UNIVERSITY OF PADOVA

DEPARTMENT OF MATHEMATICS

ADVANCED ALGORITHMS

COMPARISON OF RANDOM INSERTION, CHEAPEST INSERTION AND MST-BASED APPROXIMATE ALGORITHMS IN METRIC TSP

MOHAMMADMAHDI GHAHRAMANI 2041608

BILGE ZERRE 2041585

MARGARITA KOSAREVA 2041604

Academic Year

2021 - 2022

Abstract

Research Focus: Implement three approximation algorithms, Random Insertion, Cheapest Insertion and MST-based algorithms, for a specific kind of Traveling Salesman Problem (TSP), metric TSP in which the triangular inequality is true on all distances(weights), and compare it to the exact algorithm

This report has scrutinized the implementation and analysis of three algorithms to approximate TSP optimal solution. This study considers two different approach to perform mentioned task. The first one is to use two algorithm from a family of constructive heuristics that uses a greedy approach to find a solution, with different guarantees on the quality of the approximation. They are Random Insertion (log(n)-approximate) and Cheapest Insertion (2-approximate) algorithms. The second approach is to use a 2-approximate algorithm based on MST, that guarantees a good approximation for metric TSP. The data set consists of 13 graphs, ranging in the size of vertices from 14 to 1000. The important note regarding graphs is the fact that the triangular inequality is true on all distances(weights) in each graph.

The final inferences to analyze and compare the algorithms are made by implementing and testing Random Insertion, Cheapest Insertion, and MST-based algorithms on the same dataset by using Python as a programming language and using Google Colab as an environment.

Keywords: Approximation algorithms, MST, Metric TSP, Random Insertion algorithm, Cheapest Insertion algorithm

Table of Contents

CHAPTER 1

Introduction

Traveling Salesman Problem (TSP) is a well-known NP-hard problem in the field of graph theory. Given an undirected, complete and weighted graph G = (V, E) with weights $w : V \times V \rightarrow N$, TSP asks us to find the shortest Hamiltonian cycle in G. In graph theory, given a graph, a cycle is Hamiltonian if it touches all nodes exactly once, except the node that the cycle starts with, and it has the minimum total weight comparing to any other cycle with a such characteristic. Although TSP can be defined for any undirected weighted graph, generally the graph is assumed to be a complete graph. As a short introduction to different kind of problem difficulties, intuitively and informally, a problem is in NP if it is easy to verify its solutions. On the other hand, a problem is NP-hard if it is difficult to solve, or find a solution. Now, a problem is NP-complete if it is both in NP, and NP-hard. Therefore you have two key, intuitive properties to NP-completeness. Easy to verify, but hard to find solutions. TSP is one of the famous NP-hard problems. It is not NP-complete since it is not in NP. In other words, it is not easy to verify a solution in TSP. In order to verify a candidate solution, we need check all Hamiltonian cycles (there are (n-1)!/2 Hamiltonian cycles in a complete graph, where n is the number of vertices) in the graph and compute the corresponding total weight and pick the minimum one and compare it with the candidate solution. It requires exponential time to carry out and cannot get done by a polynomial time algorithm. That is why TSP is NP-hard and not NP-complete. According to these definitions, we can define the *metric TSP* as a general TSP problem in which the triangular inequality holds for all distances in the graph.

When it comes to NP-hard problem, approximation algorithms are introduced. There are several algorithms to estimate the optimal solution of a metric TSP. This study considers two constructive heuristics and a MST-based 2-approximate algorithm. Constructive heuristics is a large family of heuristics that build the solution one vertex at a time according to the general scheme of *Initialization*(how to chose the initial circuit), *Selection*(how to chose the next vertex to be inserted in the solution) and *Insertion*(how to chose the position where to insert the new vertex). Nearest Neighbor, Random Insertion, Cheapest Insertion, Circuit-vertex distance, Closest Insertion and Farthest Insertion algorithms are in this category. However, this paper only reviews two of them, including Random Insertion and Cheapest Insertion. Defining these three algorithms requires a previous knowledge of basic concepts such as PRIM's algorithm and this research does not go through reviewing these concepts.

1. Random Insertion Algorithm

Random Insertion is the first constructive heuristic that this study reviews. As mentioned before, constructive heuristics have a general scheme of Initialization, Selection and Insertion. In the first step, Initialization, the algorithm starts from the single-node path 0, finds the vertex j that minimize w(0, j) and build the partial circuit (0, j). In the Selection step, it randomly selects a vertex k not in the circuit. Finally, it finds 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 in the Insertion step. The algorithm then repeats from the Selection step until all vertices are inserted in the path.

