Solution for Traveling Salesman Problem and analysis of error and time complexity

University of Padova
Department of Mathematics
"Tullio Levi-Civita"

Syed Riaz Raza[1] Rana Mandal[2] Muhammad Tabish[3]

*** Homework 2 Report for "Advanced Algorithms"**

Keywords

Advance Algorithms, Nearest Neighbor, Random Insertion 2Approximation, Constructive heuristics, Complexity, UNIPD, Padova, Italy

Abstract:

In this report we will compare the execution times and the quality of the solutions that can be obtained with different approximation algorithms.

1. **Nearest Neighbor Algorithm** (Constructive Heuristics**).**
2. **Random Insertion Algorithm** (Constructive Heuristics).
3. **2-approximate algorithm based on MST.**

The report summarizes the result in following way:

- Visualize the result using matplotlib one-by-one
- Comparing the result of one algorithm with other one and vice versa
- Conclude the result

[1] **Syed Riaz Raza; E-mail:** syedriaz.raza@studenti.unipd.it; **Portfolio**: riazraza.me
[2] **Rana Mandal; E-mail:** rana.mandal@studenti.unipd.it;
[3] **Muhammad Tabish; E-mail:** muhammad.tabish@studenti.unipd.it;

Table of Contents

1. Introduction:

a. Definition of TSP (Travelling Salesman Problem):

The traveling salesman problem (TSP) is an algorithmic problem tasked with finding the shortest route between a set of points and locations that must be visited. In the problem statement, the points are the cities a salesperson might visit. The salesman's goal is to keep both the travel costs and the distance traveled as low as possible.

b. Some notations for TSP are given:

Input: undirected, complete and weighted graph G = (V, E) with weights c : V × V 7→ N

Output: shortest Hamiltonian cycle in G

2. Project Structure:

A. Implementation:

We implemented most of our algorithms using adjacency matrix as an object in TSP class. (adjMatrix)

Other than the core data structures (for each algorithm), some functions are the same in all project files like:

Core Program Units:

- class TSP (diff for each algo)
- class Prim: (diff for each algo)
- class Two Approximation: (diff for each algo)
- class Nearest Neighbor: (diff for each algo)
- class Random Insertion: (diff for each algo)

Additional functions (Same for all algo implementations):

- funct load_all_dataset (Load All Dataset)
- funct populate_each_Graph_from_Dataset (Populate dataset to given Graph class)
- funct execute_each_graph_in_dataset (Compute and execute each dataset with their complexity and export the result into csv)
- funct main (main calling function)

B. Output Result:

The result created as output for each algorithm implementation is in .csv format, and all the project files implementing the algorithm are using the same format to export the data.

- Dataset Name
- Number of Nodes
- Weight (weight for each dataset)
- nano seconds time (ns time took to compute TSP for each graph)
- seconds time (s time took to compute TSP for each graph)
- exe times (num_time calculation done on each graph)

C. Asymptotic Notation & Visualization:

To compute Error Ratio and Asymptotic complexity for each algorithm the following parameters were created for each graph in dataset (for each algorithm implementation):

Note: Our work here to solve an intractable problem and to compare the execution times and Computing Ration and Constant:

Note: n here is the number of a graph in a dataset

- ratios= TSP [estimated_time][n+1] / TSP [estimated_time][n]
- constant= TSP [estimated_time] [n] / TSP [num_nodes][n]
- reference = avg(constant)* TSP [n][num_nodes])

We have created a separate file system which import the computational results of algorithm as mentioned in above paragraph and visualize the result of algorithms in following category

- Computational complexity of the algorithm
- Theoretical(C) computational complexity of the algorithm (reference variable)

And relative error calculated as:

$$\frac{ApproximateSolution - OptimalSolution}{OptimalSolution}$$

3. TSP Solution

A. Nearest Neighbor:

Nearest Neighbor Algorithm was one of the first algorithms used to determine a solution to the traveling salesman problem. In it, the salesman starts in a random city and repeatedly visits the nearest city until all have been visited. It quickly yields a short tour, but usually not the optimal one.

The nearest neighbor algorithm is easy to implement and executes quickly, but it can sometimes miss shorter routes which are easily noticed with human insight, due to its "greedy" nature. As a general guide, if the last few stages of the tour are comparable in length to the first stages, then the tour is reasonable; if they are much greater, then it is likely that there are much better tours.

