Comparative Analysis of Optimization Algorithms for the Traveling Salesman Problem

Alejandro McEwen
School of Engineering
Department of Informatics & Systems
EAFIT University
Medellin, Colombia
Email:amce@eafit.edu.co

Viviana Hoyos Sierra
School of Engineering
Department of Informatics & Systems
EAFIT University
Medellin, Colombia
Email: vhoyoss@eafit.edu.co

Julian David Ramirez
School of Engineering
Department of Informatics & Systems
EAFIT University
Medellin, Colombia
Email: jdramirezl@eafit.edu.co

1. Introduction

The Traveling Salesman Problem (TSP) is a classic problem in computer science and operations research. It is a well-known NP-hard problem that involves finding the shortest possible route that visits a set of cities exactly once and returns to the starting city. TSP's complexity and practical relevance have made it a fundamental problem in the fields of computer science and operations research, widely studied as a means of solving real-world optimization challenges. [1]

Solving TSP is crucial in many real-world applications...

Numerous algorithms have been proposed to solve the TSP, each with its own strengths and weaknesses. This article compares and analyzes the performance of three popular TSP algorithms: Brute Force, Nearest-Neighbor, and Kruskal Minimum-Spanning-Tree. Their performance will be analyzed in terms of time complexity and solution accuracy to provide insights into the strengths and weaknesses of each algorithm. This analysis can help researchers and practitioners choose the most appropriate algorithm for their specific TSP problem.

2. Traveling Salesman Problem

2.1. Context

The Traveling Salesman Problem is a classic optimization problem that involves finding the shortest route through a set of cities that starts and ends at the same city, without visiting any city more than once. Although the problem has a simple formulation, it has proven to be notoriously difficult to solve efficiently. One variation of the problem is the Hamiltonian path, where the salesman doesn't have to return to the starting city. Another variation is when the distances between the cities are not symmetrical. These variations add to the complexity of the problem. [2]

The Traveling Salesman Problem is classified as a Nondeterministic Polynomial Complete (NP-complete) problem, meaning that its complexity grows exponentially with the input size. An efficient solution is one whose complexity can be represented by a polynomial. Unfortunately, an optimal and efficient solution has not been found for this problem. [3]

The brute-force method of solving the Traveling Salesman Problem is to check every possible path through the graph, but the complexity of this solution grows factorially with the number of cities. This makes it feasible only for small input sizes. For larger problems, computer scientists have had to rely on heuristics, which are methods that do not guarantee an optimal solution but can provide a reasonably good one in much less time.

Several heuristics have been developed for the Traveling Salesman Problem, including the nearest neighbor and Kruskal Minimum Spanning Tree (MST) algorithms. These heuristics are known to give good solutions in a fraction of the time that an optimal algorithm would take.

This paper will discuss the optimal solution to the Traveling Salesman Problem and compare it with two heuristics: the nearest neighbor algorithm and the Kruskal Minimum Spanning Tree algorithm. Additionally, the time taken by each method to obtain their respective solutions will be analyzed. By comparing the efficiency and accuracy of these methods, readers can gain insights into their strengths and weaknesses and make informed decisions about which method to use for a particular problem.Ï

2.2. Real world applications

The applications of algorithms for TSP can vary depending on the structure and restrictions that the problem acquires, either by metrics such as cost, time, number of entities (people, cars, etc.) to perform the routes and if these should be different.

2.2.1. Logistics problems. In the first instance are usually related to the search for efficient routes in a company, it could be used for supply chain management systems treated as a vehicle routing problem.

