UNIVERSITÀ DEGLI STUDI DI PADOVA
DEPARTMENT OF MATHEMATICS
COMPUTER SCIENCE

Advanced Algorithms
Traveling Salesman Problem

Homework 2 Report

Group: 5	Sara Buttau, Salvatore Gatto, Emad Efatinasab
Student IDs:	2036579, 2044587, 2044422

May 15, 2022

Abstract

The Traveling Salesman Problem can be defined by the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?". It is an NP-hard problem so, there is no polynomial-time constant factor approximation algorithm for TSP, unless P=NP.

In this report we present our implementation of Nearest Neighbor's, Random Insertion's and 2-Approximation's algorithms to find an approximate solution of a special case of the TSP, called Metric TSP. We first briefly introduce the problem in question and the three algorithms. Consequently, we explain our implementation choices and finally present our results, according to the three algorithms' computational time and error rate.

1 Traveling Salesman Problem: the theory

1.1 Introduction

Given an undirected, complete and weighted graph G = (V,E), the determination of the shortest Hamiltonian Cycle is what defines the Traveling Sales Problem (TSP). Although the problem is NP-hard, it is possible to find a good approximation of the solution only in case the graph satisfies the condition of triangle inequality, such that

$$\forall u, v, w \in V c(u, w) \leq c(u, v) + c(v, w) \tag{1.1}$$

This special case of TSP is called "Metric TSP" or "Triangle TSP" and it can be solved finding a good approximation of the solution by reducing it to the Hamiltonian Cycle problem.

1.2 2-approximation algorithm

The 2-approximation algorithm solves the problem of Metric TSP to find a Hamiltonian Cycle whose cost is at most twice than the cost of the optimal solution. The idea behind the algorithm is to use Prim's algorithm with heap structure to find the Minimum Spanning Tree (MST) of the graph G. The MST is then transformed in a Hamilton Cycle, applying a "preoder" visit to it. The algorithm is the following:

Algorithm 1 Preorder

```
print(v)
**if** internal(v), do **then**
    **for** each edge $u \in children(v)$ do **do**
        Preorder(u)
    **end for**
**end if**
```

Algorithm 2 Approximation Metric TSP

```
root = graph.V[0]
mst = Prim(graph)
preordered_nodes = ø
visited_nodes = Preorder(preoredered_nodes,root,mst)
visited_nodes.append(visited_nodes[0])
return visited_nodes
```

1.3 Heuristics

1.3.1 Nearest neighbor

The Nearest Neighbor algorithm was one of the first algorithms used to solve the Travelling Salesman Problem approximately.The algorithm quickly yields a short tour, but usually not the optimal one. The Nearest Neighbor Algorithm begins at any vertex and follows the edge of least weight from that vertex. At every subsequent vertex, it follows the edge of least weight that leads to a vertex not yet visited, until it returns to the starting point.

Algorithm 3 Nearest Neighbor Heuristics

```
unvisited list $\leftarrow V[]$
visited list $\leftarrow V[0]$
while all vertices has been visited do
    current node $\leftarrow$ visited [-1]
    adjacent vertexes $\leftarrow$ adjacent vertex to current node
    next node $\leftarrow$ MIN(adjacent vertexes that are not in the visited list)
    visited list $\leftarrow$ next node
end while
return Visited list
```

1.3.2 Random Insertion

Random Insertion is a *O(logn)*-approximation heuristics for the Metric TSP. The algorithm is based on a random choice of the new vertex to insert in the circuit, and uses the triangular inequality to choose the position in which to insert it. The algorithm is the following:

Algorithm 4 Random Insertion Heuristics

```
starting_node $\leftarrow 0$
circuit $\leftarrow []$
$j \leftarrow$ node adjacent to 0 for which w(0,j) is minimized
while circuit does not cover all vertices do
    $(i, j) \leftarrow$ edge in the circuit that minimizes w(i,k) + w(k,j) - w(i,j)
    insert k between i and j in the circuit
end while
return circuit
```

2 Our implementation

Our experiments are coded in Python and run on Google Colab. After defining our algorithms' functions, we run each of them for 1000 iterations on each input graph, compute the running time in nanoseconds using *perf_counter_ns()*, and take the average per graph. Our approximate solution is given by the weight of the cycle found, which we also use to compute the error rate:

$$\frac{approximate\_solution - optimal\_solution}{optimal\_solution}$$

2.1 Graph data structure

We implemented a class Graph in the following way:

- self.V: list of vertices of the graph.
- self.E: dictionary representing the edges of the graph. The keys of the dictionary are the tuples of the adjacent nodes *(v1,v2)* and the values are the respective weights.
- self.adjacency_list: default dictionary to represent the graph as an adjacency list.

The class Graph also contains the following methods:

- def add_vertex(self,v): the method adds the vertex “v” to the list of vertices of the graph.
- def remove_vertex(self,v): the method removes the vertex “v” from the list of vertices of the graph.

