A Brute Force, Nearest Neighbor, and Original Solution to the Traveling Salesman Problem

Riley A. Durbin

The University of Alabama

CS 570: Computer Algorithms

A Brute Force, Nearest Neighbor, and Original Solution to the Traveling Salesman Problem

This paper will discuss the three algorithms written for the Traveling Salesman Problem (TSP). Each algorithm will be discussed thoroughly. All of these algorithms were originally written in Python, but they were rewritten in C++ to fix the efficiency issues. The Brute Force algorithm simply calculates the distance for each tour possible using permutations and arrays. This algorithm is extremely time consuming and only realistic for very small graphs. The Nearest Neighbor algorithm is a popular heuristic algorithm that finds the smallest edge from each current vertex and adds that edge to the graph. This is a fast algorithm that will return a "pretty good" solution, but this tour will almost never be the most optimal tour. The original solution written is split into five steps. Each of these steps will be discussed in this paper. In short, the algorithm runs the nearest neighbor algorithm and calculates alternate tours by branching at certain points in the standard nearest neighbor tour. Once it has found the most optimal solution of these alternate tours, it runs two small brute force optimizations and then a simulated annealing algorithm to try to find a more optimal solution while removing some of the locality bias that the nearest neighbor algorithm includes. This section can be optionally looped. The motivations and details will be thoroughly discussed below along with the results of this algorithm in the competition.

A note about the compilation and running of these algorithms: I used the -O3 compilation flag when compiling each program, like "TSP_nearestneighbor.cpp -O3". This compiler flag turns on optimization options that speed up the running of the code. When compiling these programs, it is important to use the -O3 flag, or the runtime of the program will not match the times listed below. This flag alone significantly decreased the amount of time spent running each program, which was ideal for the competition.

Brute Force

The Brute Force algorithm written to solve the TSP guarantees that it will find the most optimal tour for a graph, but this approach is only realistic for very small graphs. My brute force algorithm can run any graph of size 12 or less in less than a minute. As the number of vertices increase linearly, the time it takes to brute force every possible tour increases exponentially. For example, it took my program 0.45 seconds to run the brute force algorithm for a graph of size 10, 4.87 seconds for a graph of size 11, 59.11 seconds for a graph of size 12, and 11 minutes and 55.04 seconds for a graph of size 13. I did not test it, but it can be estimated that a graph of size 14 would take over 2 and a half hours, and a graph of size 15 would take almost two days. The runtime of this algorithm is **O(N!)**, where N is the number of vertices in the graph. Although this algorithm finds the optimal tour, this runtime is extremely inefficient for most use cases.

Algorithm Breakdown

The program begins by reading the data from the graph file into a two-dimensional vector. This data structure was used because I am familiar and comfortable with it, and I thought the added functionality could be more useful than a standard two-dimensional array. The push_back and indexing operators are constant time functions, but the erase and insert vector functions used may take longer. This could increase the runtime of the functions, but I did not consider this when calculating the runtimes. The brute force algorithm assumes that the starting and ending point of the tour will be the 0^{th} vertex, but this graph could be rotated to start and end at any point and would still be the same circular tour. As a result of this assumption, the total number of potential tours or permutations that need to be tested is the factorial of the number of vertices minus 1. For 10 vertices, this is equal to 362,880 tours, but for 15 vertices, this is equal to 87,178,291,200 tours. It is clear why the runtime increases as quickly as it does. Once it calculates the number of tours to test, the program runs a basic for loop that calls the C++ “next_permutation” function and the custom bruteForce function to calculate

the distance of each permutation of the tour. This bruteForce function is slightly optimized, as it stops calculating the distance of the current tour if it exceeds the minimum distance found at this point in the loop. This slightly speeds up the process, as the program does not need to spend anymore unnecessary time once it knows that the current permutation is not the optimal tour. The program prints out the distance and tour whenever it finds a new optimal tour, and it also prints checks at every ten-millionth iteration. Finally, the program prints out the optimal tour and the time taken at the end.

