizes or maximizes a “local measure of goodness.” Overall, the greedy algorithm makes the locally best choice at each step.

3.1.2 The Nearest Neighbor

The Nearest Neighbor algorithm is a greedy algorithm that performs a form of proximity search, being able to efficiently find the nearest point, within a given set of points, to a given point. The algorithm surrounds the idea of traversing in surrounding neighborhood graphs $G(V, E)$, in which point $x_i \in S$ is uniquely associated with a vertex $v_i \in V$. The finding of the nearest points to the point p in the set S takes the form of computing the vertex in the graph $G(V, E)$. The algorithm starts from an enter-point vertex $v_i \in V$, computes the distances from the point to each surrounding vertex within its proximity $\{v_j: (v_i, v_j) \in E\}$, and then moves to the newly selected vertex based off of the one that shares the shortest distance from the enter-point vertex. This newly selected vertex becomes the new enter-point vertex. This algorithm continues to run until there is a local minimum that is reached.

The Nearest Neighbor algorithm fits into the category of greedy algorithms because of

how, while each point within an overall sequence of points is independent from each other, the Nearest Neighbor algorithm chooses its next enter-point vertex by searching for the nearest point to its current enter-point vertex. In other words, the algorithm fails to acknowledge the possible consequences of merely focusing on one city at a time to gradually produce its own “optimal” tour. This allows the algorithm to be ran with great efficiency within a short time span but decreases the “optimality” of the final length of the tour, producing a much longer tour than other algorithms such as the Lin-Kernighan heuristic.

A sub-category of the NN, which this paper’s computations don’t include, is the ‘Nearest Neighbor Algorithm from Both End-Points’ (NND). The NND aims to provide a more effective tour while maintaining the quick solving procedure of Greedy algorithms by focusing on a few cities around the salesman. Let’s say that the salesman starts at City A and visits A’s neighboring City B. Then the next city (City C) would be determined by finding the nearest city to both cities A and B. This would be determined by adding the distance between A and C and the distance between B and C. Again, City D would be dependent on Cities B and C, repeating the series of computations until the salesman reaches every city and arrives back at its starting point.

3.1.3 The Nearest Neighbor in the TSP

Fitting this into the Traveling Salesman problem, the Nearest Neighbor greedy algorithm is one that sectionalizes the larger set of cities to find the shortest possible distance between a given city, and the next city it will travel to. In other words, if

the algorithm were to ask the salesman to start from point p_0 , then the algorithm will look to find the closest point to point p_0 , p_1 , and then find the closest point to p_1 , p_2 , and so on and so forth. Connecting this back to the kilometer-run analogy, each group of cities would represent 1 of the 5 sprints of 200meters, something that is definitively overly simple and idealistic. This simplistic approach is what the Greedy algorithm warrants, as its close-mindedness is what enables it to reach a solution quickly and effectively (though it would not be a highly optimal solution).

The NN fits in this broader definition, as it suggests that the salesman start at any city, and then move to the nearest city that hasn't been visited, taking each city at a time. This process is repeated until the salesman reaches all of the cities and arrives back at its starting point. Within this concept, the NN would compute the distance or cost between the cities surrounding the salesman and direct him to the city with the shortest distance or lowest cost, repeating these computations for each and every city. Again, as the Greedy Algorithm suggests, the simplistic computations that the algorithm executes allows it to reach solutions that may be somewhat realistic.

3.1.4 Example of the Nearest Neighbor

In order to demonstrate what the Nearest Neighbor algorithm does, let us try solving for the shortest tour within the following set of points:

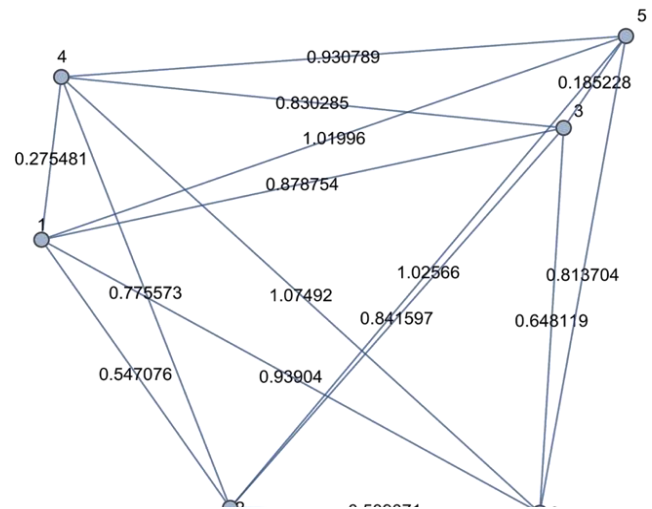


FIGURE 2. Distance between six randomly generated points

If we were to establish the starting point as point 1, the next point that would be traveled to, would be point 4, as it is the closest point to point 1, with a distance of 0.275481. Then it would be point 2, point 6, point 3, and point 5, ultimately producing a route with a tour length of 3.41343 (including the travel back to point 1 from point 5). While this may be one of the shortest tours, it isn't as short as a route that would be produced by Mathematica's FindShortestTour, which would produce a sequence of points 1, 2, 6, 3, 5, 4, and 1 again to produce a route with a tour length of 3.09576.

3.2 Lin-Kernighan Heuristic

The Lin-Kernighan algorithm is a heuristic algorithm focused on the idea of improving a tour through the interchanging of paths. The overall situation of the TSP is set up so that each path the salesman takes between cities make up one edge. These edges add up to the k edges that are marked as potential points of interchange. The idea of the LK search heuristic is analogous to the idea of trial and error, as the algorithm runs through the endless potential interchanges that

can occur, trying to test out what makeup of the k -edges would be the most efficient path to visit each of the n cities. The exchanges that occur may include joined edges, with each joined set of edges making up a single set within the two sets of interchange. In order to simplify the search procedure, there are various conditions that are imposed on these sets of interchange. Yet, before getting into the simplifications of the search process let's describe the situation by starting off with several definitions.

