CS584 Final Project

Analysis of the Nearest Neighbor and 2-Opt Swap Algorithms for approximating solutions to the Traveling Salesman Problem

Scott Rubey

The Traveling Salesman Problem has been the focus of numerous studies, comprising countless man-hours of attention since its conception in the early 1800's. First approached scientifically in the 1930's, such studies have attempted to answer the following question: "Given a list of cities, what is the shortest tour in which each city is visited exactly once, ultimately returning to the city of origin?" Modern-day applications range from the esoteric to the mundane, including such fields as astronomy and DNA sequencing, as well as enterprise logistics (think FedEx deliveries). Given the simplicity of the question, the problem is implausibly difficult to solve with 100% accuracy. The most intuitive method for arriving at the optimal solution is through selecting the minimum tour-distance -- aka "weight" -- from an exhaustive search. While this might seem reasonable given a list of ten or fifteen cities, try to imagine a list of hundreds, or even thousands! For this reason, many modern studies simply attempt to find reasonable approximations against provable lower bounds, with some recent advancements known to produce solutions within 1% of optimal. Implementation of such heuristics can be challenging, however, and even the most time-efficient implementations can prove prohibitive (days, months, even years of processing time) given large input. Thus, for the purposes of this study, we shall explore several easy-to-implement approximation algorithms with varying degrees of complexity: the Nearest Neighbor algorithm, 2-Opt Swap, and the Random Swap (which acts somewhat as an "add-on" to 2-Opt). While there are a number of algorithms known to provide closer-to-optimal results in the average case, the aforementioned strategies can provide surprisingly reasonable results for those who don't require near-perfect precision from complex algorithms.

Testing Procedures

The computer science community typically approaches the Traveling Salesman Problem (hereafter abbreviated as TSP) as a weighted, undirected, complete graph, in which each vertex (i.e. city) is connected to each other vertex by an edge (i.e. road). Edges have values representative of distance. In order to complete a tour, one must return to the starting vertex, thus executing a Hamiltonian Cycle. The problem has been demonstrated to be amongst those known as NP-complete (rather, the decision problem thereof), which signifies that there is no known polynomial time solution that yields 100%-optimal results; hence, our reliance on approximations.

As one might infer from the term "approximation algorithm," the experiments conducted herein were far less concerned with execution time than with the accuracy of the results. To be sure, execution time is not of trivial concern when it comes to large TSP input sets, even when striving for off-perfect results. The tests included in this study ranged from near-instantaneous for smaller datasets (50 to 76 cities) to roughly 46 minutes for the largest (taking the lowest weight of 10 tours of 575 cities) on a 2.5 GHz Intel Core i7 with 16GB 1600 MHz DDR3 memory. Further experiments were conducted on even larger datasets (up to 1379 cities), though with processing times of approximately 2.5 hours per tour, it became infeasible to conduct multiple tests for averaging purposes; such input sizes were quickly discarded.

Testing was conducted using datasets made available through TSPLIB[1], an online resource provided by Dr. Gerhard Reinelt, Professor of Computer Science at the University of Heidelberg, Germany. Dr. Reinelt's collection of TSP datasets has been used in numerous studies since its 1991 release. Its attractive distinction amongst TSP datasets spawns from the fact that each problem includes a mathematically proven lower bound, which makes benchmarking a straightforward proposition (and sports a certain usefulness in debugging faulty algorithm implementations). Each dataset is formulated as a numbered list of vertices, each with Euclidean 2D coordinates[2] representing the city's location. As such, the distance between each city becomes an easy calculation performed at the time the file is loaded for processing.

Due to the availability of a proven lower bound for each dataset, the obvious test metric was the relative percentage-above-optimal tour distance yielded by each approximation algorithm. Benchmarking tests were performed on twenty-five different datasets, ranging from 51 cities to 575, weighted heavily in the 100- to 150-city domain. Five of these datasets contained exactly 100 cities, with two further sets weighing in at 101 and 105. While an assortment of sizes was necessary in order to measure accuracy as correspondent to size (if any such correspondence existed), it was deemed equally important to measure whether accuracy remained constant (or fell within a particular margin of error) for same-sized datasets. Furthermore, several of these datasets "built upon" each other (the KroA and KroB series); each of these provided a listing of 100 cities, then 150 cities (of which the first 100 remained identical to the prior set), and finally 200 (reusing the first 150, as before). This allowed for the study of accuracy as a factor of identical subsets of city data, i.e. whether there was a linear divergence of accuracy, for instance, in up-scaled data.

The Algorithms

[1] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html

[2] While the datasets made available through TSPLIB come in assorted formats (adjacency lists vs. adjacency matrices, Euclidean 2D coordinates vs. latitude/longitude, etc.), this study made exclusive use of Euc_2D adjacency lists; this prevented the requirement for dynamic file-loading functionality while still yielding a diversity of test cases.