Another check is to use an algorithm such as the lower bound algorithm to estimate if this tour is good enough. In the worst case, the algorithm results in a tour that is much longer than the optimal tour.

a. Working:

1. Initialize all vertices as unvisited.
2. Select an arbitrary vertex, set it as the current vertex **u**. Mark **u** as visited.
3. Find out the shortest edge connecting the current vertex **u** and an unvisited vertex **v**.
4. Set **v** as the current vertex **u**. Mark **v** as visited.
5. If all the vertices in the domain are visited, then terminate. Else, go to step 3

b. Pseudocode:

1. **Initialization:**

```
Take the first node
        * Add it to finalPath
        * Add its index to visited
    * Delete the node from the starting set
```

2. **Selection:** `Search for nodes`

```
  * Let (V1, ..., Vk) be the current path:
    * Take the vertex Vk + 1 not present & with minimum distance from Vk
```

3. **Insertion:** `* Insert Vk + 1 after Vk`

```
    * Update the weight value
    * Delete the node Vk + 1 from the starting set
```

4. repeat from (2) until all vertices are inserted in the path.

c. Complexity:

The time complexity of the nearest neighbor algorithm is O(n^2). The number of computations required will not grow faster than n^2.

d. Visualization

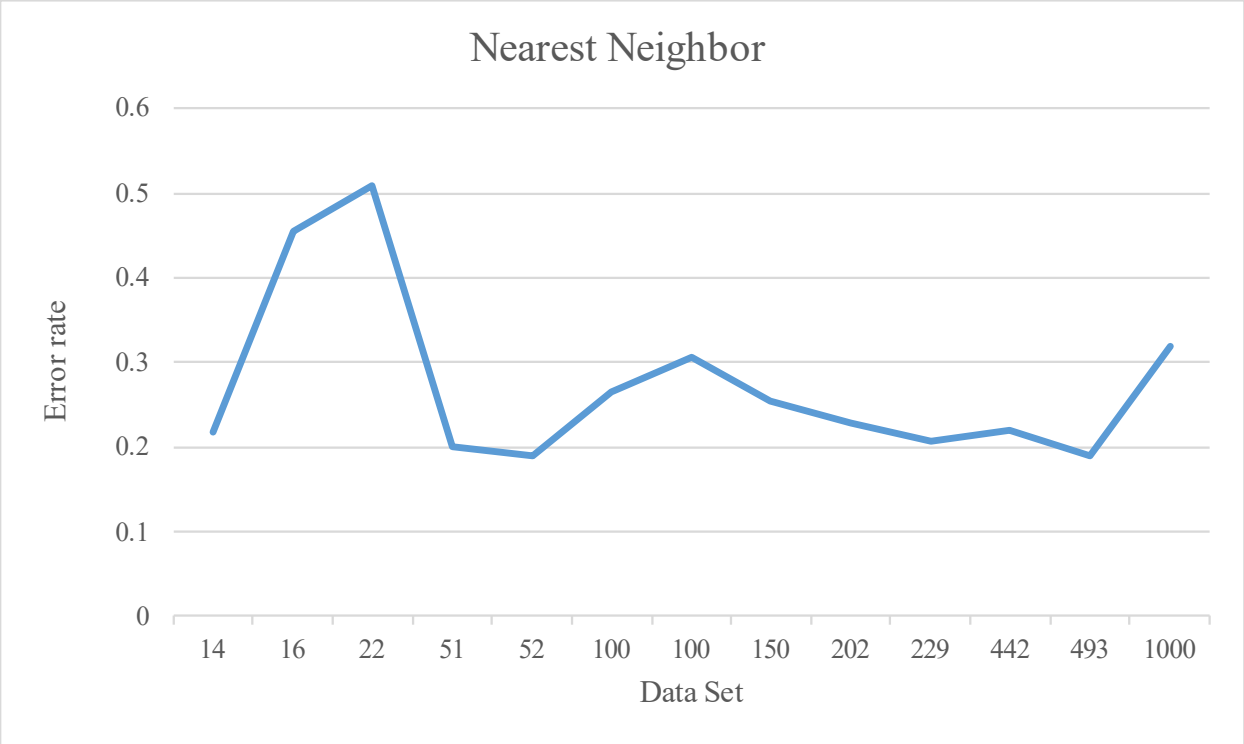

Figure 1: Nearest Neighbor error rate with for each dataset.

The graph just illustrated (fig. 1) shows that Error rate for Nearest Neighbor. As we can see the minimum error rate is near 0.2 and the maximum error rate is more than 0.5. And from the above shown graph we can see the average error rate is 0.2743154.