Less Than K Approach

Although this program is not recommended for any graphs with more than 13 vertices, it can be useful for checking the optimality of a solution K. If the program was slightly modified so that the min_distance variable held the predetermined K value, the program would search for a tour with a distance less than K. This is valuable for checking how optimal the solution is from a different solution program, like the nearest neighbor algorithm. If it can quickly find a more optimal tour through brute force, it is unlikely that the solution found by the nearest neighbor algorithm is great for the graph.

Nearest Neighbor

The Nearest Neighbor algorithm is a common heuristic approach to finding a good tour for a graph in a short time. This algorithm involves selecting the smallest unvisited edge from each vertex as it builds the tour. Its runtime is **$O(N^2)$**, where N is the number of vertices. This is significantly more efficient than the brute force approach, which is why it is much more common as a solution. While the brute force approach would take multiple days for a graph of size 15, the nearest neighbor algorithm could return a “pretty good” solution for a graph of size 15,000 in less than a minute. Unfortunately, the tours returned are usually not anywhere near the optimal solution. The nearest neighbor solution serves as a great solution for the sake of efficiency, and I used it as a starting point for my original solution.

Algorithm Breakdown

The program begins by reading and storing the edge data in the same way that the brute force algorithm does. After this, it calls the nearestNeighbor function with this edge array. My implementation of the nearest neighbor algorithm makes the assumption that the tour should start and end with the 0^{th} vertex. Usually, a more optimal solution can be found by running this function from every starting point, but this makes the program less efficient, which defeats the purpose of the algorithm as a solution for large graphs. When building the tour, the program keeps track of the unvisited vertices, the total distance of the tour, and the current tour. For each point in the tour, the algorithm checks every unvisited vertex to find the minimum edge distance. Once it finds it, it adds this vertex (nearest neighbor) to the tour, removes it from the unvisited list, adds the distance, and continues the loop. Since it needs to check every unvisited vertex for each vertex, the runtime is **$O(N^2)$**. This algorithm is slightly optimized compared to some other nearest neighbor algorithms, as it only searches through the size of the unvisited list instead of the entire list of vertices. For the first vertex, it searches through N-1 unvisited vertices. For the second vertex, it searches through N-2 unvisited vertices, and so on. As a result, this is slightly faster than some basic nearest neighbor algorithms. When I tested this algorithm with the four graphs from the competition, it performs very quickly, but the results are far from optimal. **Figure 1** shows the time and result of each graph using the nearest neighbor algorithm. Although an every-starting-point nearest neighbor approach would likely return better results, that would take **$O(N^3)$** time. I avoided this to keep it to a runtime of **$O(N^2)$**.

Graph Size	Time	Tour Length
250	0.009215s	4,609
1,000	0.128493s	5,589
5,000	4.443482s	1,004,309
15,000	50.943934s	796,026

Figure 1: The graph of times and tour lengths of four different graphs using the nearest neighbor algorithm.

It is clear that the algorithm runs quickly, but the tour lengths are much longer than most original algorithms would provide. This becomes clearer as we compare it to the results of my original algorithm.

Original Solution

When coming up with my original solution for the TSP, my primary goal was to write an algorithm that was slower than nearest neighbor but returned a much more optimal solution. I think that I succeeded. Initially, I wrote each program in Python because I have been working with Python in two of my other classes this semester. Unfortunately, Python is significantly slower than C++, so it was taking a long time for my programs to find an optimal tour. As a result, I rewrote my program in C++ and used the -O3 compilation flag to speed up the execution. This allowed me to write some functions that would normally take much longer because they were optimized and sped up during compilation. During my initial brainstorming, I had ideas to split the coordinates into quadrants and try to map them out to order them from there, but this was not realistic because the graphs contained edge weights, not (x,y) coordinates. My next approach was the one that I ended up using for the start of my algorithm. I recognized that many nearest neighbor tours resulted in suboptimal tours because the best path involved choosing some paths that were slightly longer than the nearest vertex. I will be referring to this idea of choosing the nearest path as locality bias. My goal was to remove some of this locality bias to find a better solution. My algorithm is split into five main steps, and it involves some customizable settings depending on what graph is being run. It was designed intentionally to be customizable so that I could make quick changes during the competition depending on the size of the graph. All of these functions and settings will be described in detail below.