> To illustrate this, let's consider a scenario where there are n customers require certain amounts of some commodities and a supplier has to satisfy

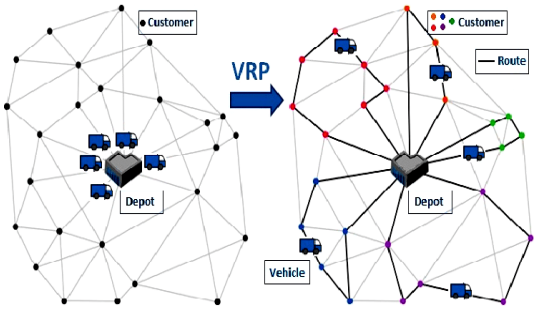
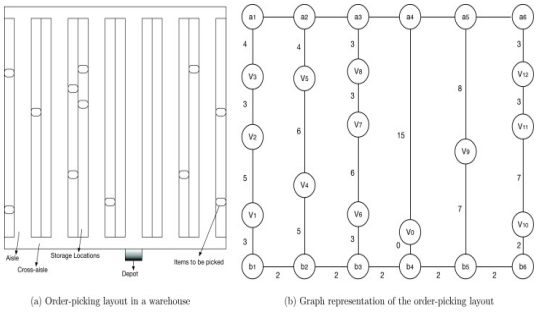


Figure 1. VRP problem can be treated as a TSP. Source: [5]

all demands with a fleet of trucks. The problem is to find an assignment of customers to the trucks and a delivery schedule for each truck so that the capacity of each truck is not exceeded and the total travel distance is minimized. [4]

Another logistic problem is the order-picking problem in warehouses. If a company has multiple warehouses and needs to deliver a list of items to various locations, it is important to select the order in such a way that the warehouse is located nearby.

The storage locations of the items correspond to the nodes of the graph. The distance between two nodes is given by the time needed to move the vehicle from one location to the other. [4]



(a) Order-picking layout in a warehouse (b) Graph representation of the order-picking layout

Figure 2. order-picking problem in warehouses problem treated as a TSP.Img Source: [6]

2.2.2. Industrial processes problems. These activities involve specific operations and processes related to manufacturing, assembly, or maintenance in various industrial sectors such as electronics, engineering, and manufacturing.

In companies in the electronics industry that specialise in the production of electronic components such as printed circuit boards, which are used in various electronic products is needed to optimize time in the manufacture and printing of circuit boards by improving drilling process productivity. Drilling is an essential step in the production of printed circuit boards, as it involves the creation of holes for the placement of components and electrical connections. A TSP approach algorithm can be used at the point where the necessary holes must be drilled to link an initial position on the circuit board to the final position, minimising the number of intermediate holes made by the drilling machine.

The 'distance' between two "cities" (holes) is given by the time it takes to move the drilling head from one position to the other. The aim is to minimize the travel time for the machine head. [4]

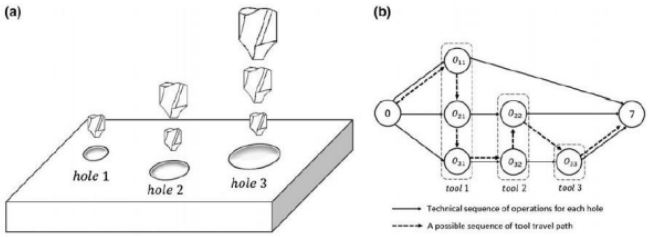


Figure 3. Multi-tool hole drilling with precedence constraints – PCTSP.Source: [7]

2.2.3. Bioinformatics and computational biology. Bioinformatics focuses on the development of computational tools to store, manage, interpret and analyse genome, proteome and biological data [8] using computational tools and methods to manage and process large datasets. The main research involves integrating information on DNA sequences, proteins, their structure and function.

In an article of the Institute of Information Systems (IWI), University of Hamburg shows one of the applications of the TSP problem to the biology and genetics field, as a way to determine the distance between the oligos (short for oligonucleotides, which are short DNA or RNA sequences), with respect to their overlapping degree.

The underlying idea is to consider the analogy between oligos in the spectrum and cities in the tour. Consequently, the distance between any two oligos is inversely proportional to the overlapping degree of the oligos. [9]

The overlapping degree refers to the existing similarity between the genetic material, according to the article, since there is an inversely proportional relationship between this parameter and the distance node to node (between oligos) it is deduced that the more similar the material of one oligo to another, the distance will decrease, given the opposite case its distance will increase.

