Theoretical and Empirical Analysis and Comparison of NP-Complete Problems and their Approximation Algorithms

Independent Study Report

To,

Dr. Tom Bylander

By

Venkat Rahul Dantuluri (fpj626)

Submitted on

10 August, 2015

Introduction

This report contains details of theoretical and empirical analysis done on two famous NP-Complete problems namely, Travelling Salesman problem and Bin Packing problem. This report does not include the reasons for the mentioned problems to be NP-Complete, but taking into consideration that they are, we compare the running times and costs of their optimization algorithms with their corresponding approximation algorithms.

It is very important to understand the certain terminology that has been used in this report.

NP-Hard – Class of NP (non-deterministic) problems that are harder than any other NP problem. [1]

NP-Complete - In computational complexity theory, a decision problem is NP-complete when it is both in NP and NP-hard. [4]

Optimization Problem - In mathematics and computer science, an optimization problem is the problem of finding the best solution from all feasible solutions. [2]

Approximation Algorithm - In computer science and operations research, approximation algorithms are algorithms used to find approximate solutions to optimization problems. Approximation algorithms are often associated with NP-hard problems; since it is unlikely that there can ever be efficient polynomial-time exact algorithms solving NP-hard problems, one settles for polynomial-time sub-optimal solutions. [3]

Note: I use the terms costs, weights interchangeably carrying the same meaning.

Travelling Salesman Problem

The problem statement of this problem states that, given an undirected graph G = (V, E) we need to visit all the vertices of the graph without visiting any particular vertex more than once. And this visit should be made with least cost possible. In technical terms we need to find the shortest Hamiltonian cycle possible. This problem is NP-Complete as its solution relies on finding the solution to the Hamiltonian cycle problem. The decision problem of finding the existence or non-existence of a Hamiltonian cycle is itself an NP-Complete problem.

I coded the optimal and approximation algorithms to Travelling Salesman problem in an attempt to perform an empirical study to prove that the approximation algorithm stated by Thomas H Cormen et al in their Introduction to Algorithms book is a 2-approximation algorithm.

The Approximation algorithm as can be inferred from the program submitted or can be referred from the book chapter 35, deals with computing a minimum spanning tree from

graph G and then performing a preorder tree walk. The resultant printed vertices form a Hamiltonian cycle with cost of traversal less than 2 times that of an optimal solution. Hence 2-approximation algorithm. Let us first have a theoretical discussion as to why this particular approximation algorithm is always 2-approximation algorithm.

Theoretical Discussion

H* be the Hamiltonian cycle which corresponds to the optimal solution to Travelling Salesman problem. Let T depict the minimum spanning tree that has been computed from the graph G. Since T consists of one less edge than H*

Therefore, the cost of T c(T) is less than cost of H* c(H*). Hence we have a lower bound on the cost of optimal solution to travelling salesman problem.

$c(T) \leq c(H^*)$

Now when we perform a tree-walk W on the minimum spanning tree T then we traverse each node exactly twice.

Hence $c(W) = 2c(T)$.

Note: It is very important to note that we are using only positive costs for edges in a graph. Hence the edges hold the triangle property of inequality. I.e. if u, v, w are vertices belonging to the same graph then $c(u,w) \leq c(u,v) + c(v,w)$

We then traverse through W and remove any vertex that has been already traversed and move it to a new set H.

Hence $c(H) \leq 2c(H^*)$

Hence our approximation algorithm is a 2-approximation algorithm.

Empirical Analysis

I coded the approximate algorithm and optimal algorithms in JAVA language (JRE 7 runtime). The program first takes a set of vertices as input and generates a "cost matrix". A cost matrix is a 2-dimentional array containing the costs of direct traversal from a source vertex to any destination vertex. In graph theory terminology it depicts the adjacency matrix.

The code is provided along with this report in separate source file for immediate compilation and execution.