- def add_edge(self, source_node, destination_node, weight): the method adds the edge (source_node, destination_node) with its *weight* to the dictionary *graph.E*. If source_node and destination_node are not inside the list of the vertices of the Graph, the method also updates the variable self.V. The graph is also updated appending the new edge to the adjacency list.
- def get_edge(self,v1,v2): the method returns the weight of the edge (v1,v2) by accessing the dictionary graph.E.

To initialize the graph, points are first stored in a list of elements of class Point called *space_points*. Each point contains the name of the node and the spatial coordinates x and y. This list is used to compute the weights of the edges, according to the variable *edge_weight_type*: if "EUC_2D", the weight is computed through the euclidean distance between the two points, whereas if "GEO", the point's coordinates are first converted to radians using the function *convert_to_rad*, and the weight is computed through the geographical distance. The weights are then added to the dictionary graph.E, along with the vertices of the graph[1]. We also checked that every graph respected the triangular inequality:

$$\forall u, v, w : weight(u, v) \leq weight(u, w) + weight(w, v)$$

2.2 2-approximation algorithm

Our implementation of 2-approximation's algorithm consists of two methods:
$Preorder(visited\_nodes, V, MST)$ and $Approx\_Metric\_TSP()$.
The method $Preorder$ is used to transform the Minimum Spanning Tree to a Hamiltonian Cycle. The method visits the node which we pass as parameter and if it is internal to the tree, it calls recursively the function for each child of the node.

The method $ApproximationMetricTSP$ starts choosing the root node to use Prim's algorithm with heap structure. The node we have chosen is the first of the list of vertices of G. Prim's algorithm is used to get the Minimum Spanning Tree of the graph. Once we have the MST, the algorithm calls the method *Preorder* to transform the MST to a Hamiltonian Cycle and saves the path inside the list *visited_nodes*. The first node is added at the end of the list to close the cycle. In the end, the method returns the list *visited_nodes*.

2.3 Nearest Neighbor

For the implementation of Nearest Neighbor:

[1] We only store the names of the vertices in graph.V, as we do not need the points' coordinates anymore

- We first copy all of the graph's vertices in the list into a new list called unvisited, then we make another list called visited and append the first vertex of the graph to this list. We use these variables to keep track of the graph's visited and unvisited vertices.
- While the sizes of the visited and unvisited lists differ: we will create a variable current node and append the node we are considering to it.
- We will use the "graph.adjacency list.get" function to find nodes that are adjacent to the current node.
- We will only take into account adjacent nodes that are not on the visited list.
- we choose the node with minimum weight and append that into the visited list.
- we will continue until all of the vertices of the graph has been visited .

2.4 Random Insertion

For our implementation of Random Insertion's heuristics, we initialize an empty list called *circuit*, in which we will store our cycle. We use the variable *unvisited* to keep track of the visited nodes in the graph (we initialize it to a copy of graph.V).

Starting from the first node of the graph, we look for the minimum-weighted edge *(0,j)* adjacent to it, through our graph.adjacency_list. Then, we insert both node 0 and node j to the circuit, and remove them from the unvisited nodes.
Iterating while *unvisited* is not empty:

- a random node k is selected from the unvisited ones with Python's random choice function.
- we look for an edge in circuit that minimizes:

  $$triangular\_inequality = w(i,k) + w(k,j) - w(i,j)$$

  and we store it in the variable

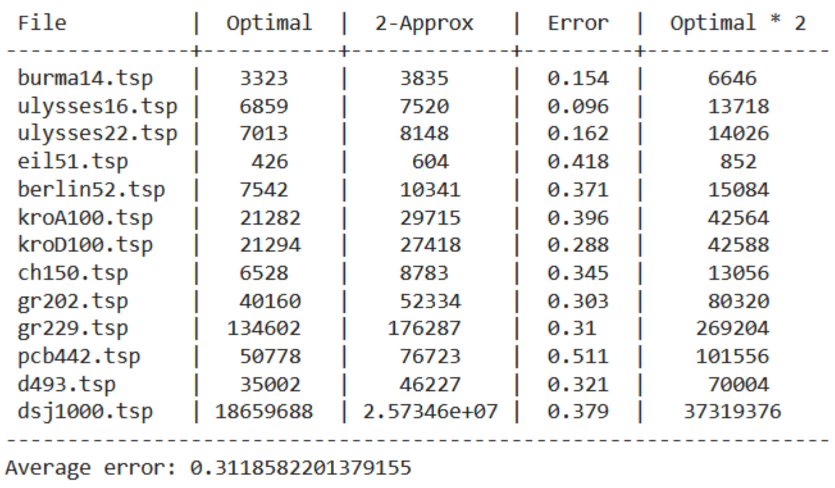
  min_triangular $\leftarrow$ *(index of i, index of j, triangular inequality)*.
- we insert k after vertex *i* in the circuit, and remove it from the unvisited nodes.