A δ -path in a graph G on k nodes is a path that contains k edges alongside $k + 1$ nodes. These nodes are all distinct to one another, except for the last one, which will be a point that was already encountered before in the sequence of points, considering the fact that the tour has to return back to its original point at some place in time.

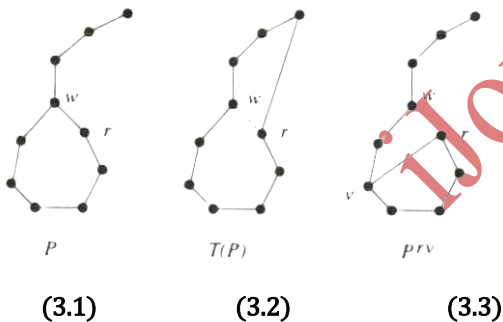


FIGURE 3. δ -path [10] – A tour is a δ -path if the last node is the same as the first node

“If P is a δ -path which is not a tour, then we can obtain a tour $T(P)$ ” as shown above. We would remove the edge wr , and instead connect r to the “first point” at the top of figure 3, thereby obtaining the edge set of a tour.

“Suppose that P is a δ -path which is not a tour. Again, let w be the last node and let wr be the first edge of the subpath of P between the two occurrences of w .” If we were to remove wr from P , then we would essentially be getting a path that ended with r . However, if we were to add another edge rv as shown in the Figure 3, then we would obtain a new δ -path P^{rv} ending at v . This represents the operation of an rv -switch.

“The Lin-Kernighan heuristic starts with a Cour and then constructs a sequence of non-tour δ -paths, each obtained from the preceding one by an rv -switch. For each δ -path P so produced, it computes the cost of $T(P)$. If this is better than the best tour known, then it is ‘remembered’ When the scan is complete, it replaces the starting tour with the best tour found in the scan.” [10]

Simplification of the overall search process is necessary in order to reach the most optimal solution possible within a reasonable amount of time. This set of criteria can be divided into 4 main criteria: 1) The sequential exchange criterion 2) The feasibility criterion 3) Positive Gain Criterion 4) The disjunctive criterion

3.2.1 The Sequential Exchange Criterion

At the i 'th stage, a sequence of $x_i, y_i, x_{i+1},$ and y_{i+1} must be existent. Therefore, if t_i represents each of the vertices on the graph of the tour, and $x_i = (t_{2i-1}, t_{2i})$, then $y_i = (t_{2i}, t_{2i+1})$, and $x_{i+1} = (t_{2i+1}, t_{2i+2})$. Therefore, it is necessary that $x_1, y_1, x_2, y_2, \dots, x_f, y_f$ is a closed sequence of chains, meaning that it would form the full tour.

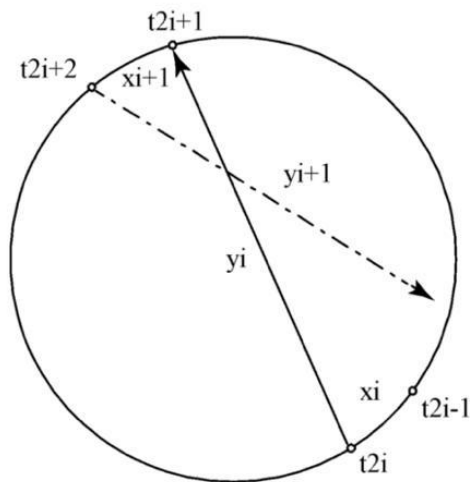


FIGURE 4. Illustration of the sequential exchange criterion

3.2.2 The Feasibility Criterion

The feasibility criterion ensures that a full tour, that connects every city to each other, is formed even after the LK path interchanges. If each city were to be represented in the form c_n (with n representing the order in which the salesman visits the city), then the link x_n that connects c_{2n-1} and c_{2n} , represented through the notation $x_n = (c_{2n-1}, c_{2n})$, must be chosen in a manner that if c_{2n} were to immediately connect to c_1 (the starting point), then a full tour must form without any breakage in between the cities. The establishment of this idea is significant in ensuring that we end the LK search heuristic with a full tour instead of arbitrary sub-tours that do not interconnect with one another. The feasibility criterion is applied for the heuristic when $n \geq 3$, so that the LK exchange algorithm can immediately form a tour so $c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow c_1$, forming a full tour. This criterion enables a faster execution and simplified coding of the LK heuristic algorithm.

3.2.3 Positive Gain Criterion

The Positive Gain Criterion ensures that each exchange in the algorithm reduces the “cost” of the tour, represented by $c()$, and results in a net gain, represented by G_r . Let’s say that the set of links that are being exchanged is represented through set X , which contains the links x_r , and that the set of links that are replacing set X is represented through set Y , which contains the links y_r . It can then be said that set $X = \{x_1, x_2, \dots, x_{r-1}, x_r\}$ is being replaced by set $Y = \{y_1, y_2, \dots, y_{r-1}, y_r\}$. In this case, paired links of $(x_1, y_1), (x_2, y_2)$, and up to (x_r, y_r) are replacing each other, making it so that the overall net gain is equivalent to the sum of the gains of each pair. Thus, if the gain of each pair is represented by $g_i = c(x_i) - c(y_i)$, then the overall net gain of the exchange is represented by $G_r = g_1 + g_2 + \dots + g_r$. It is required that each g_i is always positive, because it guarantees that the overall exchange is positive, and ensures that each exchange made within the algorithm contributes towards creating a more optimal tour. Even if the overall exchange between X and Y can be positive, one pair (x_i, y_i) could still have produced a negative gain, which the Positive Gain Criterion is able to avoid. Overall, the Positive Gain Criterion warrants improvement in the tour, and somewhat acts as a “stopping criterion” that can prevent harmful exchanges.