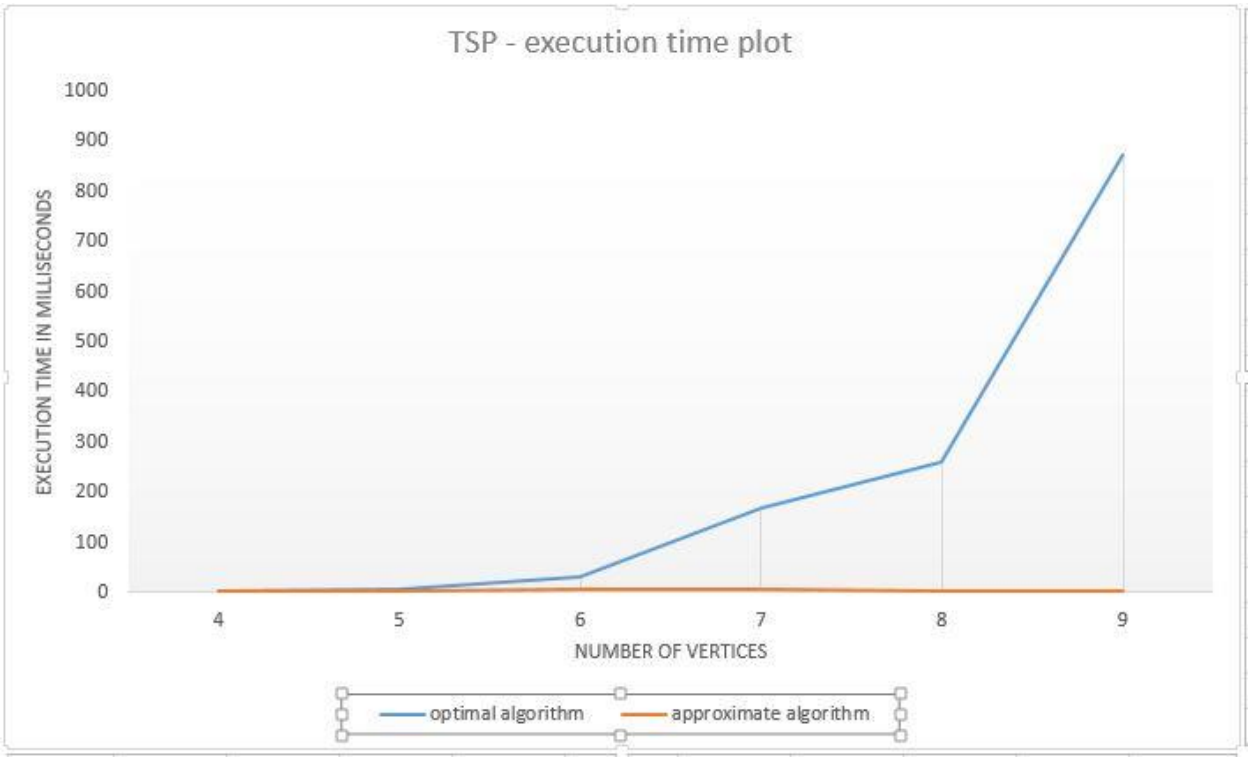
The following are the pictorial representation of the empirical comparison done between the approximate and optimal algorithm.

I used randomly generated cost matrix of increasing size of vertices for this empirical study. This random cost matrix is generated using number of vertices as seed. This will enable us to generate same cost matrix for different executions for given constant number of vertices.

To avoid effects of caching, I executed the same program for a minimum of 10 times and averaged the execution times and costs for given size of vertices set.

All the data collected is tabulated in "travelingSalesmanProblem" excel file that is provided along with this report.

Graph 1:

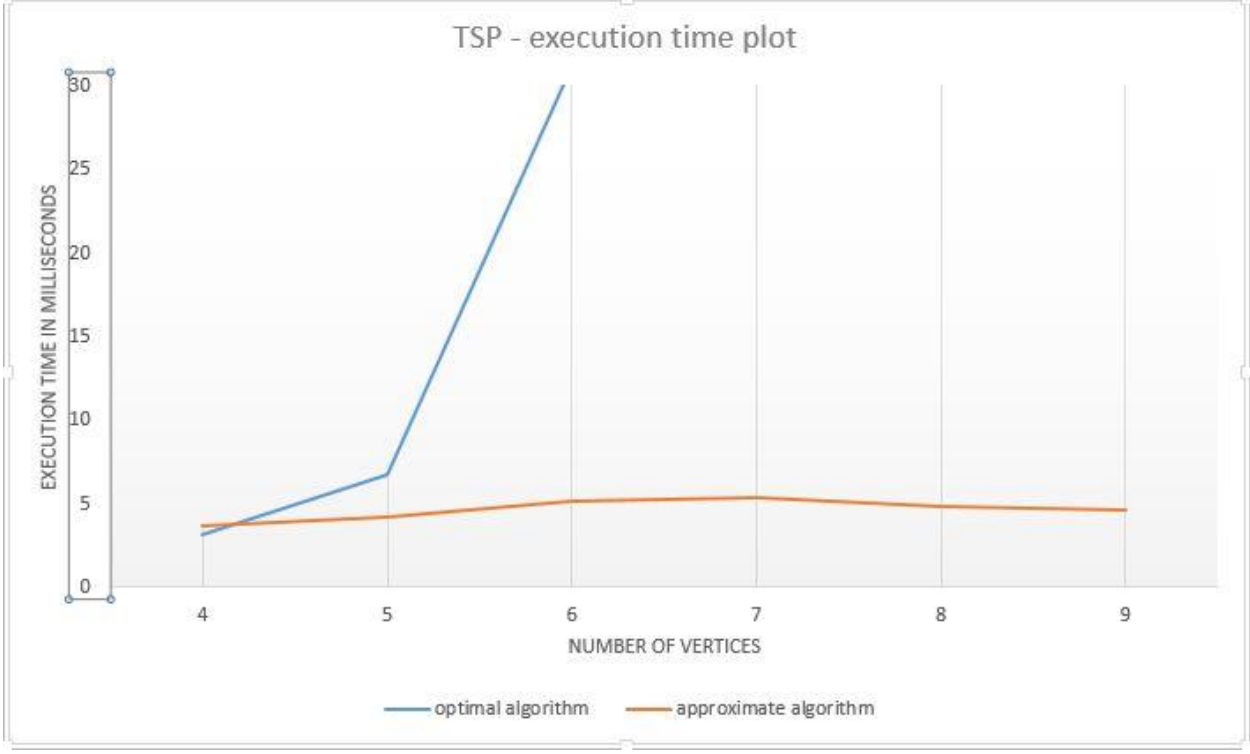


The above graph shows a comparative plot of the execution times of optimal and approximate algorithms. The dark blue line shows the actual plot of the execution time of the optimal algorithm which can be seen as trending to be an exponential graph. On other hand the approximate algorithm remains to be fairly linear as depicted by the orange line in the graph.

