TriVecta: A Hybrid Approach to Solving the Traveling Salesman Problem

Anastasia Spencer
The University of Alabama, akspencer1@crimson.ua.edu

Abstract - **The following paper compares three different approaches to "solving" the TSP. These algorithms are coded in the C++ language and takes input from a triangular graph read into a 2D matrix. The three algorithms to be discussed are the Brute Force, Nearest Neighbor and an original algorithm. The original algorithm, coined the "TriVecta", utilizes the basis of the Nearest Neighbor approach. Phase one of the algorithm performs the nearest neighbor algorithm on all cities as a starting point to generate candidate routes. During phase two, the candidate routes each go through the process of Simulated Annealing to discover the best of the candidate routes. Phase three performs a two-opt swap algorithm in order to fully optimize the final solution.**

Brute Force Approach

The Brute Force approach evaluates all permutations of paths within a graph to determine the best tour of cities with the shortest distance. While this intuitive approach may seem like it solves the TSP, the Brute Force approach fails with larger graphs as the computing time increases astronomically. The running time for this approach is $O(n!)$

The Brute Force method contains two primary functions. The first function, bruteForceTSP, works by checking each possible path created by the built in next_permutation function which rearranges the elements of the given vector into the next lexicographically greater permutation. The algorithm calculates the distance of the current path utilizing the totalDistance() function and then compares the new distance to the current best distance.

Figure 1.1 analyzes the running time for the program as the number of cities increase. The X-axis is the seconds taken and the Y-axis is the number of cities. With smaller city numbers the running time is almost instantaneous. However, the running time quickly increases seeing the first long stretch of time at 13 cities and an even longer stretch of time when the algorithm is given 14 cities, that seems to be the threshold for the Brute Force Algorithm capacity. In conclusion, the Brute Force approach is a guaranteed correct answer but only performs well on very small data sets.

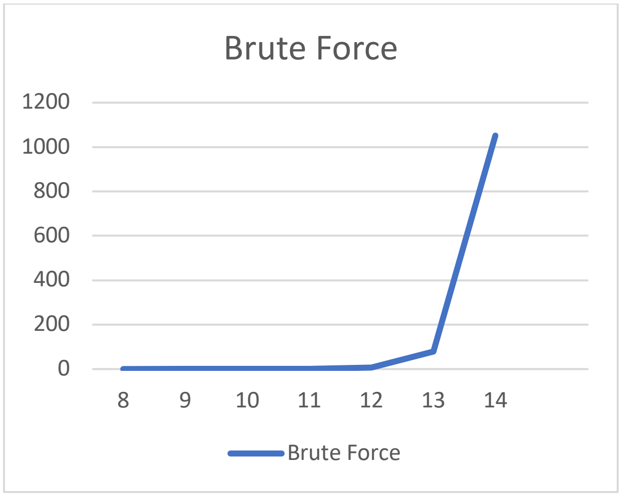


Figure 1.1

Nearest Neighbor Approach

The Nearest Neighbor approach analyzes every unvisited city and traverses to the next closest city. While this algorithm is much faster than Brute Force, running at $O(n^2)$, this approach is not without its downfalls. This approach is overall not an optimal algorithm because it rarely gives the best route for a given graph.

This program utilizes the nearestNeighbor() function which works while the size of the tour is less than the number of cities and iteratively calls the findNeighbor() function. The findNeighbor() function works by checking each unvisited city and determining if it will be the next best city in the path. Once that city is found, it adds it to the tour and marks that it has been visited.

Compared to the Brute Force approach this algorithm is much more successful on large data sets. Figure 2.1 illustrates the running time of the Nearest Neighbor algorithm compared to the Brute Force algorithm on the same number of cities. The X-axis is the seconds taken and the Y-axis is the number of cities. Clearly, the Nearest Neighbor works considerably faster but at the cost of generating a suboptimal solution.