Regarding the complexity of this algorithm, it is expected to see a fast execution since it randomly selects a vertex in the Selection step and does check any other measurements. As for approximating factor, this algorithm is a log(n)-approximation algorithm whose error ($\frac{approximatedsolution-optimalsolution}{optimalsolution}$) is upper bounded by the values of *log(n)-1* where *n* is the number of nodes.

2. Cheapest Insertion Algorithm

Cheapest Insertion is the second constructive heuristic that this study reviews. In the first step, Initialization, this algorithm starts from the single-node path 0, finds the vertex j that minimize w(0, j) and build the partial circuit (0, j). In the Selection step, it finds a vertex k not in the circuit and an edge {i, j} of the circuit that minimize w(i, k) + w(k, j) - w(i, j). Finally, it inserts k between i and j in the Insertion step. The algorithm then repeats from the Selection step until all vertices are inserted in the path.

Regarding the complexity of this algorithm, it is expected to see a slower execution compared to Random Insertion since it does not randomly selects a vertex in the Selection step and iterates in all possible k and all possible candidate edge {i, j}. That is why it is expected to be more time demanding. As for approximating factor, this algorithm is a 2-approximation algorithm whose error ($\frac{approximatedsolution-optimalsolution}{optimalsolution}$) is upper bounded by the value of 1.00.

3. MST-based Algorithm

The intuition of approximating *Vector Cover* task with *Maximal Matching* approach constructs the idea of using MSTs to approximate a metric TSP task. It first obtain the MST of a given complete graph, because the MST is originally a tree (note that complete graphs are connected graphs and it is not needed to talk about forest) which guarantees the fact that each node is touched no more than once. However, each solution of a TSP tasks must be a cycle. In order to turn the obtained tree (MST) to a cycle, this algorithm uses *PREORDER* algorithm to traverse the graph and then adds the first node at the end of the result to get a cycle. PREORDER algorithm is another way of visiting the graph which uses a recursive implementation starting from an arbitrary node and printing that. It then

checks whether the node is internal. If so, it goes in its children and call again the PREORDER on the children. Using these approach we get a Hamiltonian cycle which is a cycle (using PREORDER part) and expected to provide a good approximation of the optimal solution (using MST part).

Regarding the complexity of this algorithm, exprience shows that this method is faster than Cheapest Insertion and slower than Random Insertion algorithm. As for approximating factor, this algorithm is a 2-approximation algorithm whose error ($\frac{approximated solution - optimal solution}{optimal solution}$) is upper bounded by the value of 1.00.

In order to obtain the MST, this study uses PRIM's algorithm which uses the data structure of HEAP. It is because PRIM's algorithm is suitable for dense graphs while Efficient Kruskal's algorithm is suitable for sparse graphs. Since we are dealing with complete graphs, which are totally dense, the preferred algorithm is PRIM's algorithm.

4. Implementation Choices

This study uses Python programming language and Google Colab environment to implement these algorithms. It does not use any built-in Python library to directly implement the algorithms. It uses six libraries including ***time***, ***os***, ***random numpy***, ***pandas*** and ***matplotlib*** for loading the data, implementing Random Insertion algorithm, measuring execution time and result visualization.

This paper considers one Python classes, named ***TSP_Solver***. This class contains the implementation of three previous stated algorithms. Random Insertion and Cheapest Insertion are not using any private methods within the class, while MST-based algorithm utilizes two private methods, named ***PRIM*** and ***preorder***. The class has also another method named ***SanityCheck*** whose responsibility is to check if the produced cycles by methods are Hamiltonian cycles. The implementation of Random and Cheapest Insertion are exactly according to the provided description in the previous part and there is no prominent note to mention regarding these algorithms. About the MST-based algorithm, it first calls the *PRIM* method which is implemented using HEAP data structure and then uses *preorder* method to traverse the obtained MST. This is important that the *preorder* method starts with the exact same node that the *PRIM* method starts building the MST. Finally, it appends the starting point to the produced path to ca create cycle. One more note concerning *preorder* method is that, when visiting the tree, it simultaneously computes the cumulative weight to avoid extra unnecessary for loop. That is how ***self.total_weight_two_approx*** attribute gets updated.

The way this study represent the graphs is by their ***Adjacency Matrix***. Each element of this matrix is the distance between two corresponding nodes of the graph. Since the input graphs are complete, this matrix is supposed to have n(n-1) elements. The distances are symmetric, requiring the matrix to have G[i][j] = G[j][i], where G is the adjacency matrix of a given graph. It is also hold that G[i][i] = 0 for all nodes i, since we do not consider self loops.