The actual data that is used to plot this graph has been tabulated in "travelingSalesmanProblem" excel file and provided along with this report.

It can be seen from the above graph how the optimal solution's performance degrades drastically with increase in the number of vertices.

Graph 2



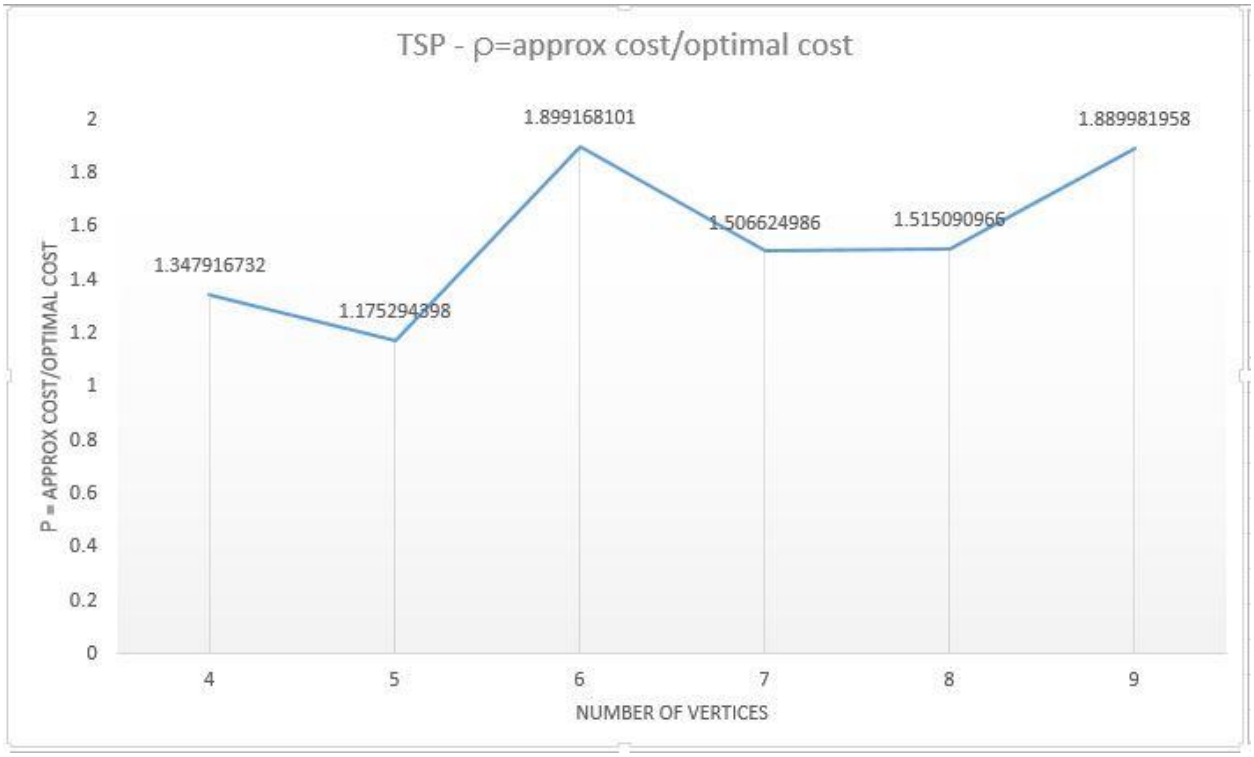
This graph dials down the scale of the execution time so that we can observe the trends of the execution time of optimal and approximate algorithms more clearly. Please note that for small number of vertices (4 or less) the optimal algorithm performs better than the approximation algorithm. But as the number of vertices increases the increase in the execution time of optimal algorithm is exponential whereas the approximate algorithm stays relatively polynomial or linear.

Note: The code provides the execution times in terms of nanoseconds. We have recorded them in nanoseconds and converted the data to milliseconds.

Now that we have observed the execution times, let us observe the graph that would shed light on the cost of the path that is an outcome of execution of each algorithm. The theoretical discussion has shown $c(H) \leq 2c(H^*)$. That is, cost of approximation algorithm is

less than twice the cost of optimal algorithm. Hence the approximation ratio ρ(n) = c(H*)/c(H) is less than or equal to 2.

Graph 3



This graph shows you the trends of the approximation ratio for various vertices. It can be clearly seen that the approximation ratios stays between 1 and 2. If the approximation ratio is one that means the approximation algorithm has achieved the same path as the optimal algorithm computes. The data set used to build this graph is tabulated in the same excel file as before.