3.2.4 The Disjunctive Criterion

The Disjunctive Criterion warrants quicker execution and simplification of the algorithm coding by requiring that the sets X and Y to be “disjoint.” In other words, any link between x_i and x_{i+1} that is broken by set Y cannot be joined back

again the future, and vice versa, that is, considering that link as “inefficient.”

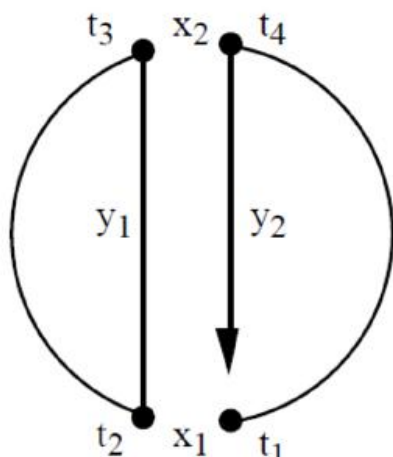


FIGURE 5. The other alternative for t4 results in two disconnected loops, so the past to t1 must not be joined, either between t1 and t4 or t2 and t3

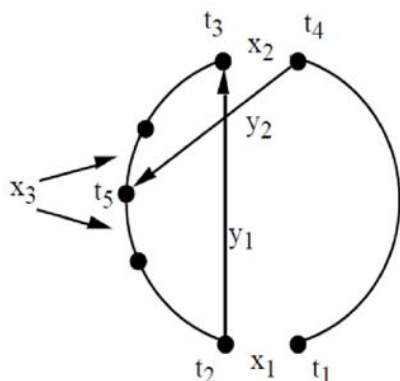


FIGURE 6. t4 is connected to t5 between t2 and t3

4 EVALUATION

4.1 Computation of Algorithms

4.1.1 Nearest Neighbor

If we refer back to section 3.1, we know that the Nearest Neighbor algorithm starts at an assigned point and moves to the nearest point available (an available point is one that hasn't been encountered by the traveling salesman), repeating this step for a finite number of times, until all the points have been encountered exactly

once. After the salesman has gone through all the available points, he moves back to the initially assigned point to finalize the route.

Before demonstrating the Nearest Neighbor algorithm through Mathematica's coding basis, we can formulate a universal algorithm that accurately portrays the operation of the evaluation method. Let's begin by assigning variables to shorten the overall complexity of the algorithm:

Symbol	Definition
<i>cur</i>	Current Point
<i>prev</i>	Previous Point
<i>NN</i>	Nearest Neighbor
<i>POI</i>	Point of Interest
<i>L</i>	List of POI
<i>DIST(x,y)</i>	Distance between x and y

With such symbols, we can represent the operation of the Nearest Neighbor algorithm as follows:

Algorithm 1. Nearest Neighbor

- 1: Initialize *cur*
- 2: Initialize *dist*
- 3: Set *prev* to *L*[0]
- 4: $a = L[0]$
- 5: Repeat until *L*
- 6: $cur = NN \text{ of } prev$
- 7: $dist = dist + DIST(prev, cur)$
- 8: $L.pop(prev)$
- 9: $prev = cur$
- 10: $dist = dist + DIST(cur, a)$

To start off, the initialization of the current point and the distance are necessary because it enables us to begin with an empty tour, which will set up

the following lines of code. Following this initialization, setting the previous point to the first element in L establishes the first point within the sequence of points that make up the tour. Setting the first point as the previous point enables us to begin capturing the lengths between consecutive points in the sequence, starting from the lengths between the first two points. In other words, if $L[0]$ were to start off as cur , then we would not have any previous point to work with, as line 6 reassigns Cur as the nearest point of $prev$; this would ultimately produce a loophole in the algorithm. Lines 5 to 9 create a loop under which we are able to constantly add on the length between the following points in the sequence (from point 1 to 2, point 2 to 3, point 3 to 4, etc.), and end when the salesman encounters all of the points in the given set of points. Then, line 10 brings the salesman back to the starting point, as it is also accounted into the final distance of the tour. Line 10 relies on the defining of $a = L[0]$ because after running through line 5, L would come out empty, necessitating $L[0]$ to be placed under another variable to be used in line 10.

Now, if we point to the representation of the Nearest Neighbor algorithm within Mathematica's coding basis, then we can get the following:

Algorithm 2. Nearest Neighbor in Mathematica

```

1:  outputData
    = shortestTourTrial["NN", citiesGraph, 1]
2:  tourNodes = outputData["TourNodes"];
3:  GeoGraphics[ {GeoPath[StateCapitals[
    [tourNodes]]], Red, PointSize[.01],
    Point[points]} ]
4:  N[outputData["TourLength"]]
  
```

Note that the variables above are defined through LKSearch.nb, ShortestTourUtil.nb, and ShortestTourTrial.nb

Now, if we dissect what each line of code means, line 1 is what classifies the code as one that represents the Nearest Neighbor algorithm through the "NN" that falls under shortestTourTrial. The term "NN" in these lines of code is what represents a wrapper function, which the shortestTourTrial helps run. In addition, the *citiesgraph* helps establish the points that are being assessed alongside *StateCapitals* in line 3, and the 1 that follows means that the tour will begin from point 1. Line 2 sets up line 3, which helps with the plotting of the overall tour through a map (which we will look into further through sections 4.2, 4.3, and 4.4). Finally, $N[]$ is what gives a numerical value of the expression in the first line, thereby ultimately giving us the tour length.