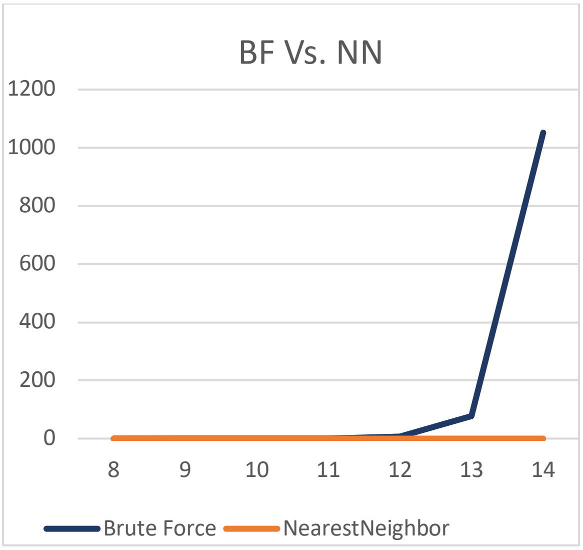


Figure 2.1

TriVecta Algorithm

The following algorithm consists of three phases beginning with an optimized version of the nearest neighbor. Next, it performs Simulated Annealing and then finally refines the solution via two-opt swap.

Each of the following phases will be tested on the same sample 100 size graph to demonstrate the continued optimization as each phase of the algorithm is added.

In the original versions of this algorithm, two-opt swap was initiated prior to Simulated Annealing, however, the sequence of these optimization techniques is imperative to discovering an optimal solution. Performing two-opt swap initially does not use the strengths of each algorithm appropriately.

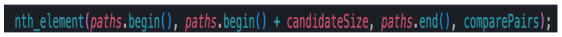
Performing two-opt swap first resulted in the swaps not generating much improvement to the path since two-opt swap primarily considers local changes. This reduced the effectiveness of Simulated Annealing since the solution was already optimized locally. However, when performing Simulated Annealing first, the program can explore a broader solution space before optimizing it locally via two-opt swap. Thus, the reasoning behind this algorithm performing Simulated Annealing and then two-opt swap.

I. Phase 1

A problem with the nearest neighbor approach previously discussed is that it only tests one starting point. It is highly unlikely that the first city in a given graph is the starting point for the best path. First, instead of beginning the tour at the same city given by the input matrix, this approach runs the nearest neighbor algorithm testing each city as a starting point. Once the lengths of each of the paths are generated, they are stored in a vector of pairs so they can be compared in phase 2 of the algorithm. The downfall of this is it adds to the running time by a factor of N. It also does not create a better solution by much. This is the reasoning behind the additions of phase 2 and phase 3. Running this algorithm on a size 100 graph generates the solution length: 3179955.

II. Phase 2

Phase 2 Consists of finding the top best starting cities using the approach discussed in Phase 1. The algorithm stores each generated path and ,via the selectCandiadates() function, selects a given number of candidates that have the shortest paths. Via testing on different size input graphs, it was found that the best number of candidates given is decided by taking 10% of the total number of cities. Since larger input graphs create larger numbers of candidates determining the shortest path from all the given paths can significantly add to the running time of the algorithm. Because of this, within the selectCandidates() function, rather than sorting the entire list of paths to determine the nth smallest paths, this function utilizes the C++ nth_element function (Figure 2.1) to sort just the smallest n elements to avoid the computational and running cost of sorting the entire list of paths. These candidate paths will then be optimized using Simulated Annealing.



```
nth_element(paths.begin(), paths.begin() + candidateSize, paths.end(), comparePairs);
```