The input graphs are not well constructed, meaning that instead of nodes and edges, we are given a bunch of coordinates which can be imagined as the location of a specific node in the space. So, given a set of N coordinates, the code computes $\frac{N(N-1)}{2}$ weights (weights are symmetric) and define the adjacency matrix. Each graphs uses its own distance measurement, being GEO or EUC distance measurements. Each measurement has its own formula to compute distances. However, regardless of the distance measurements, the produce graph does not violate triangular inequality. It is important, because the mentioned algorithms have been invented to deal with a graph on which the triangular inequality is held. After the first part, ***TSP_Solver*** class, the code considers a function named ***graph-Constructor*** which take advantage of two other functions named, ***GEO*** and ***EUC*** to produce the adjacency matrix of graphs, depending on their distance measurements. Worth mentioning that each set of coordinates is firstly converted to a list whose first element contains meta data about the data and the rest of elements contains adjacency matrix of the graph. When this data is fed to the Python class, it splits the meta data from adjacency matrix. It then uses meta data to print out some general information about the graph and uses adjacency matrix to run algorithms. Figure1 is an illustration of a graph ready to be fed into our Python class.

```
[{'distance_type': 'GEO', 'name': 'burma14', 'number_of_nodes': 14},
 [0, 153, 510, 706, 966, 581, 455, 70, 160, 372, 157, 567, 342, 398],
 [153, 0, 422, 664, 997, 598, 507, 197, 311, 479, 310, 581, 417, 376],
 [510, 422, 0, 289, 744, 390, 437, 491, 645, 880, 618, 374, 455, 211],
 [706, 664, 289, 0, 491, 265, 410, 664, 804, 1070, 768, 259, 499, 310],
 [966, 997, 744, 491, 0, 400, 514, 902, 990, 1261, 947, 418, 635, 636],
 [581, 598, 390, 265, 400, 0, 168, 522, 634, 910, 593, 19, 284, 239],
 [455, 507, 437, 410, 514, 168, 0, 389, 482, 757, 439, 163, 124, 232],
 [70, 197, 491, 664, 902, 522, 389, 0, 154, 406, 133, 508, 273, 355],
 [160, 311, 645, 804, 990, 634, 482, 154, 0, 276, 43, 623, 358, 498],
 [372, 479, 880, 1070, 1261, 910, 757, 406, 276, 0, 318, 898, 633, 761],
 [157, 310, 618, 768, 947, 593, 439, 133, 43, 318, 0, 582, 315, 464],
 [567, 581, 374, 259, 418, 19, 163, 508, 623, 898, 582, 0, 275, 221],
 [342, 417, 455, 499, 635, 284, 124, 273, 358, 633, 315, 275, 0, 247],
 [398, 376, 211, 310, 636, 239, 232, 355, 498, 761, 464, 221, 247, 0]]
```

FIGURE 1. **Example of a given graph to the Python class, including 14 vertices.** As mentioned, the first element contains the meta data about the data and the rest of elements contain the adjacency matrix corresponding to the data.

CHAPTER 2

Analysis

In this section, we will run and analyze the algorithms by answering to two questions. Moreover, we will implement an algorithm to obtain the optimal solution for TSP tasks by iterating in all possible Hamiltonian cycles. Then in order to show how powerful the approximation algorithms are, we compare the result of our algorithms with the optimal solution on small graphs (up to 8 nodes). A complete graph with N nodes has $\frac{(N-1)!}{2}$ Hamiltonian cycles. That is why this part runs on small graphs (even a small complete graph of 10 nodes has nearly 180,000 Hamiltonian cycles). This extra part is suggested by this research in the ***Originality*** section.

1. Question 1

In this part we analyze the performance of the algorithms by running them and reporting a table in which the ***solution***, ***execution time*** and ***error*** of each algorithm on input graphs are reported.

Figure1 is a table whose rows 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 $\frac{approximatedsolution-optimalsolution}{optimalsolution}$.