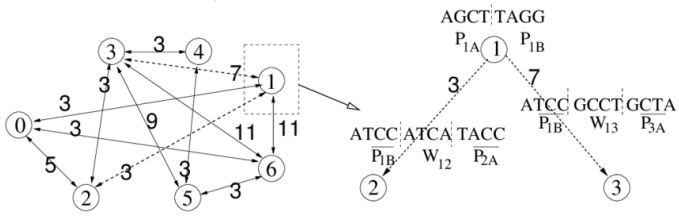


Figure 4. Graph for the travelling salesman problem (left) and its example of encoding in DNA (right). The vertex sequence is $5 \rightarrow 3$ and the edge sequence is $3 \rightarrow 5$. The optimal path for this problem is '$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 0$'. .Img Source: [10]

2.2.4. Aeronautics: aircraft scheduling. Aircraft scheduling refers to the process of organizing and coordinating the arrival and departure times of aircraft at airports. A 1988 NASA study paper, called "A Traveling-Salesman-Based Approach to Aircraft Scheduling in the Terminal Area", shows how to solve this problem, associating it to the TSP problem. The nodes of the problem are going to be the aircrafts (A1,A2,A3...An) and the weights would be related by the time given for each aircraft (time dependent on the weight, size and load of the aircraft) needed to determine the minimum gap between arrivals in the terminal area. This time is gives as a matrix as shown:

TABLE 1.- MINIMUM TIME-SEPARATION MATRIX (sec)

		Second to land		
		Small	Large	Heavy
First to land	Small	110	78	78
	Large	160	78	78
	Heavy	220	125	104

Figure 5. In the aircraft scheduling problem, intercity distances are replaced with the time separation matrix shown earlier, and the restriction of finishing the tour where it was started is removed. [11]

3. Proposed Algorithms

3.1. Brute Force

The Brute-Force approach, is an exact solution algorithm for the TSP .It is based on generating all permutations of n-1 intermediate cities (or nodes) until the best or optimal solution is found, which is the shortest tour among them. This approach is simple and easy to understand, but it can be very time-consuming and inefficient.While the size of the node number is greater, the possible solutions resulting from the permutation increase, which means that finding the minimum cost among them increases the time complexity.

1) First the matrix is initialized with the number of nodes (n) and weight values given by input as an adjacency matrix.

```
cin >> n;
    graph.resize(n, vector<long
↪   long>(n));
    int node;
    for(int i = 0; i < n; ++i){
        for(int j = 0; j < n;
↪   ++j){
            cin >> graph[i][j];
        }
    }
```

2) Then the tsp function is called. This function was implemented as an iterative loop. In tsp function a vector called "vertices" is created and initialized with the n nodes. A variable minCost is initialized with INF value,that represents an initial value that is higher than any possible weight from the graph. The iter variable is also added to keep track of the number of iterations, this being a pessimistic exit case of the program in order to prevent the algorithm from running indefinitely. This case is very likely to happen, since after a certain number of nodes, a large number of permutations and calculations of the minimum cost of each of the paths generated must be performed, therefore a large number of iterations resulting in a suboptimal output.

```
long long tsp()
{
    vector<long long> vertices;
    for (int i = 0; i < n; i++)
    {
        vertices.push_back(i);
    }
   long long minCost = INF;
   int iter=0;

}
```

3) The number of iterations is increased by 1, cost is initialized to 0, and the source node is set as the first of the vertices in the vector. Then, going through the array of vertices, the next vertex (destination) is continuously established, the source is updated with the following vertex and its weight set between it and the source node, weight which will be added to the cost variable. Then the minimum value is established between the minimum cost and the cost obtained from the path in each permutation.

```
...
do{
    iter++;
        long long cost = 0;
        int src = vertices[0];

        for (int i = 1; i <
↪   vertices.size(); i++)
        {

            int dest =
↪   vertices[i];
            cost +=
↪   graph[src][dest];
            src = dest;
        }
        cost +=
↪   graph[src][vertices[0]];

        if (minCost > cost)
            minCost = cost;
    }
...
```

4) Finally, the parameters of the iteration given by the next permutation and the pessimistic output case are established. Then minCost is returned.

```
while(next_permutation(vertices.begin(),
↪   vertices.end()) &&
↪   iter<niterMax);
return minCost;
```

3.1.1. Time Complexity. As said before, the complexity time of this solution increases factorially, that is $O(n!)$ therefore this approach is not recommended for cases where the data input comes from a very big and complex problem.