The first algorithm tested in attempting to approximate our optimal route was known as Nearest Neighbor. Nearest Neighbor is a greedy algorithm that simply iterates through each vertex in a given graph from a particular starting point, searching for the unvisited neighbor with the lowest edge-weight. The algorithm proceeds as follows:

1) Label all vertices in the graph as unvisited.
2) Select a starting vertex and mark it as visited, adding it to the Route.
3) Consider all neighbors of our starting vertex and find the one with the lowest-weight edge.
4) Set this new vertex as the current vertex and mark it as visited. Add it to the Route, and add the edge's weight to the Route's total weight.
5) Repeat from Step 3 until all vertices have been visited.
6) Complete the Hamiltonian Cycle by returning to the start and adding the appropriate edge-weight to that of the total Route.

It should come as no surprise that the starting vertex can have major implications regarding the total weight of a given Route. Total weight can be, and often is, substantially different from one starting vertex to another. This is demonstrated by the following diagram:

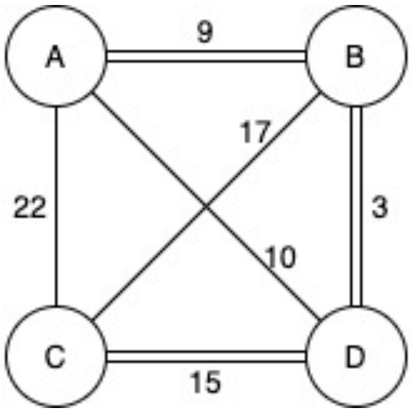

Fig. 1

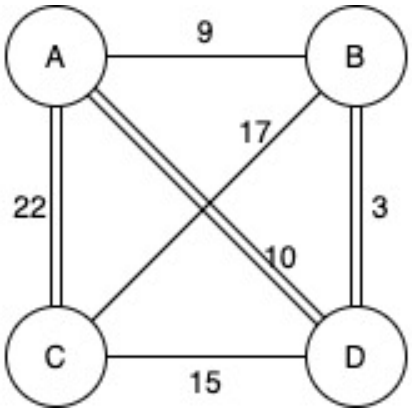

Fig. 2

Our given Route using the Nearest Neighbor algorithm follows the double lines starting from Vertex A in Fig. 1, and starting from Vertex B in Fig. 2. Note that the total Route weight starting from Vertex A is 27, while the total weight when starting from Vertex B equates to 35.

For this reason, a driver function was created with the purpose of find the lowest possible Route weight attributable to a given graph; the driver function iterated through each vertex, calling Nearest Neighbor with each new vertex set as the start. Each new minimum-total-weight Route was logged, and the function returned the lowest-weighted Route possible from the set of starting vertices.

2-Opt Swap takes an existing Route (the best Route yielded by Nearest Neighbor) and creates a series of slightly different routes, overwriting the previous best in the event a new best is discovered. There are several existing versions of 2-Opt. A common one involves examining Route edges that cross over each other, then rearranging the vertices so that they no longer cross, as demonstrated in Fig. 3:

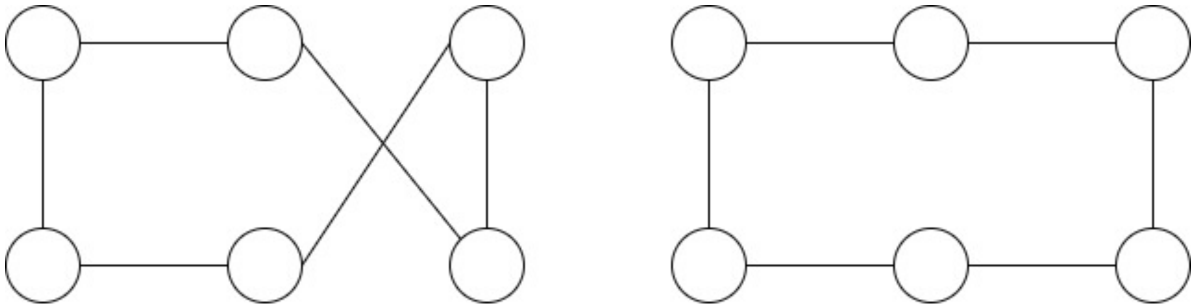
Fig. 3

Another version of this algorithm seeks to do precisely the opposite; it deletes a pair of nonadjacent edges, then reconnects the vertices using different ones[3] (Fig 4):