	RandomInsertion			CheapestInsertion			2-Approx		
	Solution	Time	Error	Solution	Time	Error	Solution	Time	Error
burma14.tsp	3437.3	0.000077	0.034397	3568.0	0.000216	0.073729	4062.3	0.000219	0.22248
ulysses16.tsp	7152.4	0.000109	0.042776	7368.0	0.000336	0.074209	7929.2	0.000344	0.156029
ulysses22.tsp	7245.9	0.000164	0.03321	7709.0	0.000822	0.099244	8344.7	0.000673	0.18989
eil51.tsp	461.6	0.000707	0.083568	494.0	0.009594	0.159624	605.4	0.00661	0.421127
berlin52.tsp	8434.2	0.000737	0.118298	9004.0	0.010484	0.193848	10246.4	0.006976	0.358579
kroD100.tsp	23126.5	0.0026	0.086057	25204.0	0.079961	0.18362	28866.6	0.046857	0.355621
kroA100.tsp	23208.7	0.002877	0.090532	24942.0	0.076991	0.171976	29299.9	0.047881	0.376746
ch150.tsp	7281.9	0.005821	0.115487	8051.0	0.259947	0.233303	9120.6	0.15313	0.397151
gr202.tsp	44562.0	0.016101	0.109612	46480.0	0.675727	0.157371	51747.4	0.370409	0.288531
gr229.tsp	147589.8	0.015854	0.09649	154038.0	0.972837	0.144396	179632.5	0.543779	0.334546
pcb442.tsp	58397.2	0.065217	0.150049	60834.0	8.037369	0.198039	70686.6	4.309475	0.392071
d493.tsp	38598.7	0.078549	0.102757	39969.0	11.013914	0.141906	45792.0	6.336664	0.308268
dsj1000.tsp	21008958.5	0.372786	0.125901	22291165.0	102.550845	0.194616	25493529.8	55.552365	0.366236

FIGURE 1. **Filled table for question 1**

It is vital to consider the fact that Random Insertion algorithm, depending on the sequence of generated numbers, and MST-based algorithm, depending on starting point, might produce the different solutions. It is unreal to use just one output of these two algorithms to do analysis. Instead

we ran these models several times and used the expected value of solutions. This approach is also used to calculate the expected value of the execution time for each algorithm. The code defines a variable named ***numberOfExecutionPerSample*** that determines how many executions we consider to calculate the expected values of solution and expected value of execution time for each algorithm and each input graph. This expected value is computed by means of a simple averaging over solutions and execution times. Note that the Cheapest Insertion algorithm does not use any randomization or does not use the starting point, meaning that the solution for a given graph is always the same and the concept of expected value might not sense that much for this algorithm.

2. Question 2

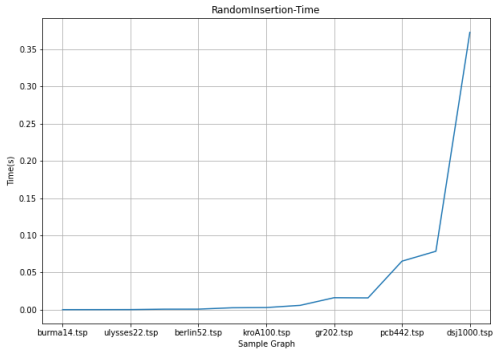
In this section we first comment on the obtained result and then provide several illustrations to visually show our interpretation. Approximation algorithms are essentially introduced to reduce the time complexity of the original problem and avoid dealing with problems that require exponentially time to execute. But this reduction is at the expense of losing accuracy and estimate the optimal solution. Hence, two fundamental aspects of the approximation algorithms that must be studied is their execution time and their produced error.

Regarding the execution time, we expect all algorithm not to be slow, because they have been introduced with the aim of time complexity reduction. So, they are supposed to run on all 13 graphs with no time concern. Albeit, they will differ in execution time compared to each other. Figure2 shows that our three algorithms run on the biggest input graph with no time issue.

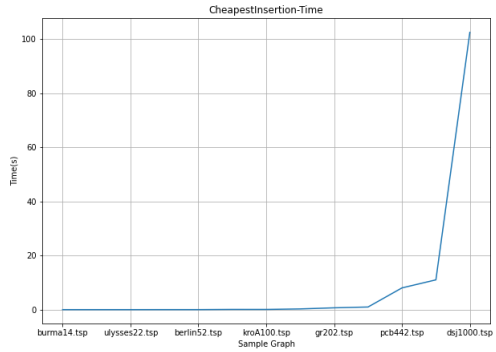
```
--------------------------------------------------
Graph: dsj1000

The result of RandomInsertion algorithm: 20919210
The result of CheapestInsertion algorithm: 22291165
The result of 2-approx algorithm: 25526005
--------------------------------------------------
```

FIGURE 2. **Approximated solution produced by each algorithm on *dsj1000* graph.**