Figure 2.1

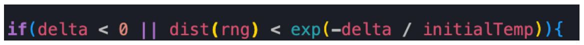
This optimization technique is based on the heating and controlled cooling of metal within the annealing process in metallurgy. The simulatedAnnealing() function will be performed on each candidate path for a given number of iterations. To increase the solution quality, the best number of iterations is 100 times the size of the input graph. Via testing on different size inputs, this seems to produce the best number of iterations to allow the algorithm to explore the space thoroughly. Next, the function chooses two cities randomly, ensuring the cities are within the range of the cities given and ensuring the same city is not chosen for both random cities. It then swaps those edges and returns the new tour along with the new length. Next, it calculates the change in cost of the new tour compared to the previous tour and stores that calculation in the variable named delta. The next step in the algorithm decides if the new path should be accepted or not based on two measurements. The first is simple: if the path is shorter than the previous path then accept the new path. The second is a little more complex. It is based on a concept called the Metropolis Criterion. This Criterion is the scaffolding for Simulated Annealing as it allows for the algorithm to accept worse solutions in some cases. The value in accepting worse solutions is that it may help to broaden the search in the solution space as it is able to explore paths that may seem worse at first but later improve. The probability in which the algorithm will accept a worse solution is calculated by the equation:

$$P = \frac{-\Delta}{T}$$

Where Δ is the difference between the newly generated path and the original path divided by the current temperature (discussed next) all raised by the exponential function. The exponential function is used to ensure a positive number result since the result of an exponential function is always positive for all real numbers:

$$\{\frac{-\Delta}{T} | \Delta \in \mathbb{R},\ T \in \mathbb{R}\}$$

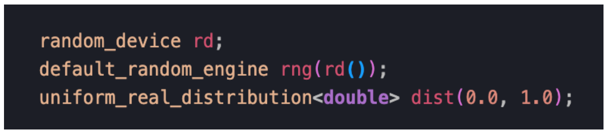
The temperature aspect of Simulated Annealing is an integral part of the process. It is often seen in decision theory and reinforced learning and achieves the transition from exploration to exploitation. At the beginning stages of the program, the initial temperature is high reflecting a high probability that a worse solution will be accepted which aids in the exploration of the space. As the program moves through its iterations, the temperature cools down and the probability that a worse solution will be accepted decreases to allow for an increase in exploitation and further refining within the path. Also found through testing on various size graphs, an initial temperature of 100 times the graph size seems to generate the best solutions. The Metropolis Criterion is utilized in the program within this if statement illustrated in the following figure:



```
if(delta < 0 || dist(rng) < exp(-delta / initialTemp)){
```

Figure 2.1

The last component of this statement to discuss is the dist(rng) that is being used to compare the result from the Metropolis Criterion. At the beginning of the simulatedAnnealing() function, a random number engine is defined. (Figure 2.2) This random number engine is used to generate a random number between 0 and 1 under a uniform distribution function.



```
random_device rd;
default_random_engine rng(rd());
uniform_real_distribution<double> dist(0.0, 1.0);
```

Figure 2.2

The reason the random number must be under a uniform distribution is to guarantee the same probability of each randomly generated number between 0 and 1. This will avoid skewed probability values. Figure 2.3 illustrates the probability density function of the uniform distribution this is referring to.

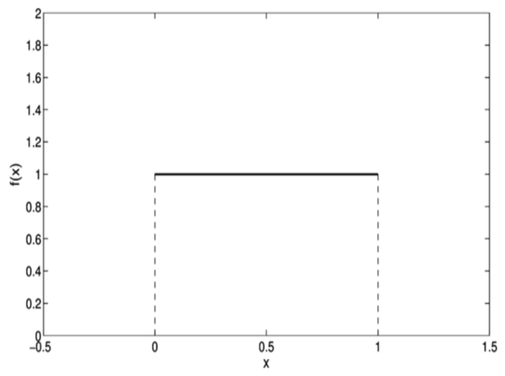


Figure 2.3

Source: Adapted from [1]

Once the decision has been made to accept or deny a swap in city order, the temperature is cooled down by a factor of the cooling rate. A high initial temperature coupled with a low cooling rate corresponds to frequent exploration which, because of the goals of this algorithm, is optimal since after Simulated Annealing, the winning path from the candidate paths will endure two-opt swap.