4.1.2 Lin-Kernighan

If we refer back to section 3.2, we found out that the Lin-Kernighan algorithm demonstrates more complexity relative to the Nearest Neighbor Algorithm, hence why we will simply look into the Lin-Kernighan algorithm within Mathematica, as shown below:

Algorithm 3. Lin-Kernighan in Mathematica

```

1:  outputData
    = shortestTourTrial["LK", citiesGraph]
2:  tourNodes = outputData["TourNodes"];
3:  GeoGraphics[ {GeoPath[StateCapitals[
    [tourNodes]]], Red, PointSize[.01],
    Point[points]} ]
4:  N[outputData["TourLength"]]
  
```

Note that the variables above are defined through LKSearch.nb, ShortestTourUtil.nb, and ShortestTourTrial.nb

The only notable difference between the Lin-Kernighan heuristic and the Nearest Neighbor algorithm in Mathematica, is that the wrapper function is now "LK", and the that there isn't any number following *citiesGraph*. It is difficult to demonstrate the differences within the wrapper functions themselves that highlight the different functionality of the two approximation methods in order to maintain the conciseness of the paper. However, it can be pointed out that the *Lin – Kernighan* algorithm doesn't depend on a certain starting point to operate, rather it runs the same way regardless. Yet, it is also important to note that tour lengths produced by the Lin-Kernighan heuristic often change, as the modifications that are made to a given tour are often different, thereby creating a different result almost every time the same code is ran for the same set of points.

4.1.3 FindShortestTour

Noting that the FindShortestTour operation is a built-in code within Mathematica, let's jump right into the FindShortestTour in Mathematica, as shown below:

Algorithm 4. FindShortestTour in Mathematica

```
1:  outputData
    = shortestTourTrial["Mathem
      atica", citiesGraph]
2:  tourNodes
    = outputData["TourNodes"];
3:  GeoGraphics[ {GeoPath[StateCapitals[
```

```
[tourNodes]]], Red, PointSize[.01],
  Point[points]} ]
```

```
4:  N[outputData["TourLength"]]
```

Note that the variables above are defined through LKSearch.nb, ShortestTourUtil.nb, and ShortestTourTrial.nb

If we compare the FindShortestTour algorithm in Mathematica with the Lin-Kernighan algorithm in Mathematica, it can be seen that the only difference is the wrapper function of "Mathematica." This is because, just like the Lin-Kernighan heuristic, Mathematica's FindShortestTour runs the same regardless of whether the algorithm is assigned a first point or not, as the algorithm will always look for the route that provides the lowest cost, regardless of where the salesman would have to start.

Why not just run the FindShortestTour function in Mathematica in order to receive the tour length?

In order to maintain consistency within the codes that find its own "optimal" tour length when given a certain set of points. Underneath the wrapper function "Mathematica," is the following:

Algorithm 5. Wrapper function "Mathematica"

```
1:  computeMathematicaTour[graph] :=
2:  Module[{
      solution, tourNodes, tourEdges, outputData}
3:  solution = FindShortestTour[graph];
4:  tourNodes = solution[[2]];
5:  tourEdges = Join[Map[tourNodes[ [# ] -
      tourNodes[ [# +
6:  1]]&, Range[Length[tourNodes]-1]],
      {First[tourNodes]--Last[tourNodes]}];
      outputData["Tour"] = tourEdges;
```

```

7:  outputData["TourNodes"] = tourNodes;
8:  outputData["TourLength"] = solution[[1]];
9:  Return[outputData];];

```

Note that the variables above are defined through LKSearch.nb, ShortestTourUtil.nb, and ShortestTourTrial.nb

Through the wrapper function in algorithm 5, we are able to note the inclusion of *FindShortestTour* which simplifies the wrapper function a lot more relative to the Nearest Neighbor and Lin-Kernighan approximation methods. Now that we have managed to go through all of the algorithms that produce the "optimum" tour length within a given set of points, let's dissect what *shortestTourTrial* exactly means, and why the code is formatted in that manner within algorithms 2, 3, and 4:

Algorithm 6. shortestTourTrial function

```

1:  shortestTourTrial[strategy, graph, startVertex_, 0, maxT: 2]:=
2:  Module[{trialOutputData},
3:  Switch[strategy,
4:    "NN", trialOutputData
      = computeNNTour[graph, startVertex],
5:    "LK", trialOutputData
      = computeLKTour[graph, maxT],
6:    "Mathematica", trialOutputData
      = computeMathematicaTour[graph],
7:    _, Message["Strategy '1' not recognized!",
      strategy]];
8:  trialOutputData["Strategy"]
      = strategy;
9:  trialOutputData["Graph"] = graph;
10: Return[trialOutputData]]

```

Note that the variables above are defined through LKSearch.nb, ShortestTourUtil.nb, and ShortestTourTrial.nb

If we dissect what is going on within Algorithm 6, starting from line 1, we are able to see the exact format of the *shortestTourTrial* function. The most important part of the function, the strategy or algorithm being used in order to produce the tour length, is specified from lines 3 to 6, where they come up with wrapper functions "NN," "LK," and "Mathematica," to represent the different functions *computeNNTour*, *computeLKTour*, and *computeMathematicaTour*. These 3 functions are defined through the different files mentioned underneath the algorithm, with similar formatting to algorithm 5, which defines how the function *computeMathematicaTour* operates underneath "Mathematica."

Following the strategy, the graph is what holds all of the points that the salesman has to travel to, and it therefore dictates the overall possible routes that the salesman can possibly take. Finally, through the *Return* function in line 10, we are able to receive the final tour length for the given set of points.

Now that we have gone over the codes that represent each of the algorithms, and how they fit their algorithms description from section 3, let's take a deeper dive into actually utilizing such codes to collect data and compare the performance of each algorithm within different scenarios: 1) 4 random points 2) US Capitals 3) Cities in Florida.

4.2 Scenario #1: 4 Random Points

To start off, in order to set up the scenario, let's generate and plot the 4 points through the following,

Algorithm 7. Set-up of 4 points

- 1: `locations = {{-3,1},{0,8},{2,-3},{-2,-2}};`
- 2: `graph = constructWeightedGraph[locations];`
- 3: `Show[graph,ImageSize -> Tiny]`

Note that the variables above are defined through

`LKSearch.nb`, `ShortestTourUtil.nb`, and `ShortestTourTrial.nb`

which produces the following graph:

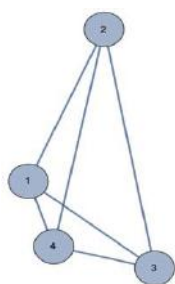


FIGURE 7. Graph of the points (-3,1), (0,8), (2,-3), and (-2,-2), which are represented by points 1, 2, 3, and 4 respectively.