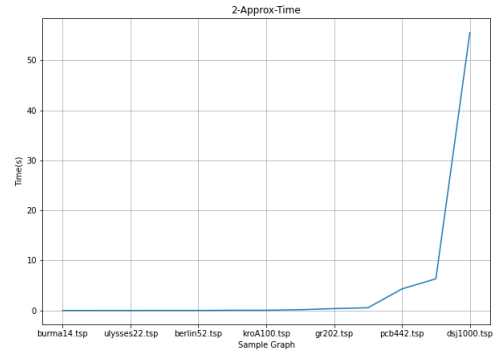
In order to show how the algorithms behave against various instances, with respect to execution time, we plot the execution time of each algorithm on all 13 graphs. Figure3 contains three plots, each indicating how time execution changes when the size of the graph increases. These plots show two important thing. The first one is that by increasing the size of the graph (increasing the number of nodes), the execution time increases. The second thing is that, the general scheme of the plots are same as each other. It is an evidence that these algorithms are almost of the same order of polynomial time complexity. This is intuitive since the main responsibility of these algorithm is to reduce the exponential time complexity to a polynomial time complexity. We computed the polynomial order of each algorithm and plot it on Figure4. According to our small dataset, Random Insertion algorithm



(A) Random Insertion



(B) Cheapest Insertion

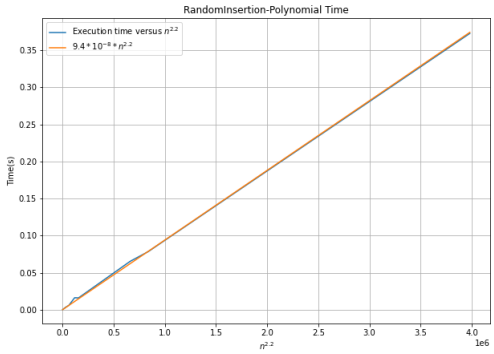


(C) MST-based

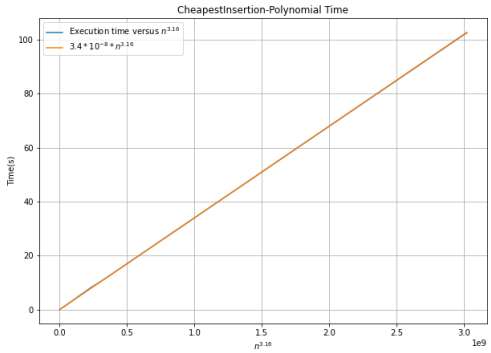
FIGURE 3. **Algorithms' execution time against various instances**

is from a complexity of $O(n^{2.2})$ with a constant factor of $9.4 * 10^{-8}$. Cheapest Insertion algorithm is from a complexity of $O(n^{3.16})$ with a constant factor of $3.4 * 10^{-8}$. MST-based algorithm is from a complexity of $O(n^{3.00})$ with a constant factor of $5.6 * 10^{-8}$.

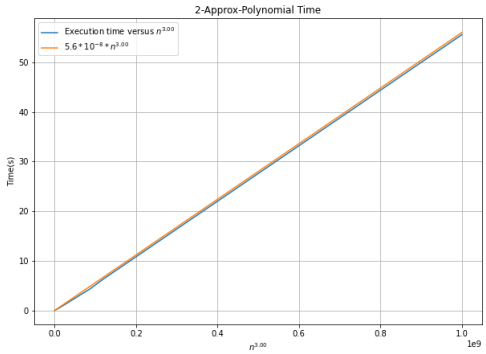
The next analysis that this study considers, is to compare the algorithm's time execution with each other. As discussed before, it is anticipated that the Random Insertion algorithm finds the solution much quicker. It is also expected that Cheapest Insertion algorithm has the slowest running time since it iterates in all possible nodes and all possible edges in the circuit. Figure5 shows the fact that, the quickest algorithm is Random Insertion. It takes only 0.37 second on average to get executed on the



(A) Random Insertion



(B) Cheapest Insertion



(C) MST-based

FIGURE 4. **Algorithms' polynomial time complexity**

biggest graph with 1000 nodes and it is very fast. Cheapest Insertion algorithm, on the other hand, is the slowest algorithm. It takes almost 102 second on average to get executed on the biggest graph with 1000 nodes. This number is approximately 55 second for the MST-based algorithm.

If we consider a strong link between algorithm efficiency and algorithm execution time, according to the results, we can claim that Random Insertion is the most efficient algorithm compared to each other. MST-based algorithm is the second most efficient and Cheapest Insertion is at the last ranking, with respect to efficiency.