Lastly, we add the first node at the end of the list to close the cycle, and return the circuit.

3 Results

3.1 2-approximation

As figure 3.1 shows, our solutions with 2-approximations's algorithm are at most twice the optimal solution. The error rate is on average around 31%. [2]



```
File           | Optimal  | 2-Approx    | Error  | Optimal * 2
---------------+----------+-------------+--------+--------------
burma14.tsp    |  3323    |   3835      | 0.154  |   6646
ulysses16.tsp  |  6859    |   7520      | 0.096  |   13718
ulysses22.tsp  |  7013    |   8148      | 0.162  |   14026
eil51.tsp      |   426    |    604      | 0.418  |    852
berlin52.tsp   |  7542    |   10341     | 0.371  |   15084
kroA100.tsp    |  21282   |   29715     | 0.396  |   42564
kroD100.tsp    |  21294   |   27418     | 0.288  |   42588
ch150.tsp      |  6528    |   8783      | 0.345  |   13056
gr202.tsp      |  40160   |   52334     | 0.303  |   80320
gr229.tsp      | 134602   |  176287     | 0.31   |  269204
pcb442.tsp     |  50778   |   76723     | 0.511  |  101556
d493.tsp       |  35002   |   46227     | 0.321  |   70004
dsj1000.tsp    | 18659688 | 2.57346e+07 | 0.379  |  37319376
-----------------------------------------------------------------
Average error: 0.3118582201379155
```

Figure 3.1: 2-approximation's error rate with respect to the optimal solution

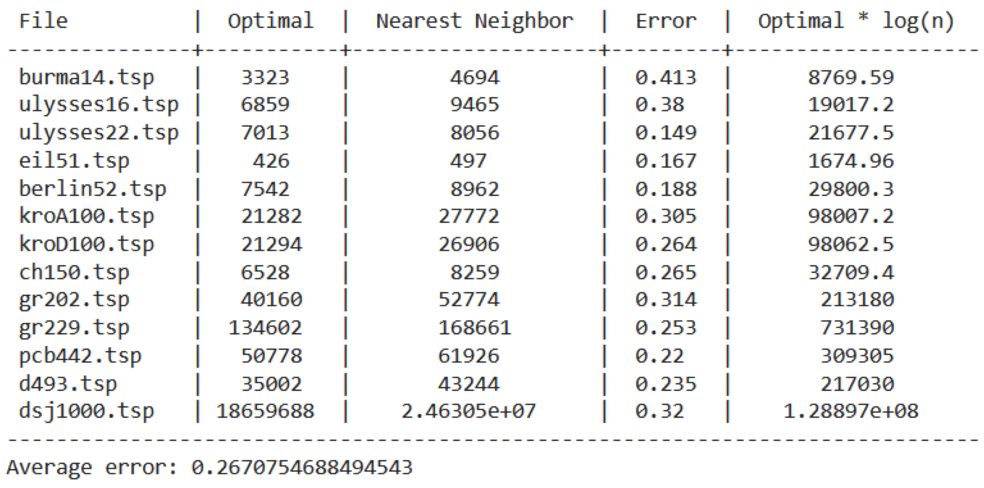
3.2 Nearest Neighbor

As illustrated in 3.2, our solutions using Nearest Neighbor's heuristics are less than twice as good as the optimal solution. On average, the error rate is around 26%.

[2] In the notebook we check that for every approximate solution with 2-approximation's algorithm:

$$approx\_solution \leq 2 * optimal.$$

.