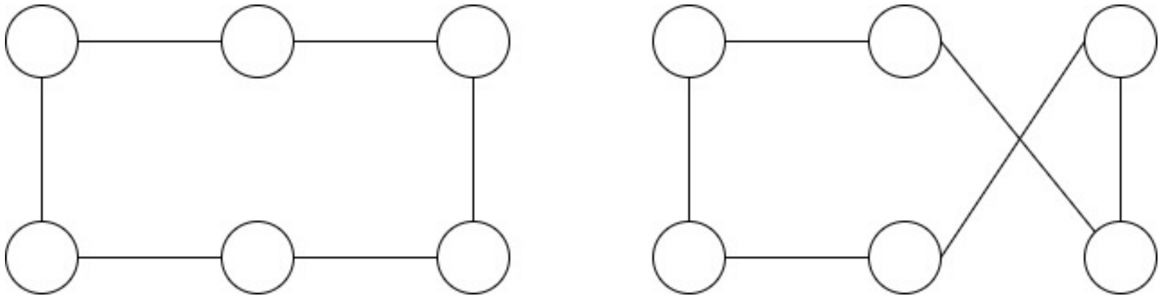
Fig. 4

For the purposes of this study, an exhaustive swap apparatus was implemented; that is, 2-Opt Swap examined every possible combination of swaps and performed one if it was valid. If a given swap produced a Route *x* of lower total weight than the previous Route of lowest weight, then Route *x* was logged as the current best. A single swap would be executed as follows:

1) Take the first *i*-1 vertices and add them to the new Route.
2) Take vertices *i* to *j* and add them in reverse order to the new Route.
3) Take all vertices from *j*+1 to the end and add them in their present order to the new Route.

Such a swap would look like this in a linear representation:

currentBest: A -> B -> C -> D -> E -> F -> G -> H -> A
i = 3, *j* = 6
1) A -> B -> C

[3] Anany Levitin. 2012. *Introduction to The Design and Analysis of Algorithms* (3rd ed) Pearson, Upper Saddle River, NJ.

    2) A -> B -> C -> G -> F -> E -> D
    3) A -> B -> C -> G -> F -> E -> D -> H -> A

In order to perform this operation exhaustively (therefore yielding the best possible Route), a driver function was employed that examined all eligible vertices[4] in assorted ranges dictated by a nested for-loop. Pseudocode for this driver function is as follows:

```
n <- number of vertices in the dataset
for i <- 1 to n-2
    for j <- i+1 to n-1
        newRoute <- twoOpt(currentRoute, i, j)
if newWeight < bestWeight
    bestWeight <- newWeight
    currentRoute <- newRoute
```

By traversing all combinations of vertices within the context of this nested for-loop, in conjunction with the associated swapping mechanism, the algorithm was guaranteed to return the best possible Route.

This brings us to the notion of the Random Swap, which, as mentioned above, was implemented as an add-on to 2-Opt. The idea behind the Random Swap is that the results produced by 2-Opt have the possibility of being improved upon by introducing a bit of non-determinism to the mix. By interjecting random valid values for *i* and *j* (see pseudocode above) at irregular intervals, the 2-Opt algorithm escapes the linear fashion of its nested for-loops, performs a swap using the randomly-determined values of *i* and *j*, then continues about its business as though nothing had changed during its prior iteration. In the implementation designed for this study, the probability of a Random Swap could be 'tuned' for the purpose of studying results at different frequencies (more on this will be discussed in the following section).

Complete pseudocode for the Random Swap, as added into the framework of 2-Opt, follows below (pseudocode showing the evaluation of potential new lowest-weight Routes is redundant and omitted for brevity):

```
n <- number of vertices in the dataset
tune <- an arbitrary integer
for i <- 1 to n-2
    for j <- i+1 to n-1
        rand <- a random integer between 1 and tune
        if rand % tune = 0
            tempI <- random integer between 1 and n-2
            tempJ <- random integer between i+1 and n-1
```

[4] The first and last vertices must remain the same in order to preserve the Hamiltonian Cycle, thus they are considered ineligible.

```
        newRoute <- twoOpt(currentRoute, tempI, tempJ)
    else
        newRoute <- twoOpt(currentRoute, i, j)
```

As exhibited above, a random integer is selected within the range of our *tune* variable. For instance, if *tune* is set to 10, then a random integer between 1 and 10 is assigned to the variable *rand*. By performing our Random Swap only in the event *rand* mod *tune* is equal to 0, we have effectively introduced a 1 in 10 chance that such a swap will occur (test results showed actual occurrences closer to 5% of the time when *tune* was set to 10). Because the new random values for *i* and *j* were represented by 'temp' variables, the loop continues after the random swap as though *i* and *j* were never modified.