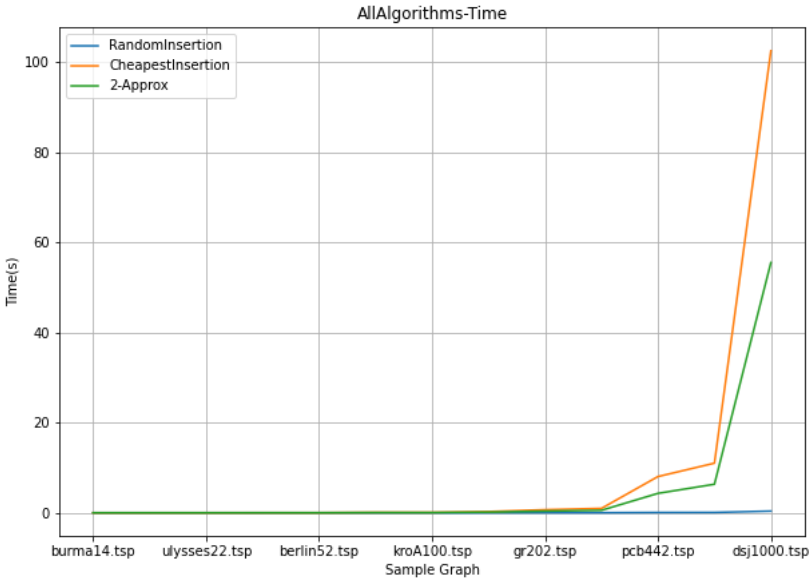


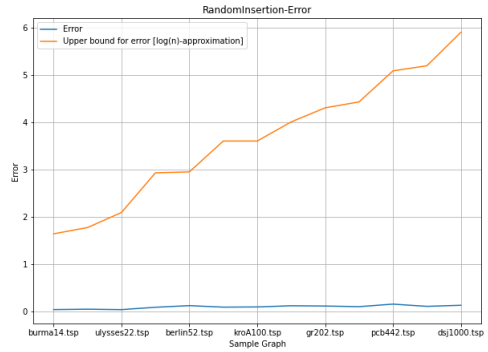
FIGURE 5. **Algorithms' execution time compared to each other**

In order to show how the algorithms behave against various instances, with respect to the produced error, we plot the error of each algorithm on all 13 graphs. Figure6 contains three plots, each indicating how the produced error changes when the size of the graph increases. It also plots the upper bound for the error for each algorithm. This value is 1.00 for Cheapest Insertion and MST-based algorithm and $log(n) - 1$ for Random Insertion.

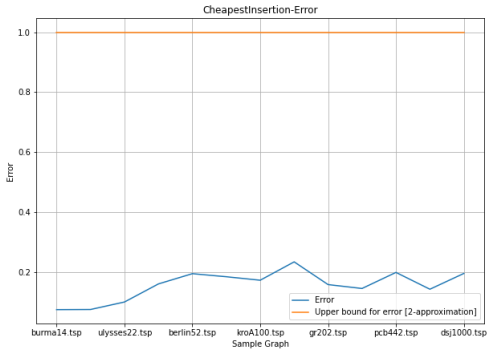
There are two important note about the plotted figures. The first one is that, all algorithms respect the upper bound for their error and it can be seen as another evidence that the algorithms have been implemented correctly. The second and more important one is that there is no correlation between the produced error and graph size. All algorithms' errors fluctuate widely with respect to various instances and there is no strong link between the error and the size of the graph.

The last analysis in this part, is to compare the algorithm's error with each other. Although, the error fluctuation along different instances does not follow a specific rule, but it might be expected that their behaviours are comparable from one algorithm to another. Figure7 shows that, in spite of wild fluctuation, Random Insertion algorithm is more accurate compared to other algorithms and MST-based algorithm is less accurate compared to other algorithms. So we can claim that Random Insertion is the most accurate algorithm, Cheapest Insertion is the second most accurate algorithm and MST-based is at the last ranking, with respect to approximation error.

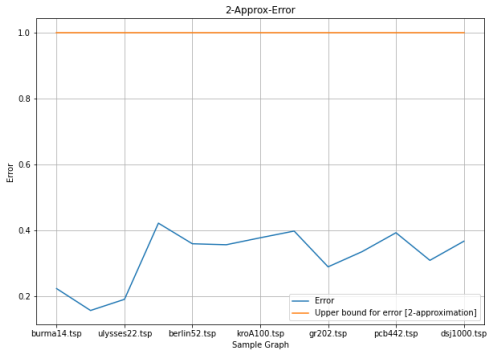
To sum up this part, Random Insertion algorithm is the most powerful algorithm with respect to both execution time and accuracy factors. Cheapest Insertion is the second best with respect to approximation error and the worst one with respect to execution time. Finally, MST-based is the best second algorithm with respect to execution time and the worst one with respect to approximation



(A) Random Insertion



(B) Cheapest Insertion



(C) MST-based

FIGURE 6. **Algorithms' approximation error against various instances**

error. Hence, **according to the small dataset provided by this study**, this part recommends us to use Random Insertion algorithm for a new graph, since it has the best performance with respect to two stated factors, execution time and approximation error. Figure8 shows this fact visually, by plotting the execution time and approximation error of the three algorithms on a randomly chosen input graph.