If we refer back to the format of the Nearest Neighbor algorithm, we know that the tour length is dependent on the starting point, which was represented by a certain number. When the points are plotted, each point is labeled a number based off of the order in which the point was plotted, starting from the first point being plotted as point 1.

Now, utilizing the algorithms established earlier, we can figure out the average distance produced by each algorithm within the given scenario, as well as the average amount of time it took for the

code to run. Every code for the scenarios were ran through 10 times, with an average being produced by 10 trials. For this specific scenario, the Nearest Neighbor algorithm code started from point 3, but this will be further touched upon later on into this paper. If we were to represent our data with bar graphs, we would receive the following two graphs:

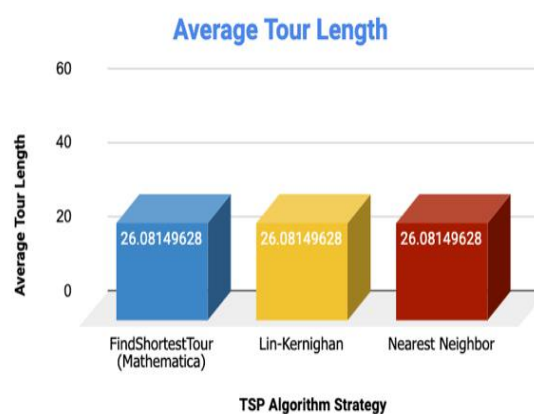


FIGURE 8. Graph representing the average tour length that each algorithm produces through 10 trials (the Nearest Neighbor algorithm started from point 3) within scenario 1.

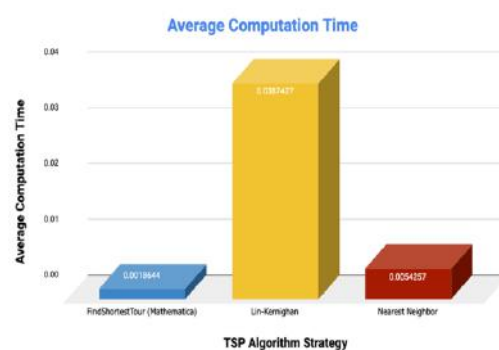


FIGURE 9. Graph representing the average amount of time (from 10 trials) it took each algorithm to run through its codes (the Nearest Neighbor algorithm started from point 3) within scenario 1.

Note that time is always measured in seconds within this paper

At first glance, it may seem that the Lin-Kernighan heuristic provides the worst results, as it provides synonymous results as Mathematica's FindShortestTour and the Nearest Neighbor, but takes more time to produce such results, which is reasonable as there aren't many possible routes that the salesman could take when given 4 points. However, it's important to note that this data doesn't account for the other 3 possible starting points that could affect the average tour length produced by the Nearest Neighbor algorithm:

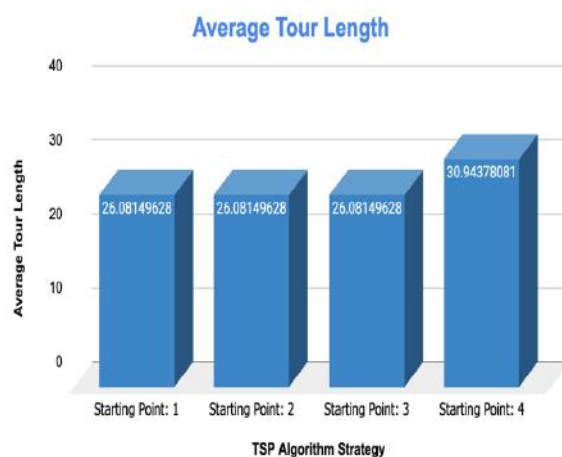


FIGURE 10. Graph representing the different tour lengths achieved by the 4 possible starting points

Through Figure 10, we can note that while 3 of the possible starting points produce the same tour with minimal cost, when the tour starts from point 4, or $(-2, -2)$, then it produces a result greater than the Lin-Kernighan and FindShortestTour methods, ultimately averaging out to 27.29706741 units. Therefore while it can be concluded that the Nearest Neighbor and FindShortestTour methods produce tour lengths in shorter periods of time relative to

the Lin-Kernighan heuristic, it is evident that the Nearest Neighbor still produces the longest tour length, on average, though it is capable of producing the shortest tour within the 1st scenario with only 4 points. In order to further investigate the trends mentioned in scenario #1, let's try increasing the number of cities in the tour, and the overall spacing of such cities.

4.3 Scenario #2: US Capitals

Formulating a scenario surrounding the capitals of the US enables us to get a better understanding regarding how each of the algorithms react as the number of cities within the tour increases. Hypothetically, it would make sense for the Lin-Kernighan heuristic to face the greatest impact in regards to the amount of time it takes for the algorithm to be ran through, as it would have more possible routes to consider relative to the Nearest Neighbor and FindShortestTour, which don't run as complicatedly as the Lin-Kernighan heuristic. However, let's test this idea through numerical data.

Before diving into the data collected, there will be 5 scenarios with a different number of cities under each scenario to observe trends within the given collection of 50 possible cities. If we graph the average tour lengths and computation times (in seconds), then we receive the following:

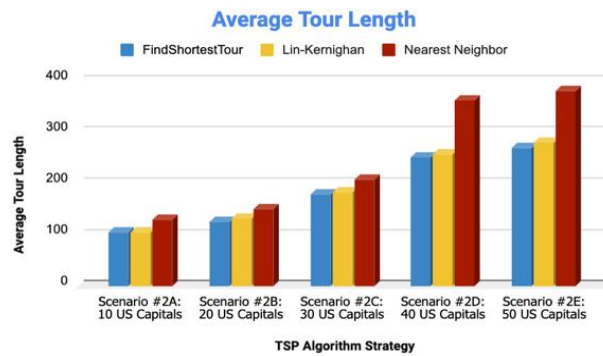


FIGURE 11. Graph representing the average tour length of each algorithm for each of the five scenarios within the 2nd scenario surrounding US capitals.