Every-Starting-Point Nearest Neighbor

When the algorithm begins, it reads in the edge graph in the same way that the previous two programs do. It then calls the whichStartingPoint function, which runs nearest neighbor once for every vertex to find the starting point that results in the minimum distance nearest neighbor tour. The goal of this is to provide the rest of the algorithm with a "pretty good" tour that it can use to edit and optimize. Since the nearest neighbor algorithm runs so quickly for most of the graphs used, this extra optimization for the starting tour is worth the extra runtime. The runtime of this function is **$O(N^3)$**, where N is the number of vertices in the graph. Similarly to the brute force algorithm, this function provides a small optimization in that it stops checking a nearest neighbor tour if the distance of it exceeds the current known minimum tour length. This slightly speeds up the algorithm, so **$O(N^3)$** is the worst-case, where each tour that it checks is smaller than the last. This function is written almost identically to the nearest neighbor algorithm from above, except it loops from 0 to the number of vertices, and it tracks the best minimum distance and the starting vertex that results in that distance. This function returns the vertex number that results in the minimum nearest neighbor tour. Although this seems inefficient, in many cases, it is worth it. For the graph of size 1,000, the normal nearest neighbor algorithm returns a tour with distance 5,589, but the every-starting-point nearest neighbor function returns a tour with distance 4,654. For the graph of size 5,000, the normal nearest neighbor algorithm returns a tour with distance 1,004,309, but the every-starting-point nearest neighbor function returns a tour with distance 648,766. This is a significant improvement, which might justify the extra 81 seconds that the algorithm takes for the 5,000-vertex graph. However, the final optimal tour distance returned at the end of the program is usually comparable when turning this function on and off, so this function is usually not worth the time when analyzing large graphs. Frankly, this function is a waste of time if the 0^{th} vertex results in the best nearest neighbor tour or if the best tour returned is only slightly shorter than the normal nearest neighbor tour. This is clear in the 250-vertex graph, where the 0^{th} vertex results in the best tour. I decided that this risk was worth it for the competition, but I also added the ability to toggle this function

in case I did not want it to run. I will go into more detail about this customization in the **Custom Options** section.

K-Scalar Alternate Tours

The main approach that I came up with to try to remove some of the locality bias of the nearest neighbor algorithm focuses on creating slightly different alternate tours of the nearest neighbor algorithm. Once the program finds the nearest neighbor from a point, it searches back through the list of unvisited edges to find any edge that is less than the distance to the nearest neighbor times a scalar K. For each of these edges, the program saves this possibility as an alternate tour which it will fully explore later. The goal of this is to eliminate some of the locality bias by always selecting the nearest neighbor. Since the more optimal tour often involves selecting a further neighbor from a vertex, this allows the algorithm to test some of these other tours while still following a basic nearest neighbor approach. The pseudocode for this algorithm will show the general premise of the function:

```
int nearestNeighborFindAlts() {
	tour.push_back(startingPoint)
	// normal nearest neighbor approach
	for each vertex besides the starting point:
		currentPoint = tour.back()
		min_distance = infinity
		nearestNeighbor = -1
		for each unvisited vertex:
			if distance from currentPoint to vertex < min_distance:
				update min_distance
				nearestNeighbor = vertex
		// once nearest neighbor is found, save alt tours
		for each unvisited vertex:
			if vertex != nearestNeighbor and distance < min_distance:
```

```
                        save tour as altTour
            add nearestNeighbor to tour
      tour.push_back(startingPoint)
      return distance of tour
}
```