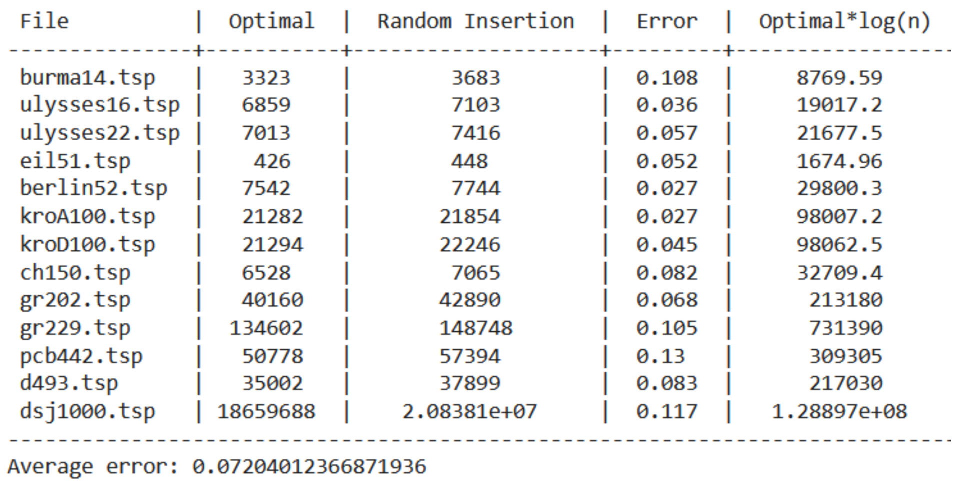


```
 File           |  Optimal  |  Nearest Neighbor  |  Error  |  Optimal * log(n)
----------------+-----------+--------------------+---------+-------------------
 burma14.tsp    |   3323    |        4694        |  0.413  |      8769.59
 ulysses16.tsp  |   6859    |        9465        |  0.38   |      19017.2
 ulysses22.tsp  |   7013    |        8056        |  0.149  |      21677.5
 eil51.tsp      |    426    |        497         |  0.167  |      1674.96
 berlin52.tsp   |   7542    |        8962        |  0.188  |      29800.3
 kroA100.tsp    |   21282   |       27772        |  0.305  |      98007.2
 kroD100.tsp    |   21294   |       26906        |  0.264  |      98062.5
 ch150.tsp      |   6528    |        8259        |  0.265  |      32709.4
 gr202.tsp      |   40160   |       52774        |  0.314  |       213180
 gr229.tsp      |  134602   |       168661       |  0.253  |       731390
 pcb442.tsp     |   50778   |       61926        |  0.22   |       309305
 d493.tsp       |   35002   |       43244        |  0.235  |       217030
 dsj1000.tsp    | 18659688  |    2.46305e+07     |  0.32   |    1.28897e+08
------------------------------------------------------------------------------
Average error: 0.2670754688494543
```

Figure 3.2: Nearest Neighbor's error rate with respect to the optimal solution

3.3 Random Insertion

As figure 3.3 shows, our solutions with Random Insertion's heuristics range around the values of the optimal solution. The error rate is on average around 7%, and every approximate solution we obtained is in fact within *O(logn)* with respect to the optimal solution.[3]



```
 File           |  Optimal  |  Random Insertion  |  Error  |  Optimal*log(n)
----------------+-----------+--------------------+---------+-----------------
 burma14.tsp    |   3323    |        3683        |  0.108  |     8769.59
 ulysses16.tsp  |   6859    |        7103        |  0.036  |     19017.2
 ulysses22.tsp  |   7013    |        7416        |  0.057  |     21677.5
 eil51.tsp      |    426    |        448         |  0.052  |     1674.96
 berlin52.tsp   |   7542    |        7744        |  0.027  |     29800.3
 kroA100.tsp    |   21282   |       21854        |  0.027  |     98007.2
 kroD100.tsp    |   21294   |       22246        |  0.045  |     98062.5
 ch150.tsp      |   6528    |        7065        |  0.082  |     32709.4
 gr202.tsp      |   40160   |       42890        |  0.068  |      213180
 gr229.tsp      |  134602   |       148748       |  0.105  |      731390
 pcb442.tsp     |   50778   |       57394        |  0.13   |      309305
 d493.tsp       |   35002   |       37899        |  0.083  |      217030
 dsj1000.tsp    | 18659688  |    2.08381e+07     |  0.117  |   1.28897e+08
----------------------------------------------------------------------------
Average error: 0.07204012366871936
```

Figure 3.3: Random Insertion's error rate with respect to the optimal solution

[3] In the notebook we check that for every approximate solution with random insertion,

$$approx\_solution \leq optimal * log(n).$$

3.4 Originalities

We further analyze our results with some additional plots, particularly useful for comparing the algorithms' error rates, running times and quality of the solutions.

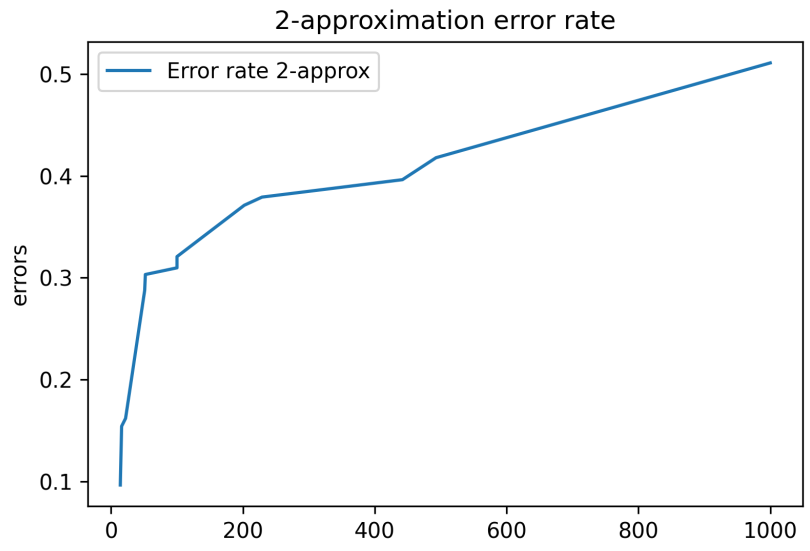


Figure 3.4: 2-approximation algorithm's error rate with respect to the optimal solution