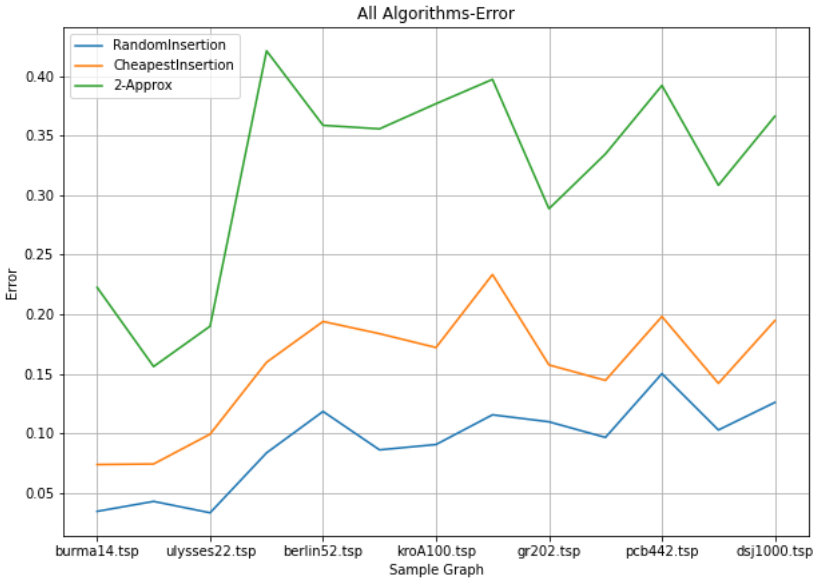
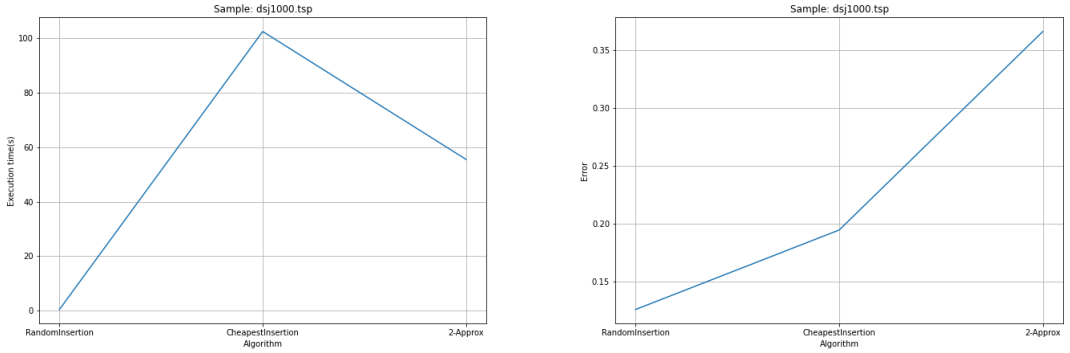


FIGURE 7. **Algorithms' approximation error compared to each other**



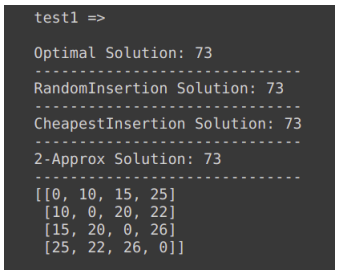
(A) Execution Time (B) Approximation Error

FIGURE 8. **Algorithms' execution time and approximation error shown for *dsj1000* graph.**