Once all of the alternate tours are stored, the checkAltTours function finishes the nearest neighbor algorithm for each tour. This very frequently results in a better tour length than the original nearest neighbor solution. The scalar that I used most was a K value of 1.5. This would test the alternate tour whenever an edge distance was less than 1.5 times longer than the nearest neighbor's distance. For example, if the distance from vertex A to vertex B is 20 (nearest neighbor), but the distance from vertex A to vertex C is 25, the algorithm would check the tours for both of these paths. This helps to remove some locality bias. This function takes much longer when there are a lot of similar edge weights, as this creates more alternate tours. However, the increase in alternate paths often results in a better solution. Many of the decisions made when designing this algorithm involved sacrificing some time for better solutions. The time complexity of the nearestNeighborFindAlts function is **$O(N^2)$** in the worst-case. The outer loop runs N times, where N is the number of vertices, and there are two inner loops that each run for the amount of unvisited vertices, which is N in the worst-case. This results in a runtime of **$O(N*(N+N))$**, or **$O(2N^2)$**, which simplifies to **$O(N^2)$**. The time complexity of the checkAltTours function is **$O(mN^2)$** in the worst-case, where m is the number of alternate tours. Since each alternate tour usually starts with a partially made tour, finishing the nearest neighbor algorithm will rarely take the full **N^2** time. Because part of the algorithm involves both of these functions, the overall runtime of the K-Scalar Alternate Tours section has an overall time complexity of **$O(mN^2+N^2)$**, which simplifies to **$O(mN^2)$**. The results of the alternate tour function are shown in the table in **Figure 2**.

Graph Size	Any-Starting-Point Nearest Neighbor Result	Alternate Tours Found	Best Alt Tour Result
250	4,609	123	3,319
1,000	4,654	556	4,130
5,000	648,766	2,936	533,907
15,000	796,026	8,318	543,422

Figure 2: The results from exploring the K-Scalar Alternate Tours for four different graphs when K = 1.5.

Longest Edge Brute Force

After the program finds a better tour by checking the alternate tours, it runs a brute force-like function to make small optimizations to the current best tour. It does this by finding the longest edge of the current tour and moving one of the vertices attached to this edge to every other position of the tour, selecting whichever new position will result in the lowest total distance for the tour. The purpose of this function is to slightly decrease the length of the tour, which can help for the competition where small margins can make a tour the most optimal tour. The function has an approximate runtime of $\boldsymbol{\Omega(N^2)}$ in the best-case. It is likely that this will take longer, however. The main loop of the function runs for the number of edges in the tour. The inner loop also runs for the number of edges in the tour, as it needs to find the maximum edge. There is a second inner loop that runs for the number of vertices, as the function needs to find the best place to move the chosen vertex to. If it finds a place to insert the vertex that will decrease the total tour length, it places the vertex there and resets the outer loop to begin looking for the longest edge in the new tour. If it cannot find a place that will decrease the length, it continues the outer loop, looking for the next longest edge. Although this is not a very efficient function, it is very effective in removing large edges and shortening the tour. It runs quickly for most graphs, but larger graphs can struggle with this function. It makes use of an important optimization in the inner loop, as it calculates the hypothetical distance after removing the vertex and hypothetical

distance of inserting the vertex in between two vertices using math instead of a loop to calculate the distance. This saves a large amount of time. It also features a small time optimization where it will skip the process of finding a new position for the vertex if the hypothetical distance after removing the vertex is greater than the current total distance. It also only calls erase and insert when it actually finds a better place to put the vertex in order to decrease the total tour length. Despite the significant time complexity, it is very effective in shortening the tour, which is the goal of a post-algorithm brute force optimization.

Every Vertex Brute Force

The algorithm features a second simpler brute force optimization that runs directly after the Longest Edge Brute Force function. Instead of looking for the longest edge and trying to move a vertex from there, this function simply tries to move every vertex to every position, updating the tour if the hypothetical tour would have a smaller length. The basic structure of the function is very similar to the previous function, but it is more concise. There is only one outer loop and one inner loop for this function, and they each run N times, where N is the number of vertices in the tour. The time complexity of this function is $\mathbf{\Omega(N^2)}$ in the best-case, but it is often worse than that because the outer loop resets whenever a vertex is successfully moved and the tour is optimized. The reason that the outer loop resets is because a vertex being moved means that the tour has changed, so the vertices at the beginning of the tour need to be tested again with the new tour to see if the overall distance can be optimized. This is time-consuming, but it is very effective in decreasing the tour length. For the four graphs previously mentioned, the distances before and after these two functions are listed below in **Figure 3**.