The Data

With each of these three algorithms, it seemed obvious to check for consistencies between datasets of similar size, and/or datasets that built upon themselves in linear fashion (as mentioned under Testing Procedures, above). Any hopes at finding such a correlation were quickly thrown out the window during final analysis of the test results. While Nearest Neighbor showed a small amount of clustering near 16%-above-optimal, complete results for this algorithm ranged from 12 - 18%. The overall average from the 25 datasets was 15.88%, making the small clustering at 16% seem unremarkable. Further analysis of the datasets sized at and around the smallest (Berlin52) confirmed this. Berlin52 did exhibit the best results of those datasets officially included in this study, but datasets of 51 and 38 cities, not included in official results, exhibited percentages-above-optimal of 13.15% and 1.65% respectively. (Since Dj38 and Eil51 did not include mathematically proven lower bounds, I could not in good faith include 1.65% as an official best, but the results are telling nonetheless.) Additionally, the next smallest dataset in the official test set (70 cities in size) registered a whopping 17.93% above optimal, blowing any notion of correlations amongst similar data sizes out of the water. The KroA series of 100, 150 and 200 cities, each subsequent set building upon the last, came in at 16.05%, 18.68% and 17.62% above optimal; KroB100, 150 and 200 registered at 16.91%, 20.98%, and 20.22%. Overall, these do seem to exhibit results in the same general range, with increases between each set of 100 and 150. Conclusions that results get worse with data size, however, would be misguided, as we then see a slight dip in our percentage (i.e. better results) at the next tier of 200. Thus, expectations that results would continue to degrade with data size / difficulty proved inaccurate.

Figures 5 and 6 below show the results of Nearest Neighbor on all 25 datasets. Fig. 5 is represented as a 1D graph for the purpose of showing the overall spread, while Figure 6 is represented in 2D with the intention of showing test results by data size.

Once again, were we to see any correlation between data size and accuracy, we would expect to see an upward slope (curved or linear) with flat areas when

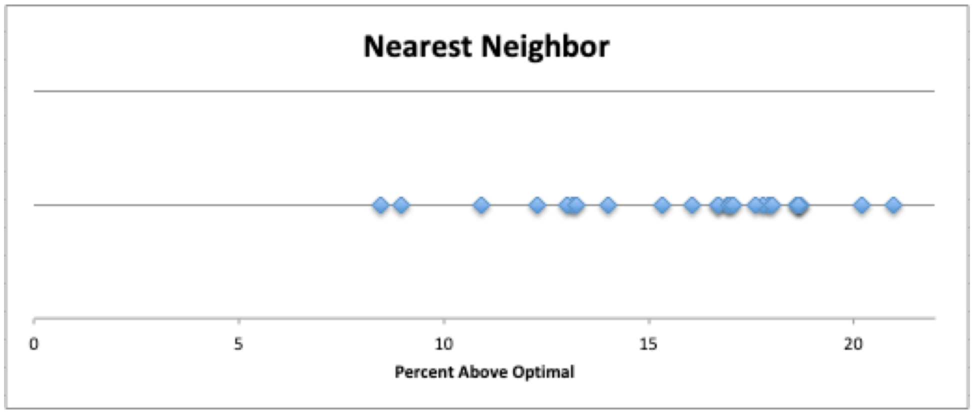

Fig. 5

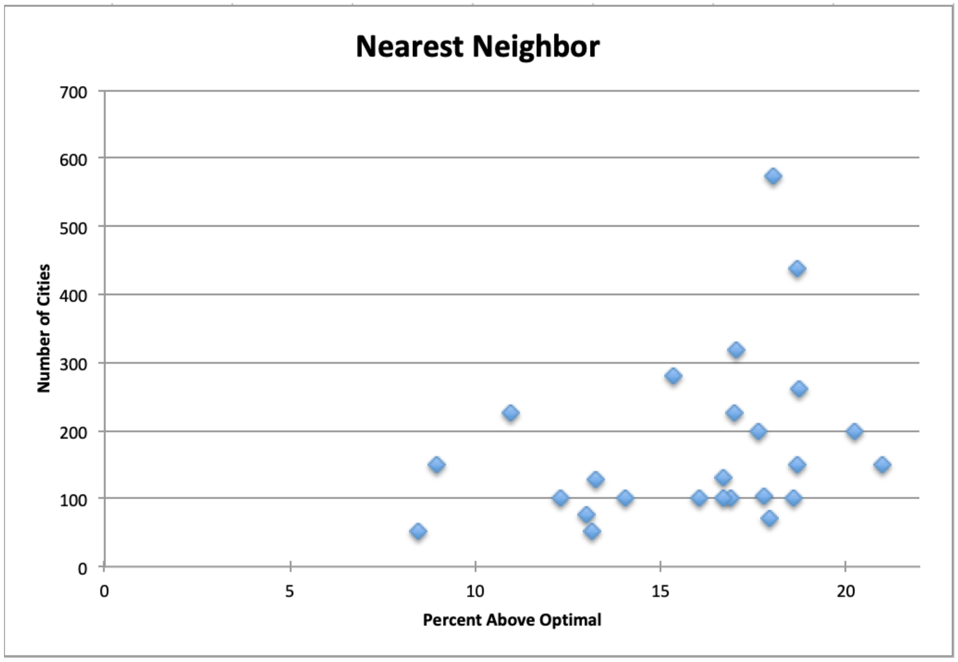

Fig. 6

testing similar sized datasets. We clearly do not see this amongst the distributed results reflected in Fig. 6.