From the above observation it is fair to assume that if an approximation ratio less than or equal to two is acceptable then the approximation algorithm stated in Corman et al. is efficient.

Another Approximation Algorithm for TSP

Now we will look at an approximation algorithm that I have devised to solve the traveling salesman problem. This algorithm uses greedy technique of listing the nearest destination

from a source and proceeding to that place before doing the same from the then arrived destination.

Let's begin the discussion by introducing the algorithm.

Given G(V,E) is a given graph and we have a matrix w(u,v) that gives us the cost or weight of direct traversal from u to v, where u,v belong to V(G).

```
optimalWeight <-- ∞
for each v <-- V(G)
      add v into C[|V|]
      add V(G)-v into R[|V|]
      pathWeight <-- 0
      v' <-- v
      while(pathWeight < optimalWeight and R is not empty)
            find v" for which w(v',v") is least
            pathWeight <-- pathWeight + w(v',v")
            add v" to C
            remove v" from R
            v' <-- v"
      if R is empty
            pathWeight <-- pathWeight + w(v',v)
            if optimalWeight > pathWeight
                  optimalWeight <-- pathWeight
print optimalWeight
```

This algorithm calculates the path weight in a greedy manner my choosing a vertex and finding its immediate nearest neighbor. Then it traverses to that neighbor and finds nearest neighbor from there. To make sure this algorithm generates a Hamiltonian path in the first place it remembers the vertices traversed in a data structure and choose its nearest neighbor from the remaining set of vertices. Because the minimum weight path thus calculated is dependent on the vertex chosen initially we repeat this algorithm with each vertex being the initial vertex in successive iterations and choose the ham cycle with least weight.

The running time of this algorithm is clearly $\Theta(V^2)$

Let us now conduct a relative study between the optimal TSP and my approximate algorithm.

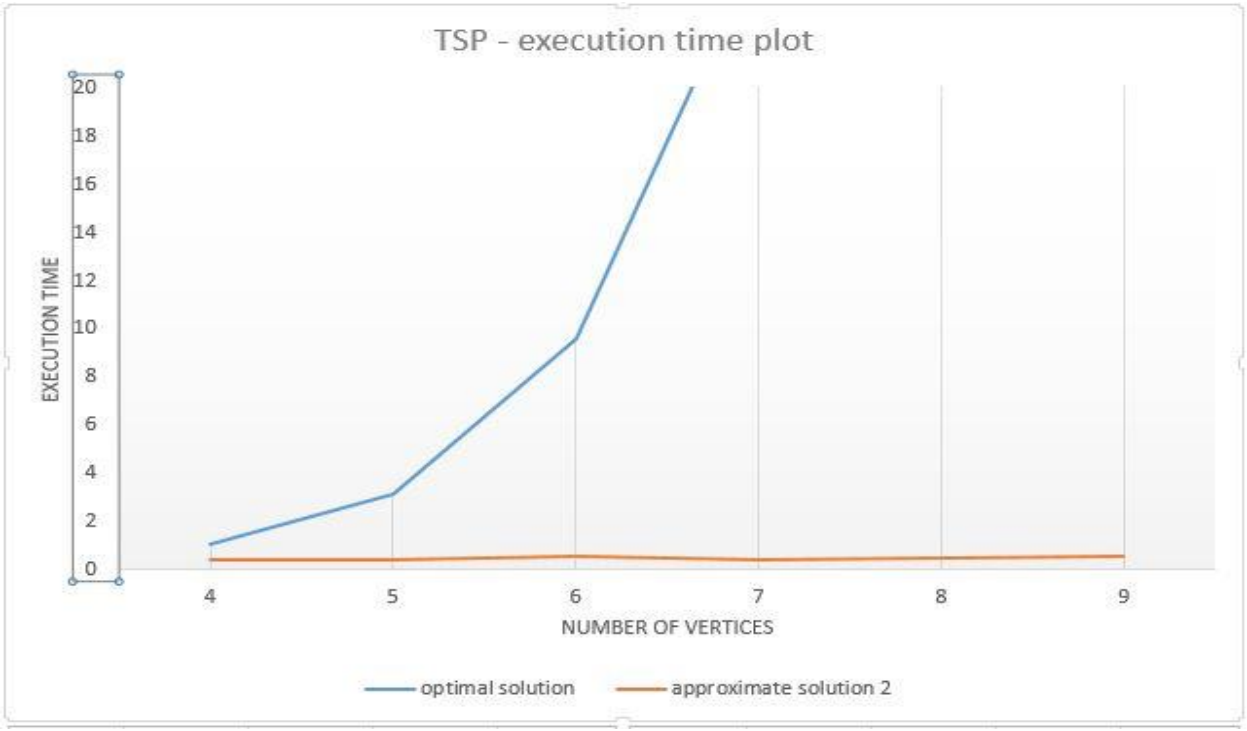
Empirical Comparison between Optimal TSP and my Approximation algorithm

The data set used in this empirical study is tabulated in the excel file "travelingSalesmanProblem-approx2". Since the cost matrix used is common for previous empirical study conducted and this empirical study I did not tabulate that.