In conclusion, through an expedited sorting process, candidate values are chosen to undergo Simulated Annealing. Then, the Simulated Annealing function generates random cities to swap. It then uses the Metropolis Criterion to decide when to choose a worse solution. At the beginning of the iterations, it explores the search space frequently because of its high temperature and as the temperature cools, it exploits more frequently. At the end of each iteration, it cools the temperature by a set cooling rate. With this addition to the algorithm, the sample size 100 graph generates a solution of: 1220976 which is a noticeable improvement from the previous 317995.

III. Phase 3

After the Simulated Annealing function takes place, it returns the best generated route to undergo the final phase: two-opt swap. This function works by iterating through all pairs of edges in the tour. It then reverses the tour between the given edges. The reversal is implemented using the built-in reverse function that acts upon the given vector. The function next determines if the given tour yields any improvement. Once each edge has been considered, an optimal path is produced. With this final addition to the algorithm, the solution is: 1052130

Conclusion

The Brute Force algorithm is not optimal for large data sets but generates the correct solution. The Nearest Neighbor algorithm works on large data sets but rarely generates an optimal solution. The TriVecta algorithm combines three phases to create an optimized solution. First, it performs the nearest neighbor algorithm on each city to evaluate the best starting points. Those paths are then used as candidate paths and undergo Simulated Annealing. The best path generated after Simulated Annealing is chosen to perform two-opt swap, resulting in an optimized solution to the Traveling Salesman Problem running in O(n^3) time. Figure 3.1 illustrates the running time of this algorithm on various size graphs. The X-axis is the seconds taken and the Y-axis is the number of cities. A drawback of the TriVecta algorithm is the chance it takes in potentially throwing away an optimal path and accepting a worse path. This can occur during the simulated annealing phase if the Metropolis Criteron is met. However, it is a risk that must be taken in pursuit of exploration. In comparison to the Nearest Neighbor Algorithm, the TriVecta generates a much better solution at the cost of a longer running time.

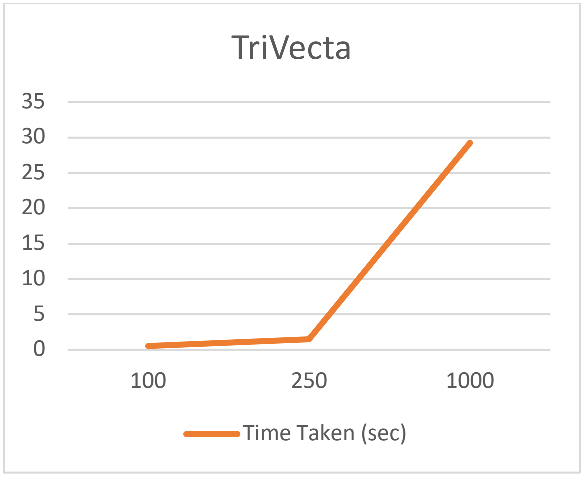


Figure 3.1

References

[1] *ResearchGate,* [Online Image]. Available: https://www.researchgate.net/figure/1-shows-an-example-of-a-uniform-distribution-on-the-interval-0-1-A-factor_fig2_40868486. [Accessed: 20-02-2024].

[2] Alyssa Walker, "Traveling Salesman Problem: Python, C++ Algorithm," https://www.guru99.com/travelling-salesman-problem.html, [Online]. Available: URL [Accessed: 20-02-2024].

[3] "C++ Program to Implement Traveling Salesman Problem using Nearest Neighbor Algorithm," https://www.tutorialspoint.com/cplusplus-program-to-implement-traveling-salesman-problem-using-nearest-neighbour-algorithm, [Online]. Available: URL [Accessed: 20-02-2024].

[4] Ali Hamdar, "Simualted Annealing – Solving the Travelling Salesman Problem (TSP)," https://www.codeproject.com/Articles/26758/Simulated-Annealing-Solving-the-Travelling-Salesma, [Online]. Available: URL [Accessed: 20-02-2024].

[5] Andy, "C++ Implementation of 2-opt to the "Att48" Travelling Salesman Problem," https://www.technical-recipes.com/2012/applying-c-implementations-of-2-opt-to-travelling-salesman-problems/. [Online]. Available: URL [Accessed: 20-02-2024].