Overall, results for Nearest Neighbor showed most datasets testing in the 13 - 19% above optimal range with a few outliers at or above 20%. The official best, as previously mentioned, was 8.47%, while the official worst was 20.98%. While this appears to be a wide spread, it is actually somewhat remarkable just how tight of a range we get from a simple greedy algorithm.

2-Opt Swap also saw a broad range of results. Notably, one test (Berlin52) actually hit the proven-optimal lower bound. Also of note was the fact that Berlin52 was the dataset that scored the best accuracy when run through Nearest Neighbor. Additionally, while the dataset that scored the worst on Nearest Neighbor did not have the worst score after being run through 2-Opt, it did come in at second worst.

This hints at the suggestion that there is a correspondence between those datasets which perform best in Nearest Neighbor and those which see the best results from 2-Opt Swap, and vice versa. Again, however, this would be a naive generalization of the overall test results, as dataset KroD100 (100 cities) saw only 2.19% improvement after testing at only a little below median with Nearest Neighbor. Furthermore, our grouping of 100-city datasets were grouped in substantially tighter fashion as the result of Nearest Neighbor than with 2-Opt, from which datasets in this group tested only one set off the high and low overall bounds.

Figures 7 and 8 reflect the distribution of results 2-Opt Swap:

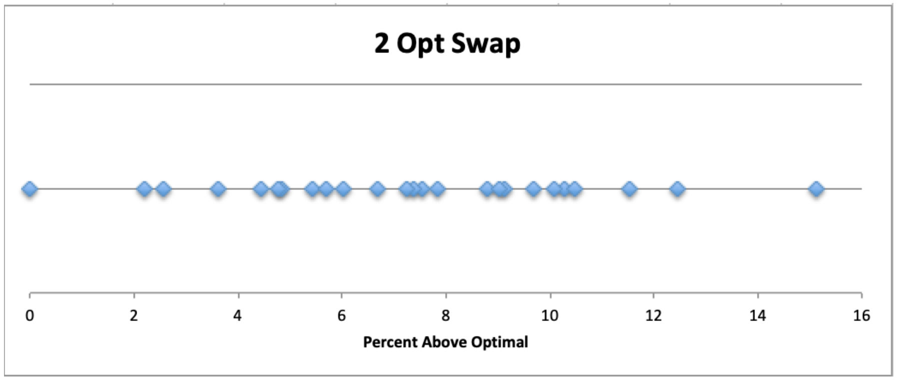

Fig. 7

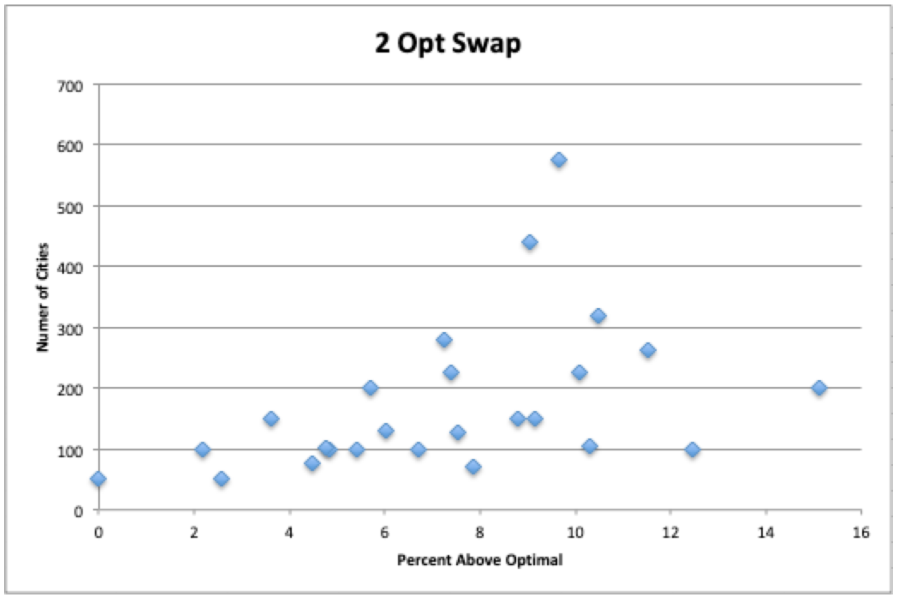

Fig. 8

As we can see, there is a wide spread in results, with no real clustering at any particular percentage above optimal. The best result was obtained on Berlin52, at precisely optimal, while the worst was KroB200 (200 cities) at 15.33% above optimal. Average amongst all 25 datasets was 7.32% above optimal. Figure 8 clearly shows no relationship between datasets of like size.