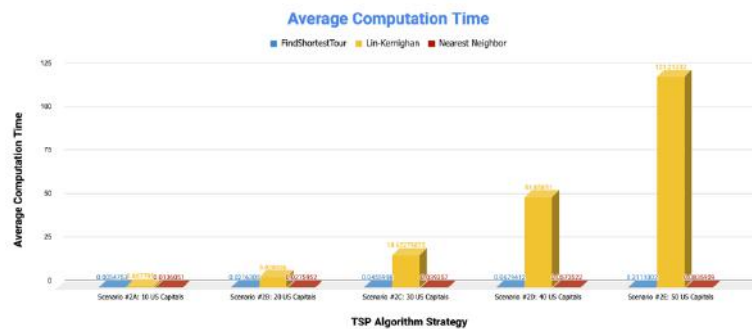
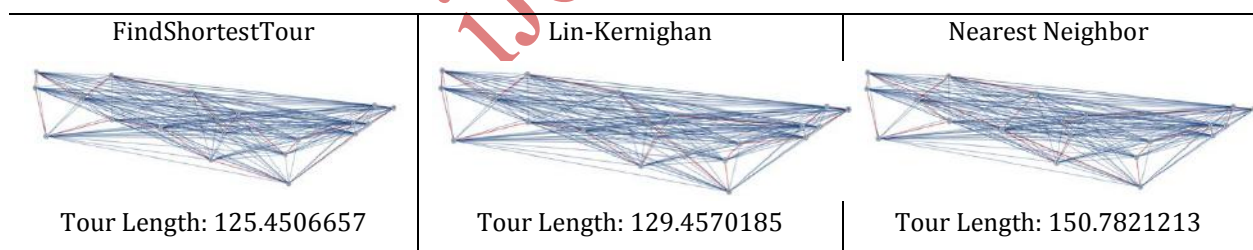


FIGURE 12. Graph that represents the average computation time (from 10 trials). Note that time is always

Scenario #2B: 20 US Capitals



measured in seconds within this paper.

FIGURE 13. Graphs that show the route formed by the respective algorithm. While the blue lines represent all the 20 cities being connected to one another, the red lines represent the exact route the algorithm devised for the salesman to follow.

Overall, it is evident that the Lin-Kernighan heuristic experiences the most drastic changes as the number of cities increases through each scenario, as the algorithm relies on trying out

almost every possible exchange between the connection of points that collectively form the tour, and increasing the number of points increases the number of possibilities that the Lin-Kernighan

algorithm would have to consider. However, it can also be noted that as the number of cities increases, the difference in tour length generated by the 3 algorithms widens, as the Nearest Neighbor's flaws shine through, failing to consider the additional possible routes that are created through the addition of more cities.

The flaws of the Lin-Kernighan and Nearest Neighbor approximation methods can also be seen in Figure 11, as the illustrations show the route provided by each algorithm experiences their own intersections in the middle of the sequence, as the

salesman often ends up following a path around an area that the salesman already encountered. Such unnecessary movement is what differs the

Lin-Kernighan and Nearest Neighbor methods from the FindShortestTour, which seems to get the best of both worlds, with both the shortest average distance and computation time.

To come to the Nearest Neighbor's defense however, the graphs above all show data when the salesman starts at point 10, meaning that there still looms the possibility that the salesman's shortest possible route with the Nearest Neighbor can match that of the Lin-Kernighan or FindShortestTour methods. In order to investigate this, let's collect data using scenario #2E with all 50 capitals being part of the salesman's point of interest. If we try modifying the Nearest Neighbor algorithm so that we can get the tour length if the salesman were to start at each of the 50 points, we would get the following:

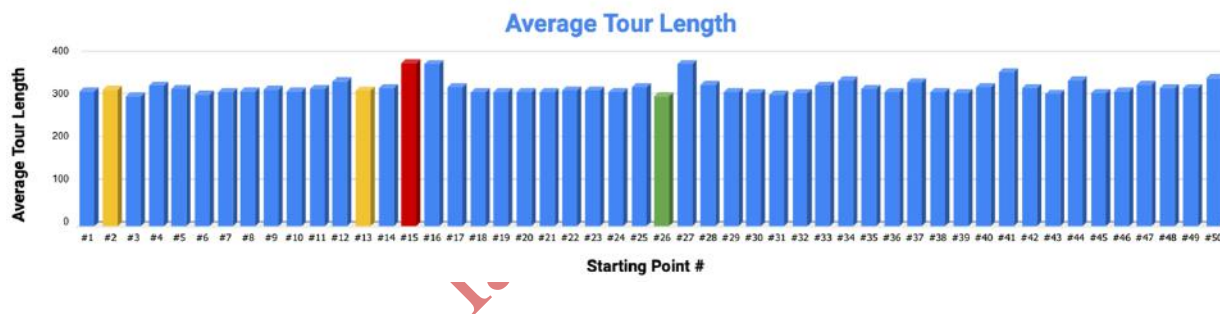


FIGURE 14. Graph of the different tour lengths achieved when the Nearest Neighbor algorithm is ran starting from each of the 50 different possible starting points. The red represents the maximum tour length reached; the green represents the minimum tour length reached; the yellow represents the median.