As figures 3.4,3.5 and 3.6 show, the error rates of the three algorithms have a similar behavior, as they all increase with size. In particular, for smaller sizes the plots show a faster growth for all algorithms. For 2-approximation, the error rate seems to increase at a slower pace from size 100 onwards, whereas for the two heuristics it presents sudden sharp changes until size 500, after which it keeps increasing in a linear way. Overall, Random Insertion is the algorithm with the lowest error rate, reaching around 12% for 1000 nodes.

We also plotted the approximate solutions found by our three algorithms with respect to the optimal solution. In this case, figures 3.7, 3.8 and 3.9 present an extremely similar performance. As expected, the three algorithms' solutions are bigger than the optimal, as they are in fact an approximation. Moreover, it is important to stress that for smaller input graphs, the difference between our algorithms' and the optimal solution is practically imperceptible, which means they do offer a good approximation. It is only for graphs with more than 500 nodes that we can appreciate the differences. As a matter of fact, Random Insertion undoubtedly offers the best behavior, having the closest slope to the optimal one, and reaching a lower total weight for the approximate solution (around 2e7 compared to 2.5e7 of the other two

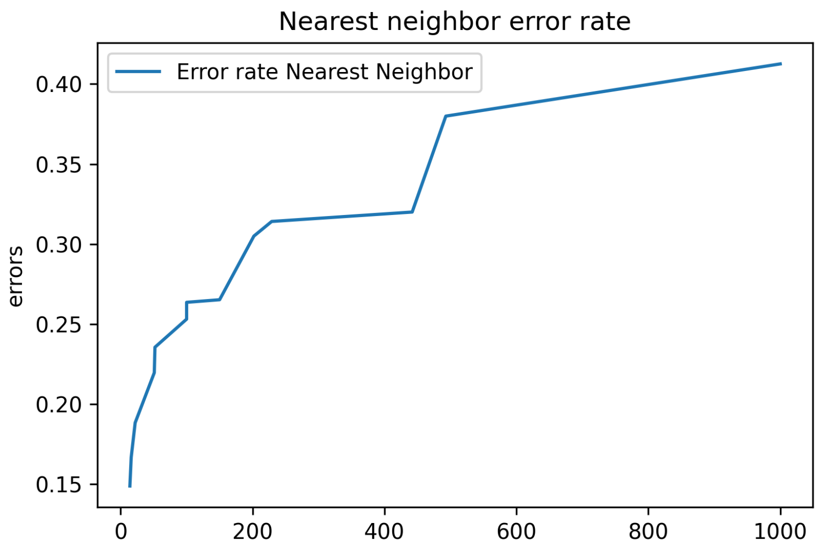


Figure 3.5: Nearest Neighbor algorithm's error rate with respect to the optimal solution

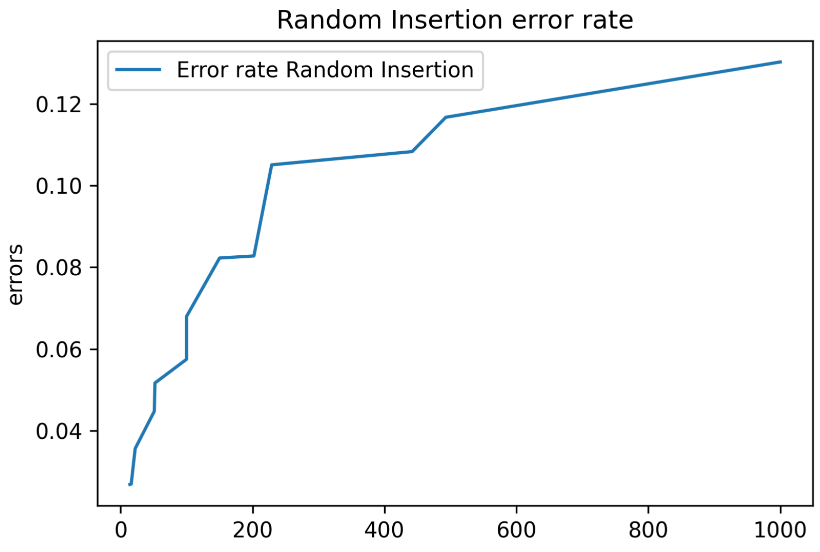


Figure 3.6: Random Insertion algorithm's error rate with respect to the optimal solution

algorithms). Figure 3.10 shows the three algorithms in one plot, thus allowing us to compare Nearest Neighbor and 2-approximation. The latter, in fact, is definitely the worst-performing one in terms of quality of solution, as it returns the most expensive

Hamiltonian cycle in terms of weights.