Improvements in results from one algorithm to the next spread from 3.54% at worst to 13.86% at best, with an average overall improvement of 8.14%.

The purely scientific nature of this testing regimen took a bit of a divergence at Random Swap due to the fact that results could change drastically from iteration to iteration, with no known best-possible result other than our proven lower bound. (This bound was never met at this stage, although one dataset came very, very close). Sometimes a series of random swaps would return an overall Route weight substantially worse than previously arrived at via 2-Opt Swap, and sometimes the results would be much better, all while testing the same dataset. In small to midsized datasets, one could easily see a difference between different values of the *tune* variable mentioned above. "Tuning" the function to perform once in ten iterations vs. once in twenty, for instance, made it possible to achieve generally better results with some datasets, and vice versa on others. Due to the nondeterministic nature of this algorithm, one could conceivably run it ad nauseam in an attempt to achieve better and better results. For the purposes of this study, datasets at the small to medium end of the size spectrum were run through 100 complete iterations of 2-Opt Swap (which in and of itself iterates through (*n* - 2) * (*n* - 1) possible Routes) with Random Swap enabled. This was performed four times per set, tuned to 5, 10, 15 and 20.[5] Datasets of 200 cities and above saw fewer than 100 attempts due to the shear amount of time required to perform each complete iteration of 2-Opt; those with 200 cities were given 50 trials, those with 225 were given 25, and so on based on execution time. In no circumstance were less than 1M Routes taken into consideration, and in most cases the number of Routes superseded 8M. The lion's share of datasets (17 of 25) benefited most from Random Swap tuned to 10 -- i.e. a random swap would occur when *rand* mod 10 was equal to 0 -- which in practice actually happened about 5% of the time. This was bookended tuning-values of 5 and 15, at 3 datasets apiece, while only one showed the best results at 20.

As with Nearest Neighbor and 2-Opt Swap (as run without Random Swaps enabled), this algorithm produced results that were all over the map. Notably, however, all datasets tested within an upper bound of 10% above optimal, with all but three testing within 8%. The best result weighed in at .23% above optimal (excluding the optimal result previously achieved by Berlin52 with 2-Opt); of interest is that this was achieved on Eil51 (51 cities), which leads one to wonder about the relationship between problem size and solvability. The worst result came from KroD100 at 9.49%, with the average percentage above optimal for all 24 datasets (again excluding Berlin 52, in which no improvement beyond optimal was possible) at 5.15%.

Figures 9 and 10 below show the resulting spread.

[5] Doing the math, given the smallest dataset of 52 cities, we get 50 x 51 x 100 x 4 possible Routes, for a total of 1,020,000 total Route weights taken into consideration. Doing the same math for datasets of 150 cities yielded over 8.8M possible Routes considered.