Scenario #2E: 50 US Capitals – Distance (Nearest Neighbor)			
Mean	Min	Max	Median
324.1618498	302.9797884	382.358161	318.0132337

The bar graph in figure 11 of the Nearest Neighbor represents a tour length of 379.4216407, meaning that it was one of the highest possible lengths achievable through the

50 points. However, even if we were to compare the minimum possible tour length achievable, when the salesman starts at point 26, to the other two methods' average tour length, then the

Nearest Neighbor algorithm would still have the longest tour length of the three. Because of this fact, it is safe to say that the Nearest Neighbor will not be able to ever produce a tour length shorter than the Lin-Kernighan heuristic and FindShortestTour method, given that the number of cities that have to be encountered is fairly high, rather than single-digit numbers of cities. We can conclude that from a time-standpoint, the Nearest Neighbor and FindShortestTour are consistent in their quick computation time, while from a distance-standpoint, the FindShortestTour method consistently provides the shortest tour out of the three methods. To further justify this claim, let's try formulating another scenario: a scenario with just as many cities, if not more, open for travel, within a shorter area of land.

4.4 Scenario #3: Florida Cities

Creating a final scenario surrounding Florida's cities enables us to get a better understanding about how the three algorithms or methods react to changes in the number of cities that the traveling salesman has to encounter, as we will be able to test out more cities within a smaller area, focused on a single state. Therefore, while the tour lengths can be expected to be shorter than the previous scenario revolved around US capitals, it can easily be expected for the computation time to increase. Scenario #3 will be divided into four scenarios: 1) 30 cities 2) 60 cities 3) 90 cities 4) 120 cities. If we were to process the data and produce a graph that represented the changing distance and computation time over the different

numbers of cities, we would get the following:

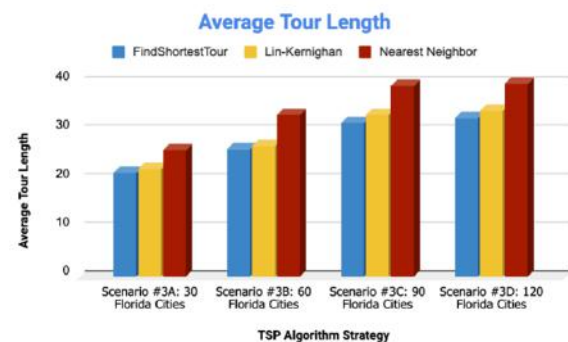


FIGURE 15. Graph of the average tour lengths (averaging 10 trials) provided by each algorithm among the four scenarios within scenario #3.

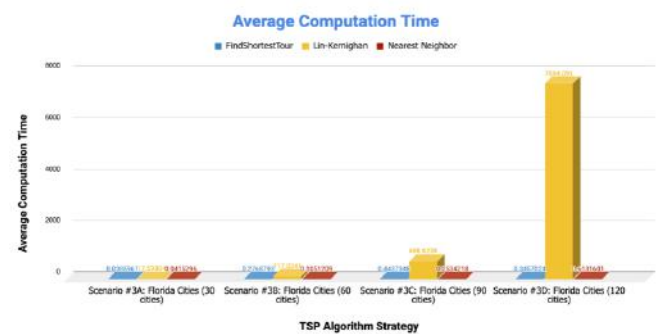


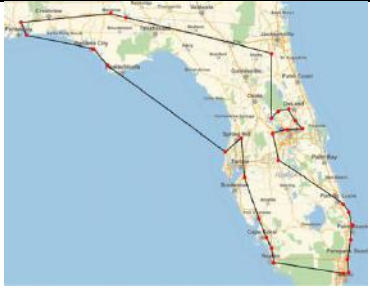
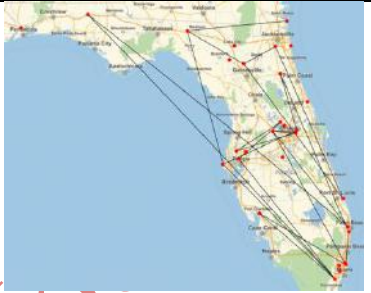
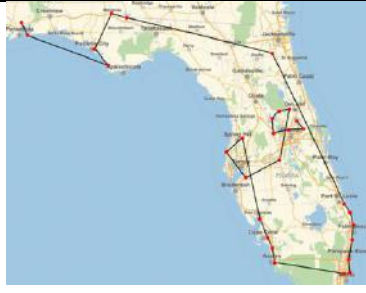



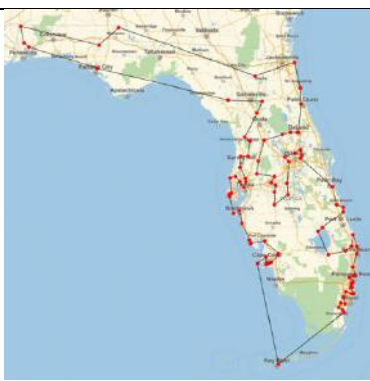
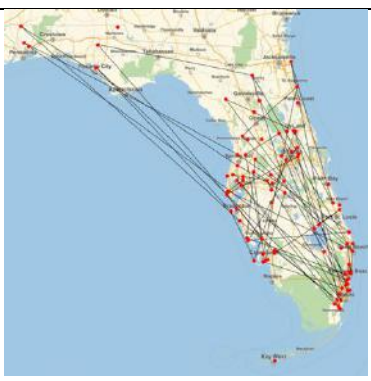
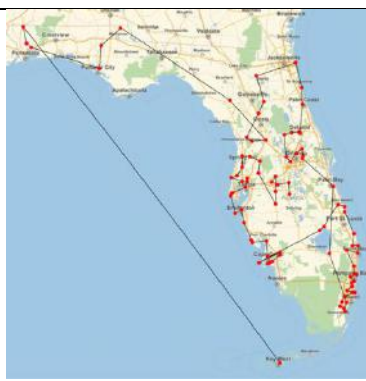
FIGURE 16. Graph of the different computation times (based on 10 trials) of the three algorithms, changing alongside the number of cities.