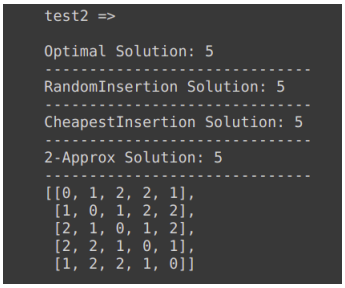
3. Originality

All mentioned algorithms are approximation algorithms that tries to estimate the optimal solution. The authors detected the fact that most of the time, the approximated solution is a very good estimation of the optimal solution. It shows how powerful these algorithms can get. This observation, made us to write a code whose responsibility is to generate all Hamiltonian cycles and then iterate on all of them (***Cycle_iterator*** function) and select the minimum-weighted one as the optimal solution (***TSP_Solver_Exact*** function). We then observed that the result of approximation algorithms are the same exact as optimal solution. We then concluded that on small graphs, there is a high probability that approximation algorithms hit the optimal solution. When generating the graphs, we took care of graphs to obey triangular inequality. Figure9 contains several graphs on which we performed

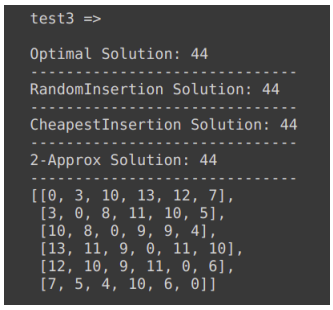
the proposed procedure. The first four graphs hold the triangular inequality and the approximated results hit the optimal solution. The last graph is a randomly generated graph on which the triangular inequality does not hold. That is why algorithm's error might be higher than their upper bound. They have all been constructed for metric TSP task and if the graph is not metric, they might produced a higher error than their error upper bounds.



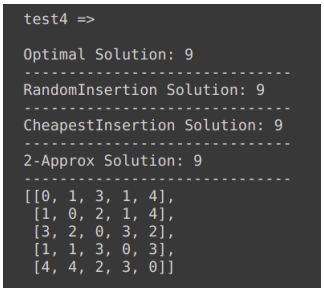
(A) Test Case 1



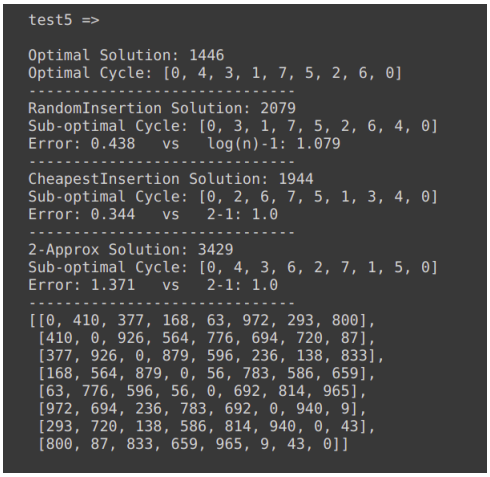
(B) Test Case 2



(C) Test Case 3



(D) Test Case 4



(E) Test Case 5

FIGURE 9. **Optimal solution versus approximated solutions on small graphs.** The figure shows if the size of the graph is small, there is a high probability of hitting the optimal solution by approximation algorithms. It also shows the fact that if triangular inequality is violated in a graph, approximation algorithms would produce a higher error compared to their error upper bound. In Test Case 5, the solution produced by MST-based 2-Approx algorithm is more than twice of the optimal solution.

CHAPTER 3

Conclusion

In this research, we implemented and reviewed three algorithms to approximate metric TSP tasks. These algorithms are Random Insertion and Cheapest Insertion from constructive heuristics family as well as 2-Approximate algorithm based on MST. We ran these algorithms and then measured their execution time and approximation error. Given a dataset of 13 graphs, we tested all three algorithms and concluded that the Random Insertion algorithm is always the best algorithm, with respect to execution time and approximation error. We then proposed an iterative algorithm to find all Hamiltonian cycles and find the optimal solution. The obtained result showed that for small graphs, the approximated solutions mostly hit the optimal solution and it shows the power of approximation algorithms. We finally observed that if the graph is not metric, approximation algorithms may produce an error which exceeds its upper bound.

Results obtained by this study endorses the fact that by accepting a trade of between time complexity and accuracy of obtained solution, we can suggest algorithms that are feasible to execute in the polynomial time and can provide a good estimation of the optimal solution. The authors believe that in order to analyze the performance of an algorithm that uses randomization or other procedures which results in creating a different solution on each execution, it is more reasonable to compute the expected value of the result and compare algorithms on average. It is a more statistical and highly-used approach even in other domains. As an instance, in Machine Learning field, since the performance of methods occasionally depend on their initial configuration and this configuration may change from one execution to another, it is common to consider Cross Validation technique which runs the method several times and report the average accuracy or average loss. This is similar to what this paper did to compute the solutions and errors of Random Insertion and MST-based algorithms. By increasing the number of executions, we will obtain a more accurate results which are the better representation of algorithms' performance and it can be seen as a possible improvement to what this study performed.