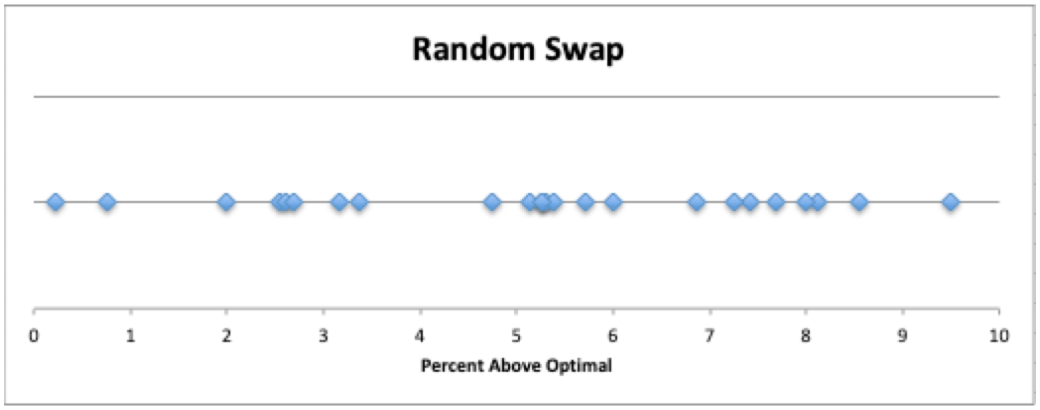

Fig. 9

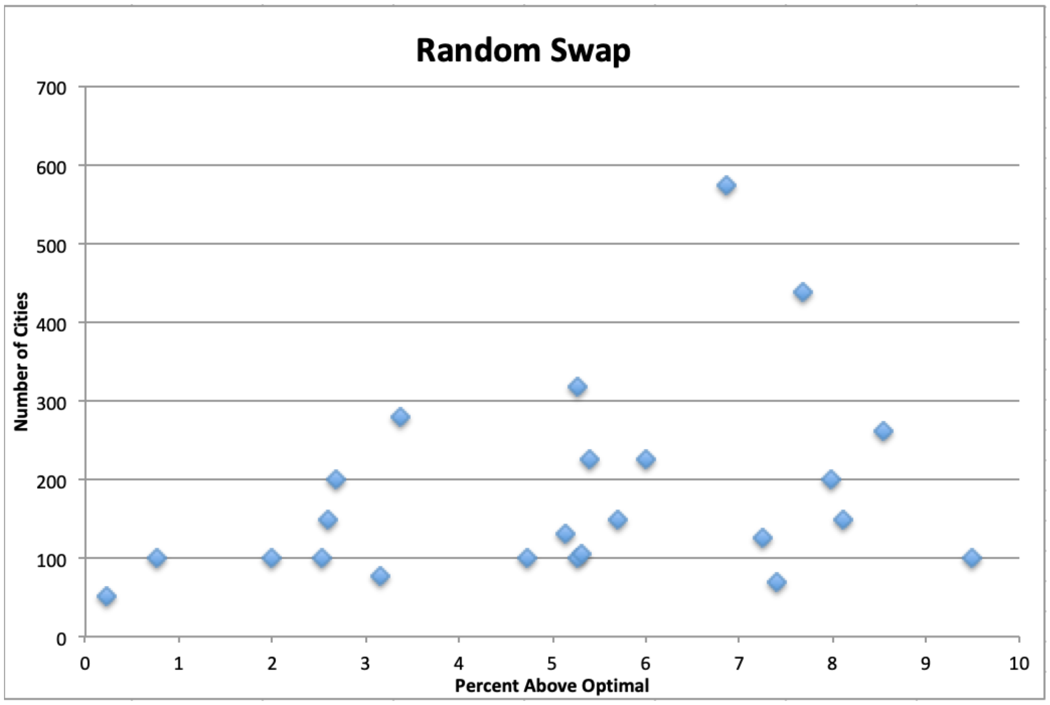

Fig. 10

Improvement between 2-Opt Swap with and without Random Swaps was mainly in the 0 - 3% arena, becoming sparser after 3%. The worst improvement between the two tested at .29%, and the best improvement was 7.14%. On average, Routes saw a 2.48% improvement from one algorithm to the next.

<u>Analysis and Conclusions</u>

The true story told by the test results in the previous section is detailed in Figure 11 below. The leftmost column of each of the three blocks shows the percent above optimal yielded by Nearest Neighbor, the middle column of each block reflects 2-Opt Swap, and finally 2-Opt Swap with Random Swap enabled is shown on the right hand side. Average, best and worst cases from each algorithm are presented as blocks for easy analysis.

On average, Nearest Neighbor proved a decent baseline given the fact that the algorithm could perform no operation "smarter" than greedily choosing the edge of lowest weight. The results were quite spread out in the best and worst cases, however, making it far too unpredictable for real-world deployment. (Of course, Nearest Neighbor was never intended as a "real world" solution within the context of this study.)