The Source code this time has been improvised to automatically calculate the average of the 10 samples to avoid effects of caching. Each sample has a thread execution break of 1 second. Care is taken that this will not affect the execution times of the optimal or new approximation algorithm.

Let's begin.

Graph 1



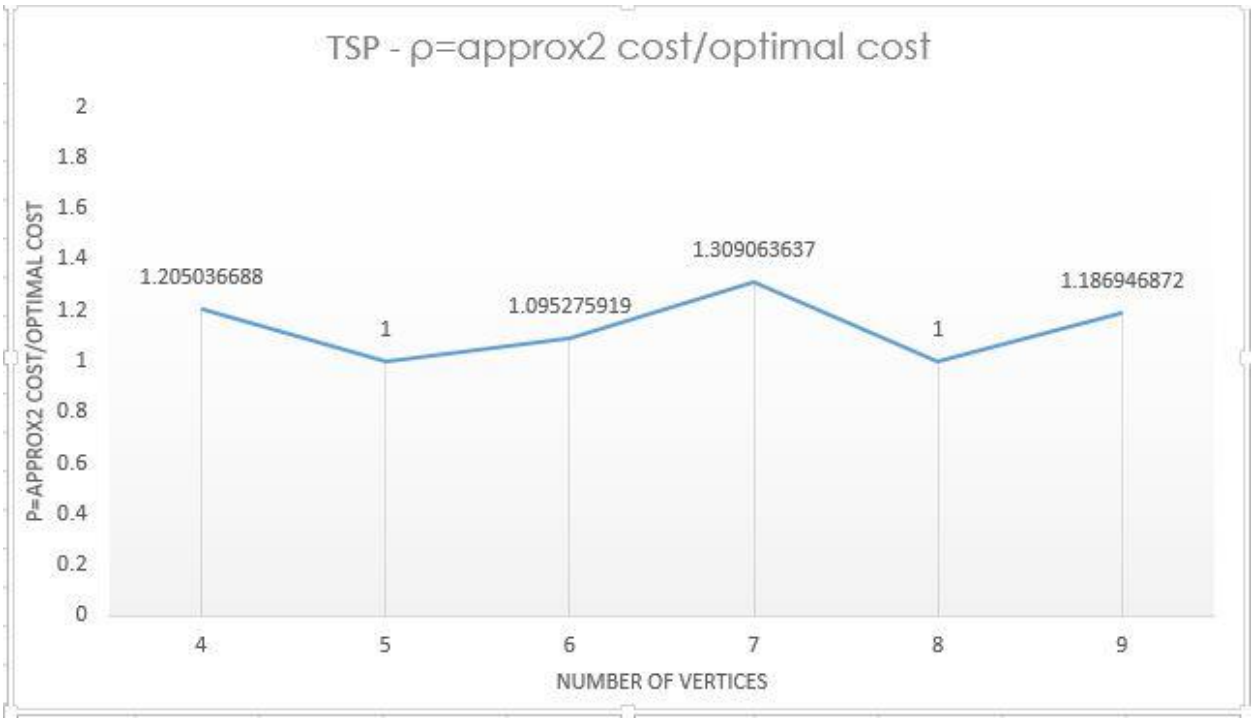
This graph shows a similar trend as the previous approximation algorithm had. But a closer look shows that its performance is slightly better.

You could also notice that no point we could see the performance of the optimal algorithm becoming better than the performance of the approximation algorithm.

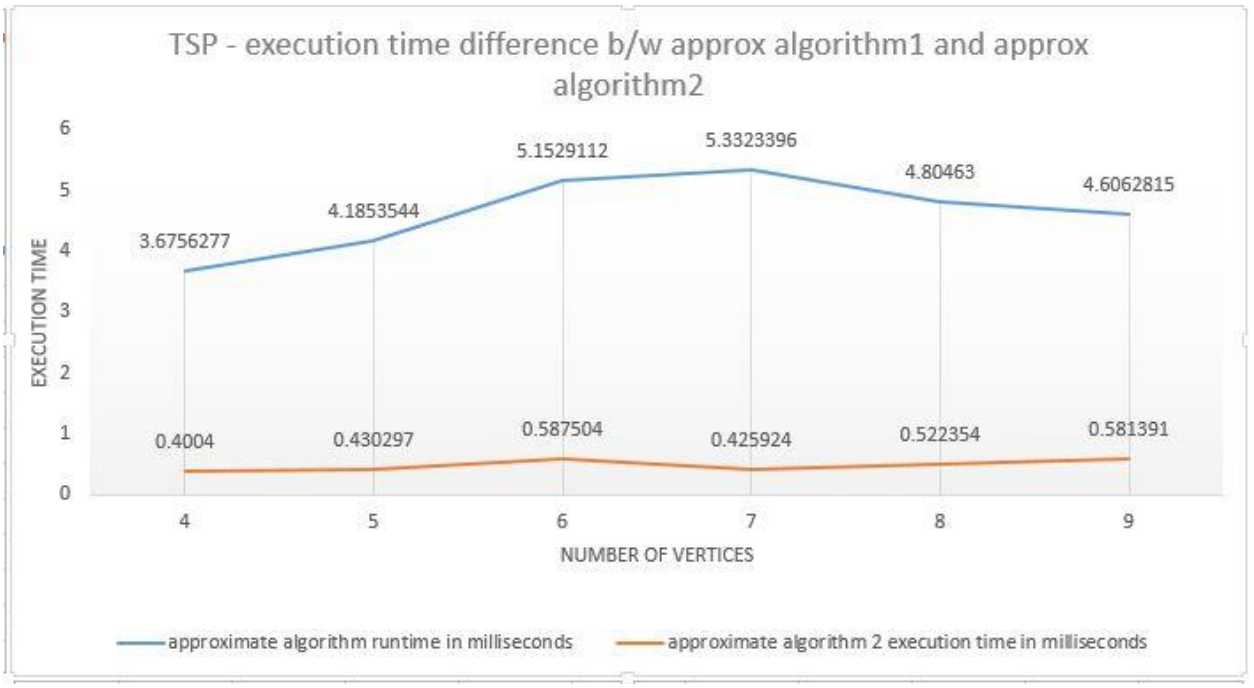
Let us now look at the graph that show how the approximation ratio of the algorithm behaves with increasing number of vertices.