3.2. Nearest-Neighbor

The Nearest-Neighbor algorithm is a heuristic approach to finding the shortest path between two points. This method involves choosing the closest neighboring point to the current point and repeating the process until the destination point is reached.

The algorithm assumes that selecting the nearest point at each step will lead to the shortest possible path overall, making it a greedy strategy. Although this approach may not always result in the absolute shortest path, it generally provides good results and is straightforward to implement. [12]

Overall, the Nearest-Neighbor algorithm is a simple yet effective way to solve the shortest path problem, making it a popular choice in various applications such as routing and logistics.

The algorithm has three steps:

1) First, create the graph of the problem from the input.

```
cin >> n;
dist.resize(n, vector<long
↪   long>(n));
visited.resize(n, false);
int node;
for(int i = 0; i < n; ++i){
    for(int j = 0; j < n; ++j){
        cin >> dist[i][j];
    }
}
```

2) Then start a path from each node and grab the minimum from does paths.

```
long long nearestNeighbor(){
    long long minCircuit = INF;
    for(int i = 0; i < n; ++i){
        fill(visited.begin(),
↪   visited.end(), false);
        minCircuit =
↪   min(minCircuit,
↪   nearestNeighborFromNode(i,
↪   i));
    }
    return minCircuit;
}

```

3) Finally, find the path from visiting the shortest edge that is available in the graph, and when there are no edges available visit return to the starting edge. This is done in a recursive way to simplify the programming.

```
long long
↪   nearestNeighborFromNode(int
↪   node, int& start){
    int nextNode = -1;
    long long cost = INF;
    visited[node] = true;
    for(int i = 0; i < n; ++i){
        if(!visited[i] &&
↪   dist[node][i] < cost){
            nextNode = i;
            cost = dist[node][i];
        }
    }
    if(nextNode == -1){
        return dist[node][start];
    }
    return cost +
↪   nearestNeighborFromNode(nextNode,
↪   start);
}
```

3.2.1. Time Complexity. The Nearest-Neighbor algorithm has a time complexity of $O(n^3)$, as it involves three nested for loops iterating through n elements. Initially, the algorithm considers each node as a starting point and then

examines every edge connected to that node. Since the graph is complete, this step effectively requires examining every node again. Finally, the algorithm searches through every node to find the shortest path.

Although there is a slightly more efficient version of the algorithm that eliminates one of the for loops, resulting in a time complexity of $O(n^2)$, this approach may produce sub-optimal results. Specifically, when the algorithm reaches the last node, it must return to the first node regardless of the cost, potentially leading to a sub-optimal overall path.

3.3. Greedy

The proposed algorithm is based on the Kruskal Minimum-Spanning-Tree (MST) Algorithm with some modifications.

Kruskal is a greedy algorithm used to find the MST of an undirected graph. It starts with an empty set of edges and on each iteration adds the edge with the smallest weight that does not create a cycle until all vertices are included in the tree. We can apply this algorithm to the TSP, because a tour that visits all vertices can be represented as a MST with some other properties.

To adapt Kruskal's algorithm for TSP, we make a few modifications.