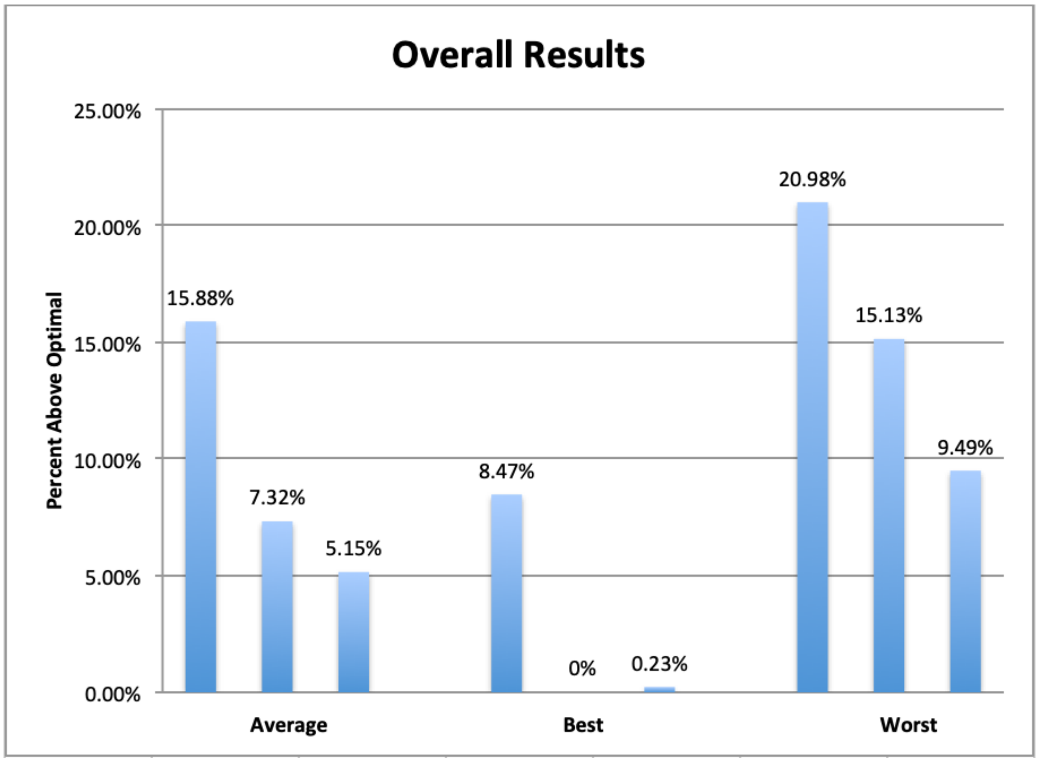

Fig. 11

Looking next to our 2-Opt Swap algorithm, we see a marked improvement across the board. While results saw a wide distribution, they were, on average, 8.56% closer to optimal than Nearest Neighbor had provided as input. In the best case, 2-Opt actually hit the optimal value, while in the worst case, tests still registered nearly 6% improvement.

If 2-Opt Swap took a sledgehammer to Nearest Neighbor, then Random Swap took a scalpel to 2 Opt. Results showed an average of just 2.48% improvement from 2-Opt to Random Swap. The best ending results came from datasets that had already exhibited strong results with 2-Opt and Nearest Neighbor.[6] The worst

[6] While there was no direct correlation between which datasets performed better than others when presented in the context of each algorithm, it is reasonable to conclude that a better starting route provided by Nearest Neighbor would yield a better overall final result after further processing by the subsequent algorithms. For

results also saw greater than 5% improvement from 2-Opt's worst, however. Thus, given the ease with which Random Swap can be implemented, there is little reason not to do so if implementing 2-Opt Swap as well. (To be sure, an average improvement of 2% over a hypothetical prior best of 5% above optimal yields a 40% improvement -- no small number.)

While average and best cases produced the most dramatic results (huge improvements between Nearest Neighbor and 2-Opt in the average case, an optimal result hit in the best), it is the worst case that requires special mention. As mentioned above, 2-Opt provided over 5% improvement over Nearest Neighbor, with another 5% drop when Random Swaps were enabled. This took a pretty terrible looking input (relatively speaking) and brought it to within 10% of optimal. Considering the average case was over 5% above optimal, the 4.34% difference in end results between the two seems scant.

It would be of interest to perform further tests on smaller datasets (fewer than 70 cities) with proven lower bounds. The lack of such proven bounds on several of the small datasets tested created constraints as to what could sensibly be considered scientific; as mentioned in previous sections, several such datasets were excluded for this reason. However, several details remain of note, even if they are not considered part of the official record. First, all four datasets of size 38 to 70 resulted in Routes within 2% of optimal weight. Of these, one actually hit optimal, while another was within one-quarter percent (Eil51 resulted in a total Route weight of 427, with an optimal lower bound of 426). Unfortunately, due to the unproven nature of several of the datasets in this size range, any correlation between data size and accuracy remain inconclusive.

Feasibility of real-world deployment of the algorithms implemented herein depends solely upon the requirements of the end user. Large organizations such as USPS and FedEx may require faster turn times from their software than, say, an astronomical study, for which accuracy may trump timeliness. While Nearest Neighbor is clearly best suited as a starting point, it would be interesting to survey differences in average and worst cases between the Swap algorithms and more complex algorithms such as Christophides and the Lin-Kernighan heuristic. For those who merely require accuracy to within ten percent of optimal in the worst cases, are not using datasets of over 1000 vertices, or are unconcerned with the time required to produce results on such complex datasets, 2-Opt Swap as implemented with Random Swap may provide a simple and effective solution.

---

instance, given the best case performance of Nearest Neighbor at 8.47% above optimal, it can be expected that a near-average improvement yielded by 2-Opt will produce a closer-to-optimal result than we would have seen in a worst case of 20.98%. There are, as exhibited by the test results, exceptions to everything, but this does help explain the right-leaning triangle shape one can imagine when viewing each 2D scatterplot in the previous section.