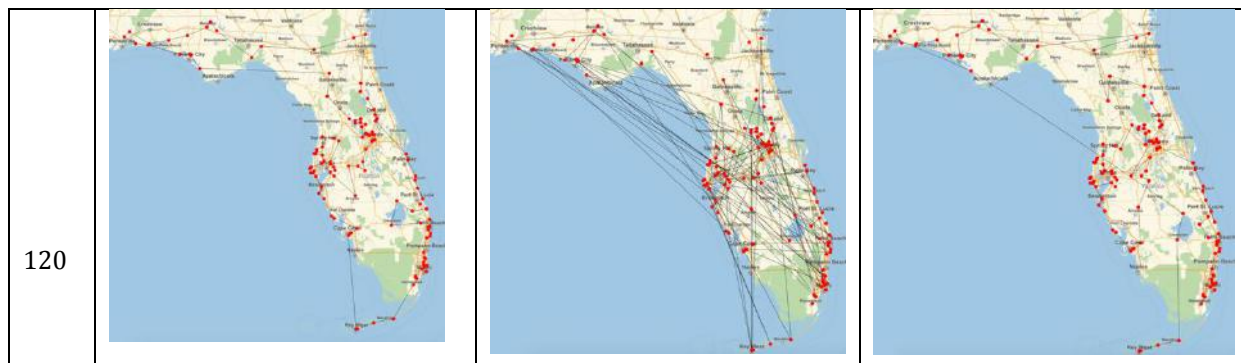
Through figure 15 and 16, we are able to point out how, regardless of the increasing number of cities that have to be accounted for when building an optimal route for the salesman, the Nearest Neighbor and FindShortestTour methods are able to always finish their computations under a second, while the Lin-Kernighan algorithm experiences an exponential increase, taking up to 2 hours and 6 minutes on average to compute a single route for the salesman to follow. More importantly, despite the amount of time that it takes the Lin-Kernighan heuristic to generate such a tour, it consistently faces a fair difference in the tour length produced on an average of 10

trials, relative to the FindShortestTour that is able to consistently find the fastest route and shortest tour out of the three methods in the shortest amount of time. It is also worth noting that the Nearest Neighbor and FindShortestTour methods are able to provide consistent routes (assuming that the Nearest Neighbor's starting point is fixed) that maintain themselves even as the trial number increases. While it is difficult to explain the functionality of the FindShortestTour as it isn't fully disclosed by Mathematica, the Nearest Neighbor algorithm is able to produce the same

result over and over again because it is taking one city at a time, simply looking for the next closest city that the salesman can go to, ultimately only leaving the salesman with one possible route if he were to follow the Nearest Neighbor method from a given point or city.

Now, let's investigate the graphs of the 12 tours that were collectively created by the three algorithms over the course of the 4 sub-scenarios:

Scenario #3: Florida Cities – Graphs			
# of cities	FindShortestTour	Lin-Kernighan	Nearest Neighbor
30			
60			
90			



5 CONCLUSION

Overall, throughout this paper, we were able to utilize 3 scenarios—1) 4 random points 2) US Capitals 3) cities in Florida—in order to compare and contrast the effectiveness and efficiency of Mathematica's FindShortestTour, the Lin-Kernighan heuristic, and the Nearest Neighbor algorithm and how they react to different situations. As a result of the experiments that were carried out, we were able to justify that the Nearest Neighbor and the Lin-Kernighan approximation methods are opposites of one another, with the Nearest Neighbor being able to come up with longer routes in shorter computation time than the Lin-Kernighan heuristic, while the FindShortestTour is the best of both worlds, being able to produce the shortest routes out of the three algorithms within the shortest amount of time.

References

- [1] De Mol, Liesbeth. "Turing Machines." *Stanford Encyclopedia of Philosophy*, Stanford University, 24 Sept. 2018, plato.stanford.edu/entries/turing-machine/.
- [2] Dean, Walter. "Computational Complexity Theory." *Stanford Encyclopedia of Philosophy*, Stanford University, 20 July 2016, plato.stanford.edu/entries/computational-complexity/.
- [3] Lin, S., and B. W. Kernighan, "An Effective Heuristic Algorithm for the Traveling-Salesman Problem." *Operations Research*, vol. 21, no. 2, 1973, pp. 498–516, https://www.jstor.org/stable/169020?origin=JSTOR-Pdf&seq=1#metadata_info_tab_contents, December 18, 2020
- [4] Gözde Kizilates, and Fidan Nuriyeva, "On the Nearest Neighbor Algorithms for the Traveling Salesman Problem", https://page-one.springer.com/pdf/preview/10.1007/978-3-319-00951-3_11, December 18, 2020
- [5] G. Dantzig, R. Fulkerson, and S. Johnson, "Solution of a Large-Scale Traveling-Salesman Problem," <https://www.jstor.org/stable/pdf/166695.pdf?refreqid=excelsior%3A0cb634f5dc863c87107e7b48f262d762>, December 28, 2020
- [6] V. Scott Gordon, and Terrill J. Slocum, "The Knight's Tour-Evolutionary vs. Depth-First Search," <https://athena.ecs.csus.edu/~gordonvs/papers/knightstour.pdf>, December 18, 2020
- [7] Norman Biggs, "The Icosian Calculus of Today," www.jstor.org/stable/20490184, January 4
- [8] "Dantzig, Fulkerson, and Johnson's Cutting-Plane Method," University of Waterloo,

- <http://www.math.uwaterloo.ca/tsp/methods/dfj/index.html>, December 18, 2020
- [9] https://en.wikipedia.org/wiki/Linear_programming_relaxation
- [10] <http://math.mit.edu/~goemans/18433S15/TSP-CookCPS.pdf>
- [11] "Integer Programming." *Integer Programming - an Overview* / *ScienceDirect Topics*,
www.sciencedirect.com/topics/mathematics/integer-programming.
- [12] Jiyao Gao, Branch and Bound, [https://optimization.mccormick.northwestern.edu/index.php/Branch_and_bound_\(BB\)](https://optimization.mccormick.northwestern.edu/index.php/Branch_and_bound_(BB)),

*i*Journals