B. Random Insertion:

Random Insertion also begins with two cities. It then randomly selects a city not already in the tour and inserts it between two cities in the tour. Rinse, wash, repeat.

a. Pseudo Code:

1. Initialization: start from the single-node path 0. Find the vertex j that minimize w(0, j) and build the partial circuit (0, j);

2. Selection: randomly select a vertex k not in the circuit.

3. Insertion: find the edge {i, j} of the partial circuit that minimize w(i, k) + w(k, j) − w(i, j) and insert k between i and j;

4. repeat from (2) until all vertices are inserted in the path.

b. Complexity

Time complexity: O(n^2) Visualization:

c. Visualization:

Figure 2: Random Insertion error rate for each data set.

The graphs just illustrated (fig. 2,) show that error rate starting from 0(minimum) to near 0.16(maximum). and as we can see that the minimum error rate is 0 so we can say that is very efficient. And the average error rate is 0.099008.

C. Two Approximation Algorithm:

When the cost function satisfies the triangle inequality, we may design an approximate algorithm for the Travelling Salesman Problem that returns a tour whose cost is never more than twice the cost of an optimal tour. The idea is to use **Minimum Spanning Tree (MST)**.

1. Let 0 be the starting and ending point for salesman.

2. Construct Minimum Spanning Tree from with 0 as root using Prim's Algorithm.

3. List vertices visited in preorder walk/Depth First Search of the constructed MST and add source node at the end.

a. Why 2 Approximation:

1. The cost of best possible Travelling Salesman tour is never less than the cost of MST. (The definition of MST says it is a minimum cost tree that connects all vertices).
2. The total cost of full walk is at most twice the cost of MST (Every edge of MST is visited at-most twice)
3. The output of the above algorithm is less than the cost of full walk.

b. Prims Algorithm in brief:

Creating a set mstSet that keeps track of vertices already included in MST.

Assigning a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.

[The Loop] While mstSet doesn't include all vertices

- Pick a vertex u which is not there in mstSet and has minimum key value. (minimum_key())
- Include u to mstSet.
- Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v.

c. Approximation factor:

If the triangular inequality is respected: I Nearest Neighbor, Random Insertion and Farthest Insertion give a log(n)-approximation I

Closest Insertion and Cheapest Insertion find a 2-approssimation

d. Time complexities:

The time complexity for obtaining MST from the given graph is O(V^2) where V is the number of nodes. The worst case space complexity for the same is O(V^2), as we are constructing a vector<vector<int>> data structure to store the final MST.

e. Visualization:

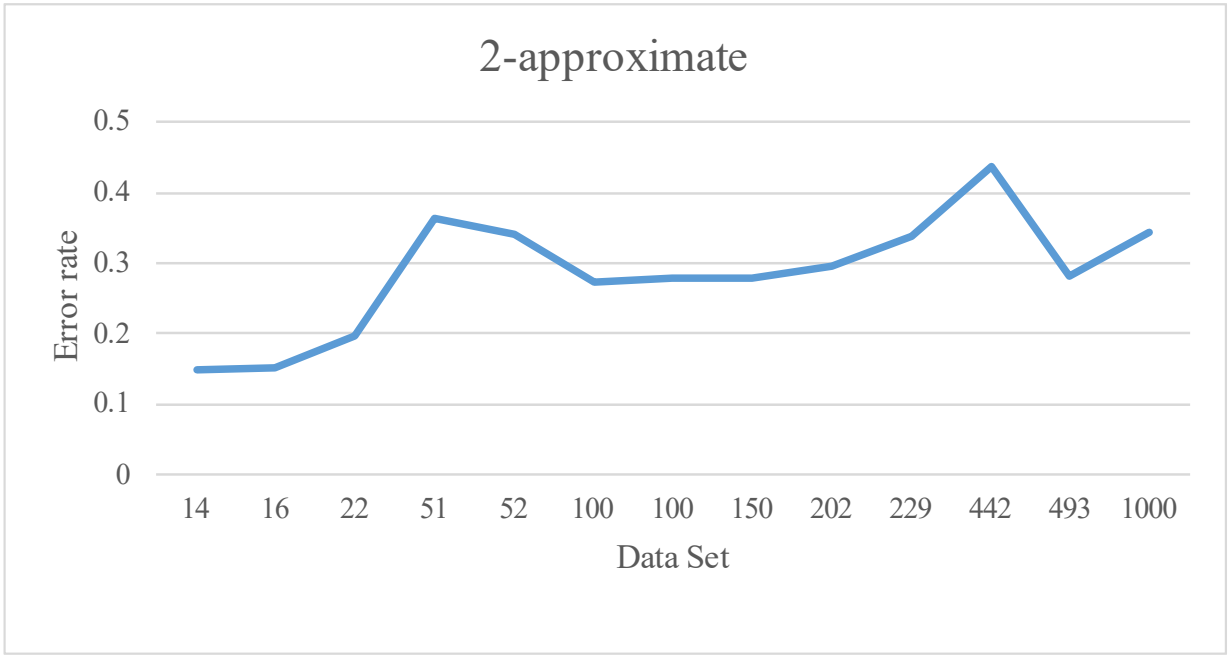

Figure 3: 2 Approximation algorithm error rate for each data set.

The graphs just illustrated (fig. 3) that the Error Rate is minimum 0.15 and maximum 0.4. And the average error rate is 0.286992.

4. Question 1:

Run the three algorithms (the two constructive heuristics and 2-approximate) on the 13 graphs of the dataset. Show your results in a table like the one below. The rows in the table correspond to the problem instances. The columns show, for each algorithm, the weight of the approximate solution, the execution time and the relative error calculated as:

$$\frac{ApproximateSolution - OptimalSolution}{OptimalSolution}$$

Instance	Nearest Neighbor			Random insertion			2 Approximation		
	Solution	Time(NS)	Error	Solution	Time(NS)	Error	Solution	Time(NS)	Error
burma14	4048	0.000305	0.2181	3336	0.000277	0.0039	3814	0.0005762	0.1477
ulysses16	9988	0.000357	0.4561	7108	0.0004177	0.0363	7903	0.000511	0.1522
ulysses22	10586	0.000576	0.5094	7499	0.0005553	0.0692	8401	0.0009334	0.1979
eil51	511	0.002442	0.1995	482	0.0022051	0.1314	581	0.004621	0.3638
berlin52	8980	0.002526	0.1906	8675	0.0023586	0.1502	10114	0.0042765	0.341
kroD100	26947	0.008881	0.2654	23280	0.010828	0.0932	27112	0.0170665	0.2732
kroA100	27807	0.012	0.3065	23534	0.0082532	0.1058	27210	0.0136536	0.2785
ch150	8191	0.019588	0.2547	7173	0.0174658	0.0988	8347	0.0391449	0.2786
gr202	49336	0.039137	0.2284	44094	0.0313031	0.0979	51990	0.0900801	0.2945
gr229	162430	0.055076	0.2067	147855	0.0404945	0.0984	180152	0.0919448	0.3384
pcb442	61979	0.232967	0.2205	58511	0.2341759	0.1522	73030	0.3178809	0.4382
d493	41660	0.287488	0.1902	39299	0.2342507	0.1227	44892	0.430566	0.2825
dsj1000	24630960	1.121442	0.32	21032671	0.9385765	0.1271	25086767	1.5908348	0.3444

5. Question 2

Comment on the results you have obtained: how do the algorithms behave with respect to the various instances? There is an algorithm that is always better than the others? Which of the three algorithms you have implemented is more efficient?