1) First, we create a list of edges in the format (Source, Target, Weight)

```
struct edge{
    ll u, v, w;
    edge(ll _u, ll _v, ll _w) :
↪   u(_u), v(_v), w(_w) {}
    edge() : u(0), v(0), w(0) {}
    bool operator<(const edge
↪   &other) const {
        return w < other.w;
    }
};

for (ll i = 0; i < graph.size();
↪   i++) {
    for (ll j = 0; j <
↪   graph[i].size(); j++) {
        if (graph[i][j] == 0)
↪   continue;
        if (i == j) continue;
        edges.push_back(edge(i, j,
↪   graph[i][j]));
    }
}
```

2) Next, we sort the edges by weight

```
sort(edges.begin(), edges.end());
```

3) We then add the edges if they don't create cycles (By searching with union find) and are not already connected to any other edge (We have a list with the degree of each node)

```
// Union find and break condition
ll find(ll u) {
    return parent[u] == u ? u :
↪   parent[u] = find(parent[u]);
}

bool merge(ll u, ll v) {
    u = find(u), v = find(v);
    if (u == v) return false;
    return true;
}
//-------
if (!merge(e.u, e.v)) continue;

// Node degree break
if ((degree[e.u] < 2) &&
↪   (degree[e.v] < 2))

```

We stop when our tour has N nodes. This is because we have created a path that goes over each vertex once.

```
if (path.size() == n) break;
```

This algorithm, being greedy, has limitations because of its nature. The biggest one comes from the decision-making as:

1) Sorting isn't always the most optimal option
2) Once we make a choice we can't go back

3.3.1. Time Complexity. The three key sections to calculate this are:

1) When we create the edge list we go through all the items in the original matrix, this is N^2. You could argue this is part of the input process so it wouldn't count, but as this is an algorithm-specific action we will count it.
2) When we sort the edges which is $MlogM$
3) When we do the actual path construction. The loop runs from 0 to M, it being the number of edges. On each iteration we run a Union Find for a target and source, which runs down to $logN$ (We use N as UF runs for each vertex, not edge). These two combined form a $MlogN$ time complexity

Based on the last items, the time complexity of this algorithm is $O(N^2 + M \log |M| + M \log |N|)$, where $|M|$ is the number of edges and $|N|$ is the number of vertices in the graph.

4. Performance Analysis

In this section we will present how each algorithm performed on the dataset.

The dataset consisted of 29 files, with each one containing an N*N matrix where each (i, j) position represented the edge weight between i and j. The N went as low as 16 and as high as 1002.

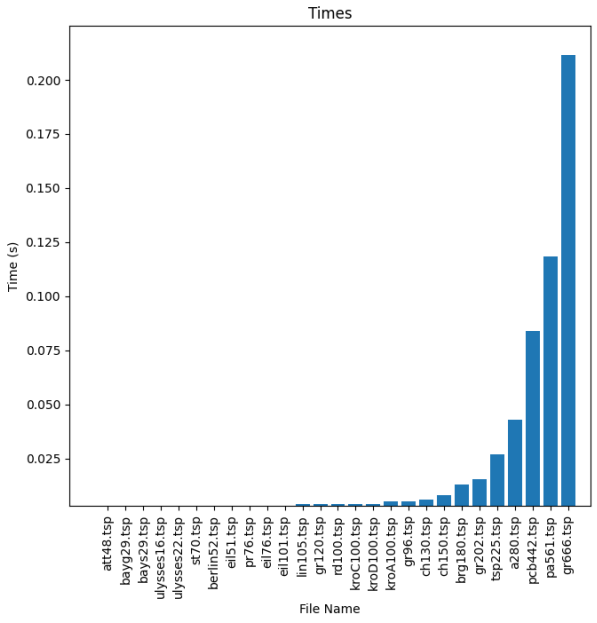
Each case was valid and had a tested optimal route. The testcases were taken from a Waterloo University dataset [13]

Now comes the result for each algorithm.