Graph 2

It is interesting to note that the approximation ratio in few cases is actually one. This means the approximation algorithm actually provided the same result as the optimal algorithm but this time with very less execution time.

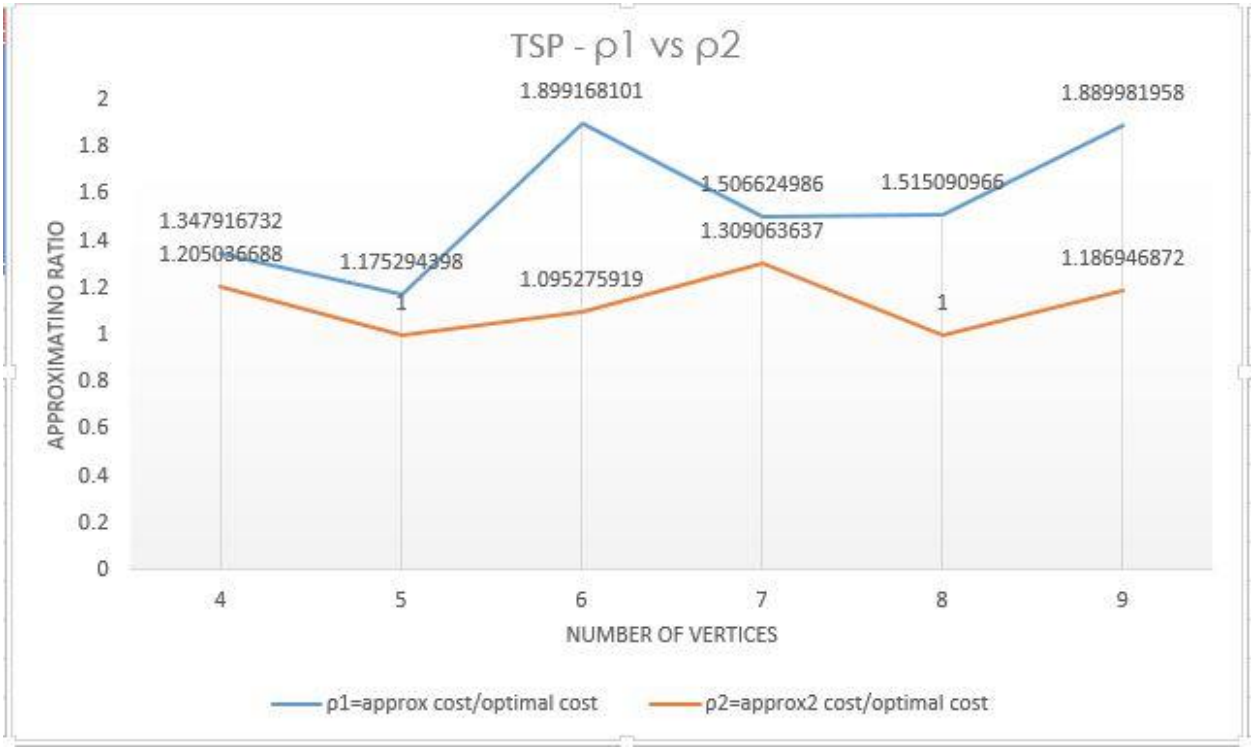

TSP - ρ=approx2 cost/optimal cost
P=APPROX2 COST/OPTIMAL COST
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1.205036688
1
1.095275919
1.309063637
1
1.186946872
4
5
6
7
8
9
NUMBER OF VERTICES

Graph 3


TSP - execution time difference b/w approx algorithm1 and approx algorithm2
EXECUTION TIME
6
5
4
3
2
1
0
3.6756277
4.1853544
5.1529112
5.3323396
4.80463
4.6062815
0.4004
0.430297
0.587504
0.425924
0.522354
0.581391
4
5
6
7
8
9
NUMBER OF VERTICES
approximate algorithm runtime in milliseconds
approximate algorithm 2 execution time in milliseconds

By collating the data collected in both studies we can draw graphs that can give us comparative picture of both approximation algorithms. This current graph shows trends in execution time of both algorithms. We can clearly see that the second approximation algorithm that has been devised using greedy techniques performs better than the former stated in Corman et al. book.