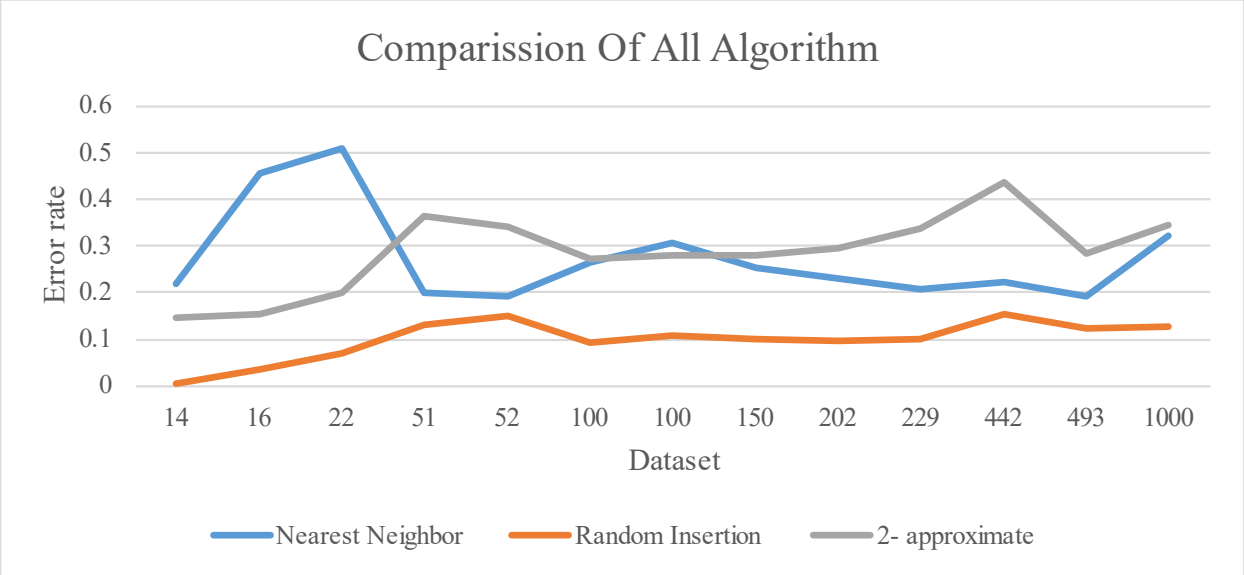

Figure 4: Comparison of the error rate between Nearest Neighbor, Random Insertion and 2-Approximation with execution of each graph.

The graphs just illustrated (fig. 4) show three comparisons between the Nearest Neighbor, Random Insertion and 2-Approximation algorithm on a linear scale

In this case it is possible to clearly see that Random Insertion algorithm turns out to be more efficient than Nearest Neighbor algorithm and 2-Approximation Algorithm. From this therefore we can conclude that the Random Insertion algorithm always tends to do better than the others.

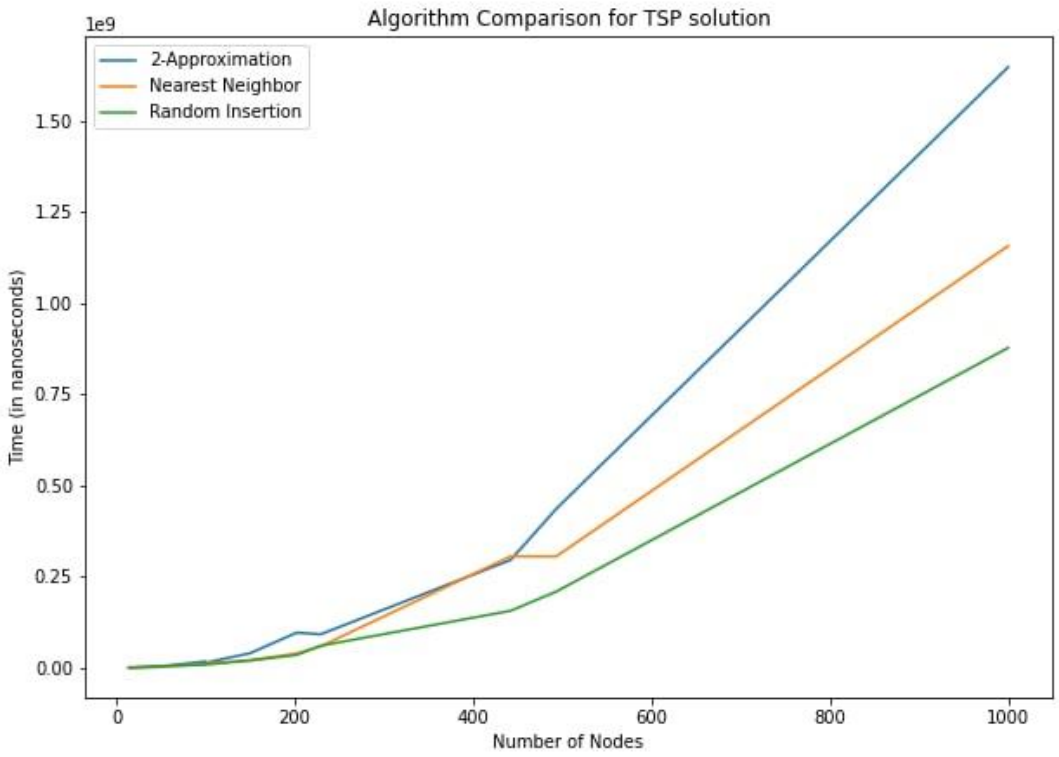

Figure 5: Comparison of three algorithm based on TSP solution with respect to number of nodes and time(ns).

The illustration (fig. 5) shows three comparisons between the Nearest Neighbor, Random Insertion and 2-Approximation algorithm on the basis on TSP solution. As shown above we can state that the time complexity (w.r.t increasing number of nodes) random insertion is more efficient than others two algorithms (Nearest Neighbor & 2 approximation).