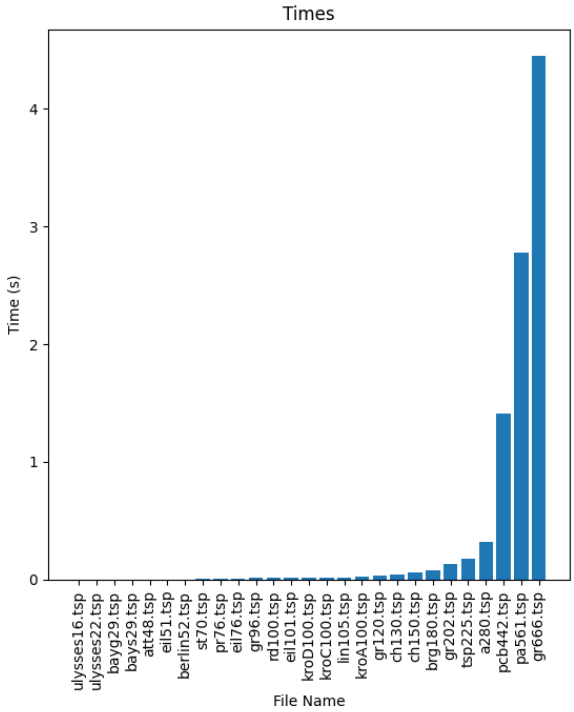
4.1. Time

The following are the times it took each algorithm to finish the dataset. The graph shows each case in ascending order and with seconds as the unit of measurement.

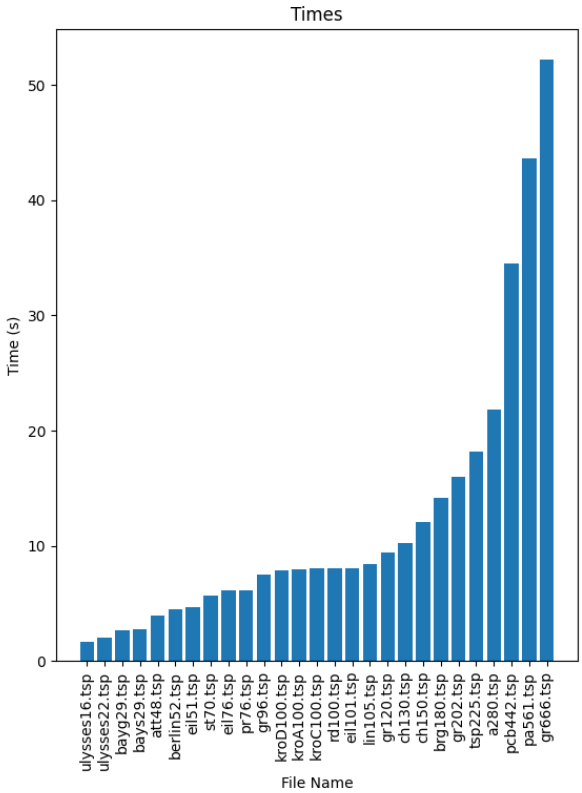
Greedy



Nearest Neighbor



Brute Force



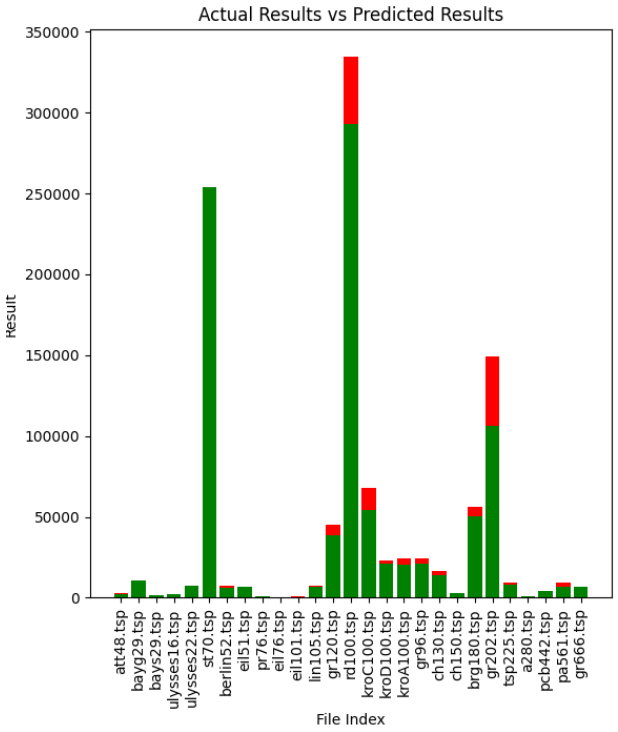
As expected $BruteForce$ was the slowest running algorithm between the three, managing to get the highest time of all with over 50 seconds, this happens clearly as it computes all possible permutations of the vertices. Meanwhile, the $NearestNeighbor$ got a max of 5 seconds and an average of under 1 second for files with an N lower than 400.