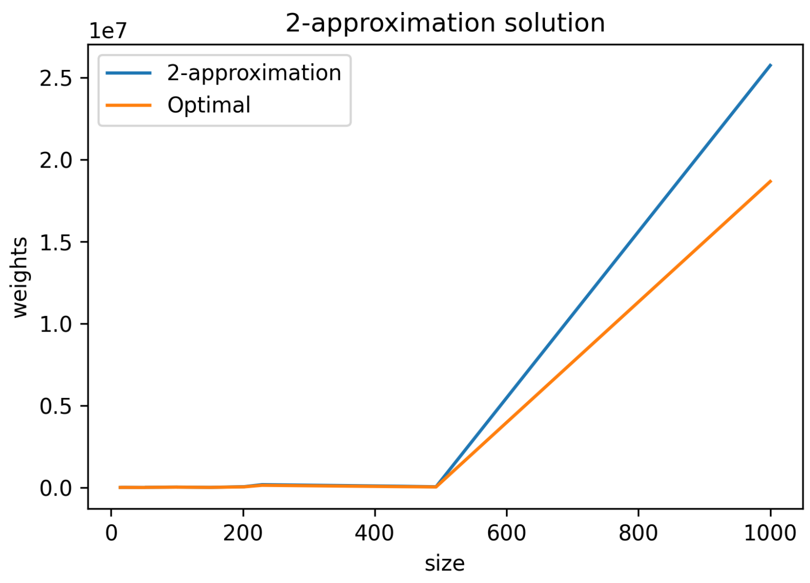


Figure 3.7: Solutions for 2-approximation algorithm with respect to the optimal

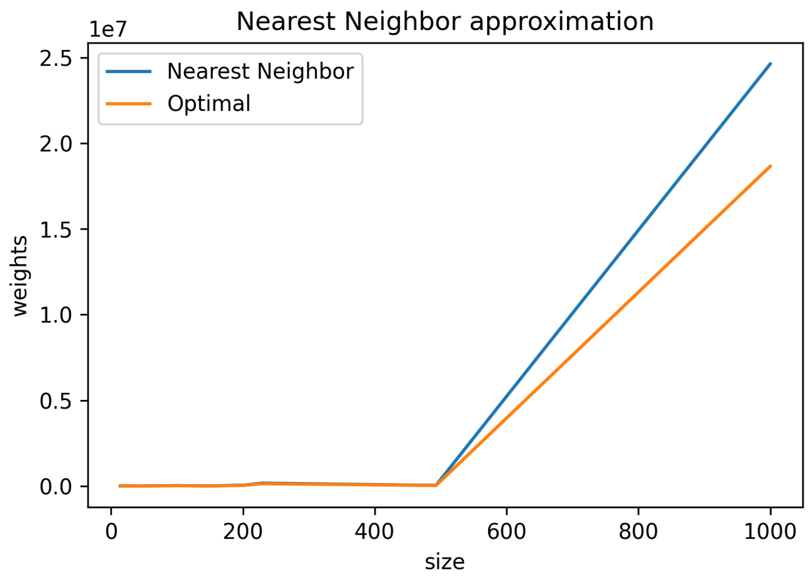


Figure 3.8: Solutions for 2-approximation algorithm with respect to the optimal

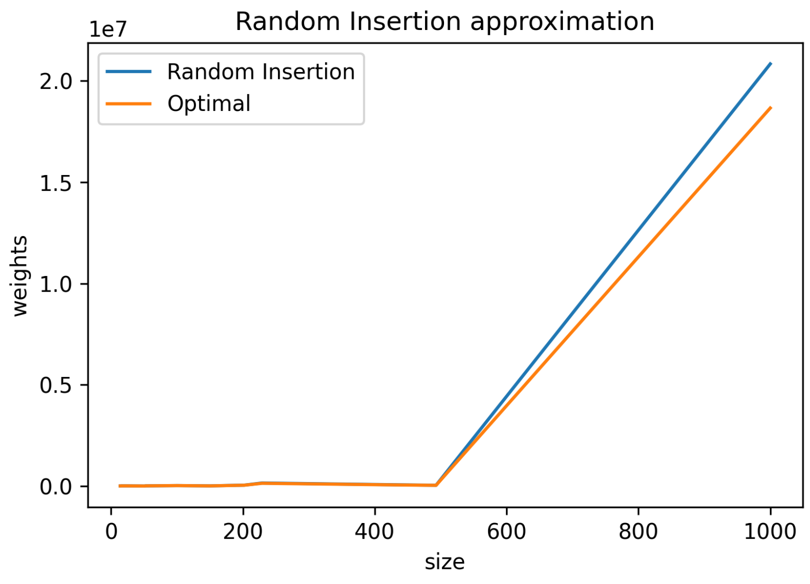


Figure 3.9: Solutions for Random Insertion algorithm with respect to the optimal

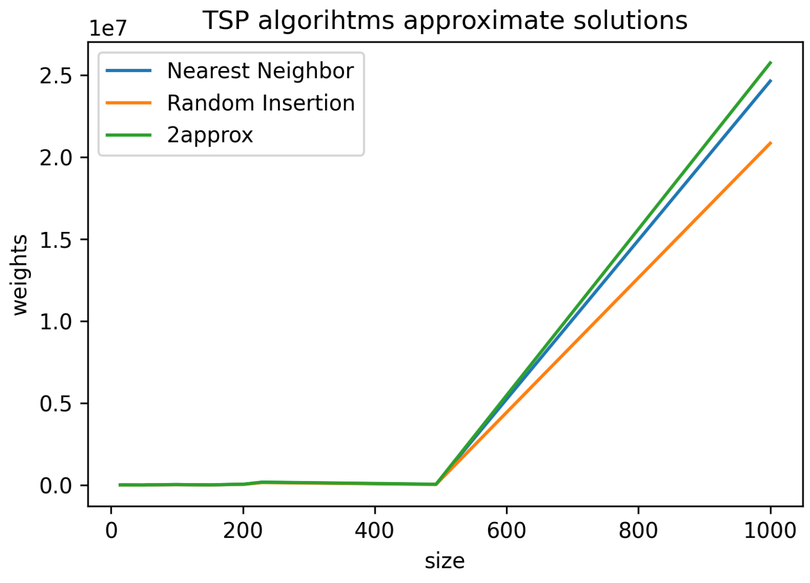


Figure 3.10: TSP algorithms approximate solutions comparisons

Lastly, we plotted the three algorithms' running times in figure 3.11. Once again, our implementation of Random Insertion is definitely the most efficient one, followed by 2-approximation and Nearest Neighbor. Although for smaller input sizes (≤ 200) the three algorithms are circa computationally equivalent, for larger sizes the three curves diverge, Random Insertion reaching an average running time of 1e06 ns and Nearest Neighbor of 6e06 ns. See figure 5.1 in the Appendix section for more in-depth values.

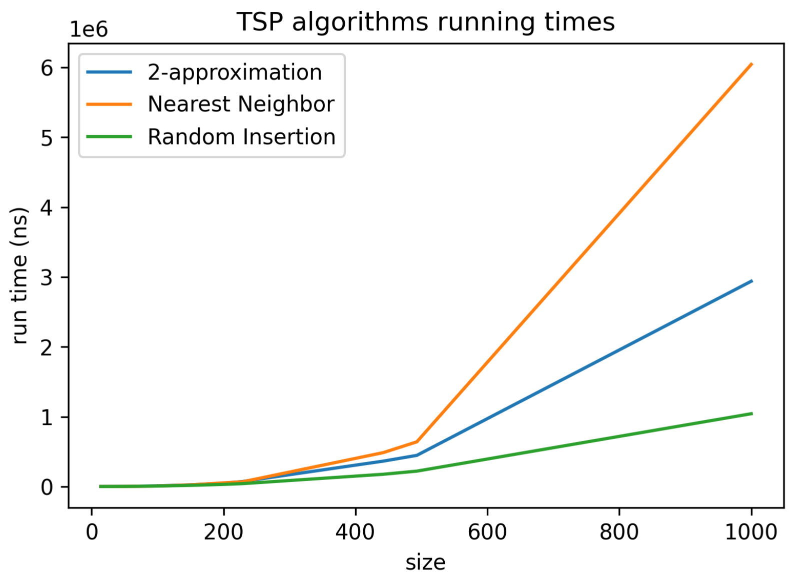


Figure 3.11: TSP algorithms running times comparisons

4 Conclusions

In conclusion, we have implemented three algorithms in Python for solving the Traveling Salesman Problem, given as input a complete, weighted and undirected graph G that respects the triangle inequality condition. We implemented a 2-approximation's algorithm to get a solution that is at most twice the optimal solution of the problem, and two heuristics to get a solution with a log(n) approximation, which are Nearest Neighbor and Random Insertion. After briefly introducing in chapter 1 the theory behind the main concepts in question, such as the TSP problem and the three algorithms, in chapter 2 we presented a description of our implementations, describing the main data structures we used, and in chapter 3 we introduced our results.

Each algorithm was executed for 1000 iterations on each graph given as input. We presented our results with an in-depth analysis of the algorithms' differences in terms of quality of performance (i.e. the weight of the approximate solution with respect to the optimal one), the error rate and the average running times with respect to size.

Overall, we found that for smaller sizes the three algorithms can be considered about equivalent, although as size increases, they tend to different behaviors.
The worst approximation is given by the 2-approximation algorithm, with an average error of 0.31. As for Nearest Neighbour and Random Insertion, Random Insertion is the most precise one, with an average error of 0.07%, against the 0.26% average error of Nearest Neighbour.