As it can be shown from (fig:4) that the efficiency of the random algorithm is more than the nearest neighbor and 2 Approximation algorithm, the error rate is less in the random insertion as compared to the Nearest neighbor and 2 approximations.

6. Originality

- In this assignment we have optimized naive Random Insertion part where we must maintain partial circle with min cost**. We have used the working application of k-center algorithm (Facility location problems with outliers)** which uses 3 variables (partial_circle, adjMatrix, nodes).

- All algorithms including Nearest Neighbor, Random Insertion and 2 Approximation used 2D adjacency matrix.

- We created our own custom implementation of Priority Queue.

- We created one-for-all system to extract and execute dataset for different TSP solution. The TSP class can be used to execute other heuristic algorithm (with little to no changes).

7. Conclusion

The problems we faced during the work of our assignment:

- We ran into problem of Maintaining the partial circle in the Random Insertion because we reused the prims algorithm from 1st assignment and the error rate was greater than 1.5 so we had to reduce the error rate and tried to optimize the prims for TSP.

To conclude, what we have turned out for the algorithms is in line with what we expected, in fact:

- **Nearest Neighbor** is the highest one in error rate amongst three algorithms and the second efficient in terms of average time with increasing number of nodes.

- **Random Insertion** can be ranked first in all terms, as it is the fast and efficient in terms of average time with increasing number of nodes and one with least error rate amongst three algorithms.

- **2 Approximation Algorithm** can be ranked in second place with respect to error rate touching line with Nearest Neighbor, and the worst in average time with increasing number of nodes. The result of 2-Approx is ambiguous because maybe we can also use another MST algorithm which is more optimized like Kruskal with Union-Find

Concluding did good on our side, although our use of the Python language, although relatively simple, it was too slow as compared to working in Java. With java we could have made a quite good interactive application, and we wouldn’t have face typo error or library errors.

8. Dataset Result:

A. Nearest Neighbor

Dataset	# of Nodes	TSP result	Time (ns)	Time(s)	Execution Time
burma14	14	4048	291540.558	0.000292	2044
ulysses16	16	9988	383723.651	0.000384	1019
ulysses22	22	10586	642851.814	0.000643	1378
eil51	51	511	2991694.84	0.002992	426
berlin52	52	8980	3224192.07	0.003224	164
kroD100	100	26947	9094924.32	0.009095	111
kroA100	100	27807	12071702.7	0.012072	112
ch150	150	8191	20490159.6	0.02049	52
gr202	202	49336	39077350	0.039077	26
gr229	229	162430	58460488.2	0.058461	17
pcb442	442	61979	305381400	0.305381	4
d493	493	41660	305032250	0.305032	2
dsj1000	1000	24630960	1156798700	1.156799	1

B. Random Insertion:

Dataset	# of Nodes	TSP result	Time (ns)	Time(s)	Execution Time
burma14	14	3455	304361.3718	0.000304	1662
ulysses16	16	7495	330368	0.00033	1100
ulysses22	22	7559	572849.5787	0.000573	1424
eil51	51	469	2244229.106	0.002244	481
berlin52	52	8207	4099014.872	0.004099	195
kroD100	100	24439	8365454.478	0.008366	134
kroA100	100	22537	8375222.047	0.008375	127
ch150	150	7259	19744541.51	0.019745	53
gr202	202	45008	34844210	0.034844	30
gr229	229	146713	60194130.43	0.060194	23
pcb442	442	57567	156046850	0.156047	4
d493	493	38535	208495500	0.208496	4
dsj1000	1000	20975443	877430200	0.87743	1

C. 2 Approximation:

Dataset	# Of Nodes	TSP result	Time (ns)	Time (s)	Execution Time
burma14	14	3814	480670.989	0.000481	1820
ulysses16	16	7903	580390.49	0.00058	1041
ulysses22	22	8401	908055.417	0.000908	480
eil51	51	581	5293788.44	0.005294	199
berlin52	52	10114	4389928.46	0.00439	253
kroD100	100	27112	17289541.5	0.01729	53
kroA100	100	27210	13710138.8	0.01371	80
ch150	150	8347	39826250	0.039826	26
gr202	202	51990	96071507.1	0.096072	14
gr229	229	180152	91616663.6	0.091617	11
pcb442	442	73030	295390867	0.295391	3
d493	493	44892	435337900	0.435338	2
dsj1000	1000	25086767	1647512900	1.647513	1

EOF