The fastest one was the greedy, with each case running under 1 second. It is faster because it makes locally optimal choices at each step without considering the global picture. This makes them much simpler and faster than other algorithms that require more extensive computations to determine the optimal path.

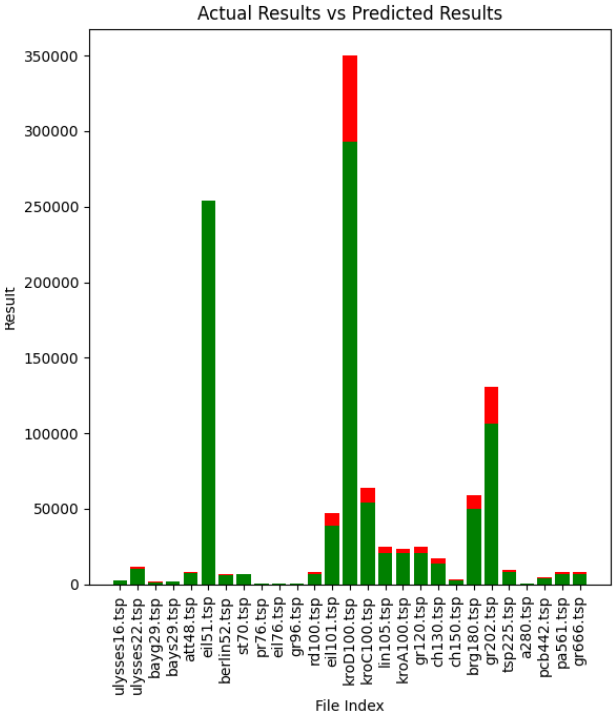
4.2. Accuracy and precision

The following are the plots showing how close, for each case, each algorithm got to the real optimal answer.