Graph 4



This graph that gives us a pictorial comparison between the approximation ratios of the two algorithms. ρ1 depicts the approximation ratio of former approximation algorithm and ρ2 the later.

We can clearly see that ρ2 costs are closer to the optimal algorithm than ρ1.

Bin Packing Problem

Suppose we have a set of n items each having a specific weight. We need to pack these items in bins. Each bin has identical capacity c. if w_i is the weight of ith item then $0 < w_i < c \; \forall \; 0 < i < n$.

Optimal solution to bin packing problem deals with finding the minimum number of bins that can be used to pack all the items.

Bin packing problem is NP- Hard problem as the NP-Complete problem of Sub-set sum can be polynomial reduced to bin packing problem. And also there is no known verifiable algorithm that could verify the validity of a certificate in polynomial time.

I coded the optimal and approximate solutions to the bin packing problem in C language using the native gcc compiler and gdb debugger that are provided with Ubuntu distribution. This was my first frantic attempt with C language after a long time and through this attempt I gained a lot of exposure on the benefits of vim editor and gdb debugger.

The algorithm to the optimal solution is fairly simple. I repeatedly executed the subset sum optimal algorithm (EXTRACT-SUBSET-SUM [5]) until the set of items is empty.

Then the number of times we execute the subset sum algorithm would be the number of bins used.

Note: In the source code I used a different but similar nomenclature. I used “trucks” instead of bins and “boxes” instead of items.

The approximation algorithm used is called first fit algorithm.

First fit algorithm is a pure greedy technique of packing all the items in sequence until the bin is full, continuing the process until all the items are packed.

Now we shall look into the empirical study comparing the performance trends and approximation ratio trends of the first fir algorithm.

The data is tabulated in a excel file named “BinPacking”.

Graph 1

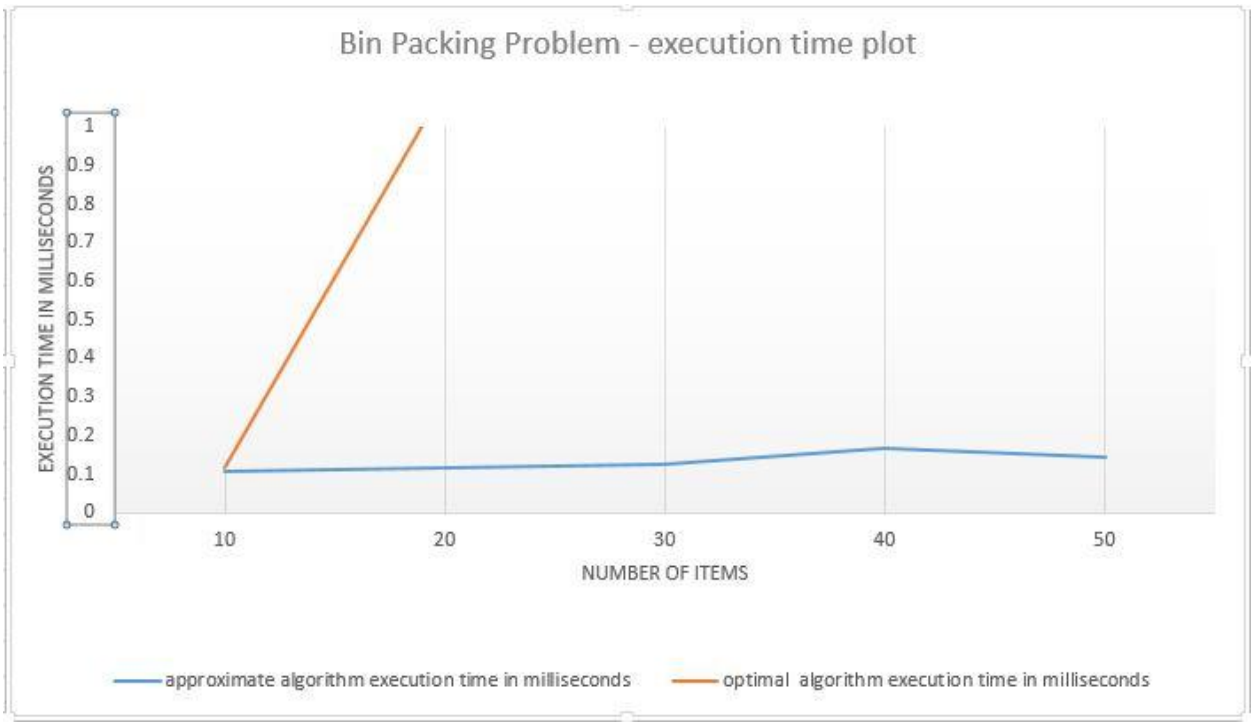
This graph details the trend in execution time of both approximation algorithm and optimal algorithm. In this plot we can clearly see how the optimal algorithm takes exponential trend whereas the approximate algorithm remains polynomial.