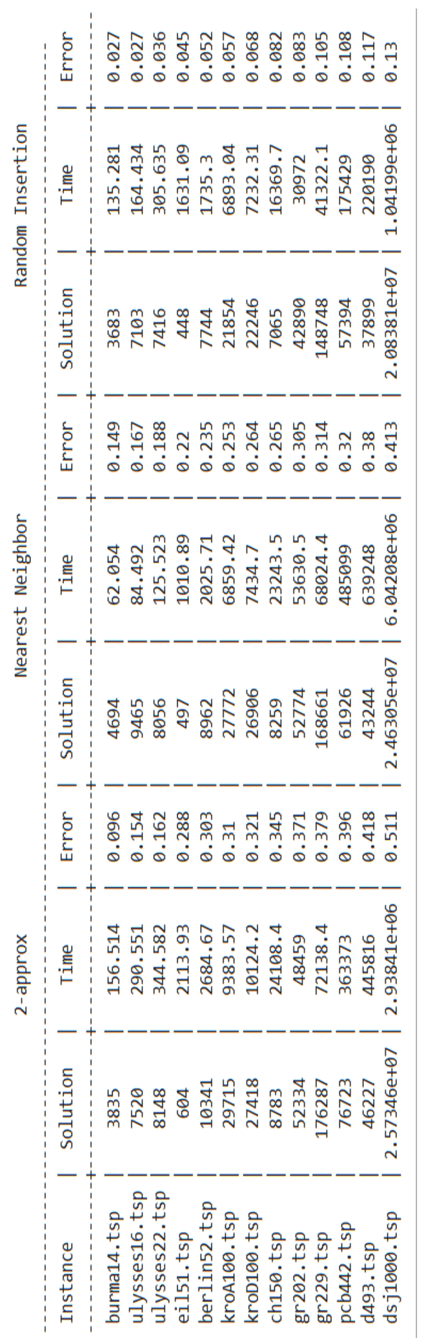
Finally, as for their running times, Random Insertion once again proved to be the most efficient one, reaching an upper-bound of average running time of 1e6 nanoseconds, compared to the worst performing 2-approximation algorithm, which reached an upper-bound of 6e06 nanoseconds.

References

[1] Thomas H. Cormen (2009) *Introduction to Algorithms, Third Edition*, The MIT Press

[2] Lecture notes

5 Appendix



```
                                  2-approx                              Nearest Neighbor                          Random Insertion
-----------------------------------------------------------------------------------------------------------------------------------------
Instance       |   Solution   |     Time     |  Error  |   Solution   |     Time     |  Error  |   Solution   |     Time     |  Error
---------------+--------------+--------------+---------+--------------+--------------+---------+--------------+--------------+---------
burma14.tsp    |     3835     |   156.514    |  0.096  |     4694     |    62.054    |  0.149  |     3683     |   135.281    |  0.027
ulysses16.tsp  |     7520     |   290.551    |  0.154  |     9465     |    84.492    |  0.167  |     7103     |   164.434    |  0.027
ulysses22.tsp  |     8148     |   344.582    |  0.162  |     8056     |   125.523    |  0.188  |     7416     |   305.635    |  0.036
eil51.tsp      |      604     |   2113.93    |  0.288  |      497     |   1010.89    |  0.22   |      448     |   1631.09    |  0.045
berlin52.tsp   |    10341     |   2684.67    |  0.303  |     8962     |   2025.71    |  0.235  |     7744     |   1735.3     |  0.052
kroA100.tsp    |    29715     |   9383.57    |  0.31   |    27772     |   6859.42    |  0.253  |    21854     |   6893.04    |  0.057
kroD100.tsp    |    27418     |   10124.2    |  0.321  |    26906     |   7434.7     |  0.264  |    22246     |   7232.31    |  0.068
ch150.tsp      |     8783     |   24108.4    |  0.345  |     8259     |   23243.5    |  0.265  |     7065     |   16369.7    |  0.082
gr202.tsp      |    52334     |    48459     |  0.371  |    52774     |   53630.5    |  0.305  |    42890     |    30972     |  0.083
gr229.tsp      |   176287     |   72138.4    |  0.379  |   168661     |   68024.4    |  0.314  |   148748     |   41322.1    |  0.105
pcb442.tsp     |    76723     |   363373     |  0.396  |    61926     |   485099     |  0.32   |    57394     |   175429     |  0.108
d493.tsp       |    46227     |   445816     |  0.418  |    43244     |   639248     |  0.38   |    37899     |   220190     |  0.117
dsj1000.tsp    | 2.57346e+07  | 2.93841e+06  |  0.511  | 2.46305e+07  | 6.04208e+06  |  0.413  | 2.08381e+07  | 1.04199e+06  |  0.13
```

Figure 5.1: TSP algorithms performance comparisons