Graph Size	Tour Length Before Brute Force	Tour Length After Brute Force	Time Taken for Brute Force
250	3,319	2,275	0.000449s
1,000	4,130	2,172	0.292172s

5,000	533,907	257,297	24.573849s
15,000	543,422	254,630	23m 42.502267s

Figure 3: The results from running the two Brute Force Optimizations for four different graphs.

It is clear that these post-algorithm brute force optimizations are effective in decreasing the total length of the tour. For small graphs, the time spent running these functions and decreasing the length of the tour is worth it, but for very large graphs like the graph of size 15,000, they are very slow. It is clear at this point that the original algorithm is not nearly efficient enough for large graphs, but it performs very well for small and medium-sized graphs. There are also factors like the relative closeness of the graphs that impact functions like this, but the total number of vertices seems to be the most important factor.

Simulated Annealing

All of the functions up to this point have focused on finding a "pretty good" tour and incrementally improving this tour through small local improvements. The simulated annealing section works to improve the tour by removing the locality bias in the changes that it makes. This implementation is a version of simulated annealing based off of a brief description from Carnegie Mellon and a detailed description from the Second Edition of Numerical Recipes in C. Both of these sources can be found in the **References** section. Simulated annealing is based on the thermodynamics concept where metal is heated to a high temperature and allowed to slowly cool. The simulated annealing algorithm involves selecting a random move for the tour and evaluating if it will improve the length of the tour. If it does, it is accepted. If it does not, there is a chance that it will be accepted even though it is a "bad" move. The algorithm that I wrote involves three main variables that can be adjusted: initialTemperature, coolingRate, and numIterations. For my program, I set these variables to 1,000, 0.9, and 10,000,000, respectively. If I decreased the cooling rate, the algorithm would be less likely to make "bad" moves, but the goal of this section of the program is to intentionally make some "bad" moves so that some of the locality bias can be removed. With the current setup, the function should make a

significant amount of both good and bad moves, because the number of iterations is set to a high value. This makes the process slower, but it should create a better solution. For each iteration of the loop, the program randomly chooses between swapping two elements or reversing a section of the tour. Based on this choice, it mathematically calculates the hypothetical new tour length after making this change. The program then checks if this hypothetical new tour length is shorter than the current tour length. If it is, the swap or reversal is made. If it is not, there is a chance based on how "bad" the change is that the change is made anyway. This chance decreases as the loop continues. This function introduces a random aspect of the algorithm, which has some positive and negative implications. It helps to remove some of the locality bias, which often leads to better solutions. The biggest drawback is that this randomness greatly impacts the expected outcome of the program. In order to find the most optimal solution, this section can be repeated, but the amount of time taken by the program can vary. The pseudocode for this function is outlined below:

```
void simulatedAnnealing() {
      currentTemperature = 1000.0;
      coolingRate = 0.9;
      numIterations = 10000000;
      for the number of iterations:
            bool doSwap = random true or false value
            if (doSwap):
                  select two random distinct indexes
                  calculate the new tour distance if these were swapped
            else:
                  select two random distinct indexes
                  calculate the new distance if this segment was reversed
            if new distance is better or we want a worse solution:
                  make the swap or reversal
```

```
                    update the distance
                currentTemperature *= coolingRate
        longestEdgeBruteForce()
        eachVertexBruteForce()
        return;
}
```