In order to avoid any effects of cache I took average of 5 readings for each value of "number of items".

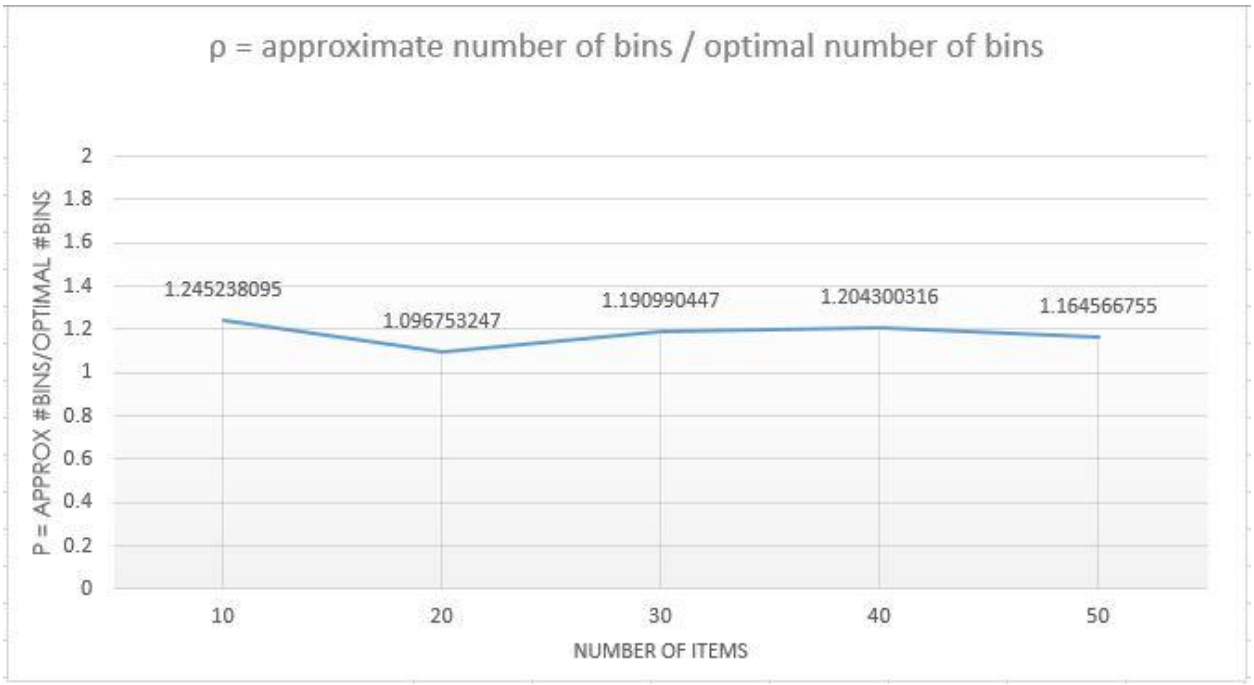
The scale chosen for the Y-axes does not allows us to observe the trend in the approximation algorithm execution time. To enable that I dialed down the Y-axes

Graph 2

In this graph we can observe how the first-fit algorithm which is our choice of approximation algorithm for bin-packing problem


Bin Packing Problem - execution time plot
EXECUTION TIME IN MILLISECONDS
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
10
20
30
40
50
NUMBER OF ITEMS
approximate algorithm execution time in milliseconds
optimal algorithm execution time in milliseconds

Graph 3


ρ = approximate number of bins / optimal number of bins
P = APPROX #BINS/OPTIMAL #BINS
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1.245238095
1.096753247
1.190990447
1.204300316
1.164566755
10
20
30
40
50
NUMBER OF ITEMS

Since the performance of the optimal algorithm degrades so quickly and the approximation algorithm stays polynomial even for huge number of items it might be beneficial to go for approximation algorithm. Above graph shows the trends in approximation ratio behavior for different values of number of items.

We can clearly see that the approximate algorithm stay close to the optimal algorithm as the line is close to horizontal hypothetical line at $\rho = 1$.

Conclusion

This independent study has given me hands on experience in coding and implementing two of the most popular NPC and NP-Hard problems. It gave me a holistic understanding of why exponential running time algorithms are useless for real world applications and how clever approximations can be made with acceptable over-heads.

Through this independent study I had the opportunity to explore the dreaded territory of C programming language like pointers and structures. In a relentless effort to implement the algorithms I overcame many obstacles that are thrown to a professional programmer like fixing segmentation faults etc. Finally I emerged triumphant and have become a better programmer.

I thank Dr. Bylander for providing me with this opportunity.

Venkat Rahul Dantuluri

@01518869 / fpj626

fpj626@my.utsa.edu

Ph: 210 965 7404

References

[1] Introduction to Algorithms 2nd Edition, Thomas H. Cormen et al.

[2] https://en.wikipedia.org/wiki/Optimization_problem

[3] https://en.wikipedia.org/wiki/Approximation_algorithm

[4] https://en.wikipedia.org/wiki/NP-complete

[5] Pg. 1045 Introduction to Algorithms 2nd Edition, Thomas H. Cormen et al.