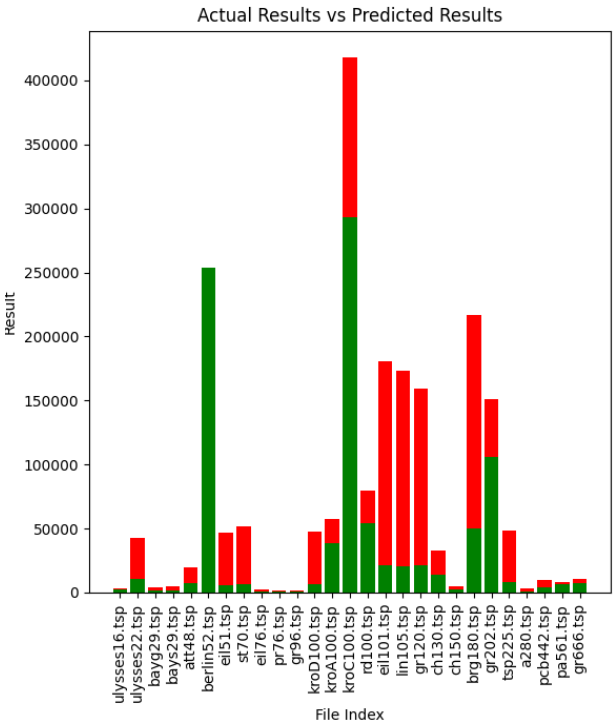
Greedy



Nearest Neighbor



Brute Force



5. Conclusion

The traveling salesman problem is known as an NP-hard problem,meaning that it is a non-deterministic polynomial-time hard problem, indicating that there is no known algorithm that can solve these problems in polynomial time. Being a problem that can be related to many others in real life applications, it is of great importance to look for ways to attack and solve it. It has been proven that even though an exact solution is needed,using approximation algorithms offer a more efficient alternative. Each performance was analyzed based on key parameters such as time complexity and solution accuracy in the three types of approaches selected to solve it, from the exact solution ($BruteForce$) algorithm, the $Greedy$ algorithm, and $NearestNeighbor$. During the analysis, it was observed that the $BruteForce$ algorithm's accuracy decreased significantly as the number of nodes increased.Therefore the gap between the output it provided and the real optimal solution grew much more than the other approximation algorithms when finding the solution with the same amount of data, also is important to emphasize that by generating so many permutations, the number of iterations increased to such an extent that it reached the point where it had to exit the cycle using the pessimistic exit case, i.e. the number of iterations needed to reach the optimal solution was not completed.On the other hand, conversely, the computational complexity time increased the most in contrast to the other two approaches. Now, comparing the approximation algorithms, in terms of computational speed, $Greedy$ was the fastest. But in terms of accuracy and finding the most optimal solution $NearestNeighbor$ was better.

The choice between the $Greedy$ algorithm and the $NearestNeighbor$ algorithm depends on the specific requirements of the problem. If finding the optimal solution is critical, the $NearestNeighbor$ algorithm is preferred. However, if efficiency and speed are the primary concerns, the $Greedy$ algorithm may be a may be an appropriate choice.

References

[1] "A brief History of the Travelling Salesman Problem - The OR Society — theorsociety.com." https://www.theorsociety.com/about-or/or-methods/heuristics/a-brief-history-of-the-travelling-salesman-problem/. [Accessed 30-May-2023].

[2] M. Jünger, G. Reinelt, and G. Rinaldi, "The traveling salesman problem," *Handbooks in operations research and management science*, vol. 7, pp. 225–330, 1995.

[3] C. H. Papadimitriou, "The adjacency relation on the traveling salesman polytope is np-complete," *Mathematical Programming*, vol. 14, no. 1, pp. 312–324, 1978.

[4] R. Matai, S. P. Singh, and M. L. Mittal, "Traveling salesman problem: an overview of applications, formulations, and solution approaches," *Traveling salesman problem, theory and applications*, vol. 1, 2010.

[5] A. Gupta and S. Saini, "An enhanced ant colony optimization algorithm for vehicle routing problem with time windows," in *2017 ninth international conference on advanced computing (ICOAC)*, pp. 267–274, IEEE, 2017.

[6] W. Lu, D. McFarlane, V. Giannikas, and Q. Zhang, "An algorithm for dynamic order-picking in warehouse operations," *European Journal of Operational Research*, vol. 248, no. 1, pp. 107–122, 2016.

[7] R. Dewil, İ. Küçükoğlu, C. Luteyn, and D. Cattrysse, "A critical review of multi-hole drilling path optimization," *Archives of Computational Methods in Engineering*, vol. 26, pp. 449–459, 2019.

[8] "What can i do with a major in bioinformatics." https://www.biology.pitt.edu/sites/default/files/publication-images/BINF_Info%202157.pdf, 2023.

[9] M. Caserta and S. Voß, "A hybrid algorithm for the dna sequencing problem," *Discrete Applied Mathematics*, vol. 163, pp. 87–99, 2014.

[10] J. Youn Lee, S.-Y. Shin, S. June Augh, T. Hyun Park, and B.-T. Zhang, "Temperature gradient-based dna computing for graph problems with weighted edges," in *DNA Computing: 8th International Workshop on DNA-Based Computers, DNA8 Sapporo, Japan, June 10–13, 2002 Revised Papers*, pp. 73–84, Springer, 2003.

[11] R. A. Luenberger, "A traveling-salesman-based approach to aircraft scheduling in the terminal area," tech. rep., 1988.

[12] "FAPPlet — digitalfirst.bfwpub.com." http://digitalfirst.bfwpub.com/math_applet/asset/3/TSP_NN.html.

[13] U. of Waterloo, "TSP Test Data — math.uwaterloo.ca." https://www.math.uwaterloo.ca/tsp/data/index.html. [Accessed 30-May-2023].