Overall, this is a simple function with the goal of making random changes to the tour, with the probability of making bad changes decreasing as the loop continues. This function is effective in decreasing the tour length, and a loop containing this function proves to be incredibly successful in finding a tour with a near-optimal length for the TSP. The simulated annealing function has a worst-case time complexity of **O(AN)**, where A is the numIterations variable and N is the number of vertices in the graph. The A value comes from the outer loop, and the N comes from the reverse function in the worst-case. This is very unlikely however, as this worst-case complexity assumes that every iteration of the loop involves reversing the entire graph. As a result, the average-case runtime is much closer to **O(A)**, since reversals only have a 50% chance of being attempted, and they are only executed if they would improve the tour or if the algorithm intentionally makes a bad swap. If they are executed, the reversal will contain a random number of elements, so it is unlikely that this reversal will take **O(N)** time. This function calls both brute force functions at the end of the function, which each have a time complexity of $\mathbf{\Omega(N^2)}$ in the best-case. Although this program is not incredibly efficient, it is very effective in finding a short path for a small or medium graph in a reasonable amount of time.

Custom Options

To make this program more flexible, I included some optional parameters that can be changed before the compilation and execution of the file. These custom options include the option to change the K value for the alternate tours, the option to run the nearest neighbor from every starting point, and the option to loop the simulated annealing section until a tour of length X is found. In my testing, I have

found that a K value of 1.5 is optimal for most graph sizes, but the ability to change it to 1.0 to eliminate the alternate tours section is important if the graph is very large with a large amount of edges with similar lengths. This would lead to an overwhelming amount of alternate tours which would slow the program down too much. For most cases, however, 1.5 is a good K value. The option to toggle running the any-starting-point nearest neighbor function is great in that it can speed up sequential executions of the program. Once the program is run once for a graph, the best starting point found in this execution can be manually set for future executions and this function can be toggled off. This improves the runtime significantly. The option to loop the simulated annealing section is the most important part of this program in regard to the competition. Once the program runs once, I am able to input that tour length into the integer X, and the program will loop the simulated annealing section until it finds a tour with length less than X. This X can be set to a target value or the current best-known score. Since the simulated annealing section has a relatively quick runtime, this loop takes only a few seconds for small and medium graphs. This allows the program to continuously use the simulated annealing section to remove locality bias and find better solutions.

Competition Graph Results

Unfortunately, I had a bug in my code on the day of the competition. The simulated annealing loop was running the simulated annealing function with the same tour every time. This allowed it to sometimes get lucky and find a better solution, but it prevented me from finding truly optimal solutions and competing with the lowest scores on the leaderboard. After the competition, I noticed this bug, and I fixed it so that it would update the tour every time that it found a better solution and use this tour for the simulated annealing loop. This means that it applies pseudo-random changes to the most optimal graph found at that point until it finds a better one. Then, it runs the loop on this graph. It continues to do this until it finds a solution less than the specified X value. **After this change, my algorithm found a shorter tour for each of the four graphs than any of the other solutions on the day of the competition.**

Unfortunately, this does not matter for the competition, but the results are displayed below in **Figure 4**. The time to complete the initial execution and the time spent to beat the competition leader are displayed in **Figure 5**. Based on the figures, my fixed algorithm found better solutions in every graph, and it would have been able to perform three of them during the competition. This would have been a much better outcome for the competition, but I am still proud of the solution that I developed.

Graph Size	Competition-Winning Solution	Solution Found by My Algorithm After Bug Fix
250	1,904	**1,772**
1,000	1,771	**1,612**
5,000	202,564	**197,079**
15,000	196,900	**191,625**

Figure 4: The optimal solutions in the competition and by my fixed algorithm. Each of these solution files were submitted with this report on Blackboard.

Graph Size	Time Taken to Run Once	Time Taken to Beat Competition Leader
250	1.543889s	17.021187s
1,000	3.348727s	23.428087s
5,000	2m 27.139350s	1m 4.599972s
15,000	43m 11.250717s	52m 29.09418s

Figure 5: The times taken for the program to complete one time and the times taken to run the simulated annealing loop to beat the competition leader.

References

Carnegie Mellon Simulated Annealing Article - https://www.cs.cmu.edu/afs/cs.cmu.edu/project/learn-43/lib/photoz/.g/web/glossary/anneal.html

Press, W., Teukolsky, S., Vetterling, W., and Flannery, B. (1992). Numerical Recipes in C, Second Edition.