Reduced Complexity Divide and Conquer Algorithm for Large Scale TSPs

Hoda A. Darwish, Ihab Talkhan
Computer Engineering Dept., Faculty of Engineering
Cairo University
Giza, Egypt

***Abstract*—The Traveling Salesman Problem (TSP) is the problem of finding the shortest path passing through all given cities while only passing by each city once and finishing at the same starting city. This problem has NP-hard complexity making it extremely impractical to get the most optimal path even for problems as small as 20 cities since the number of permutations becomes too high. Many heuristic methods have been devised to reach "good" solutions in reasonable time. In this paper, we present the idea of utilizing a spatial "geographical" Divide and Conquer technique in conjunction with heuristic TSP algorithms specifically the Nearest Neighbor 2-opt algorithm. We have found that the proposed algorithm has lower complexity than algorithms published in the literature. This comes at a lower accuracy expense of around 9%. It is our belief that the presented approach will be welcomed to the community especially for large problems where a reasonable solution could be reached in a fraction of the time.**

Keywords*—Traveling Salesman Problem; Computational Geometry; Heuristic Algorithms; Divide and Conquer; Hashing; Nearest Neighbor 2-opt Algorithm

I. INTRODUCTION

Divide and Conquer is an algorithm method used in search problems. As the search problem increases this method proves to be one of the best in reaching quick solutions; not only does it breakdown the search problem for easier calculations, in some cases it also allows for parallelizing the search hence reaching faster results. It has come to our notice that not many or not enough tries were given to the Divide and Conquer method when it comes to the Traveling Salesman Problem (TSP). The trend in resolving TSP is for Local Search algorithms and Evolutionary algorithms. Most of the research targets enhancing the constraints and fitness functions of these 2 categories of algorithms to reach a better solution. In most cases, these enhancements affect computational complexity making the resulting algorithms unfeasible for large scale problems.

For TSP, eliminating the long paths between any 2 cities/points in advance enables us to find quickly a more optimum solution. By dividing the search space or plane into pieces, we are effectively eliminating the paths between cites at the 2 ends of the search space thus, decreasing the number of paths we need to search. The plane/space is divided into "Buckets" each holding a set of points that are within a specific distance from each other. In the most ideal situation of evenly distributed points, the Heuristic TSP would now need to find the path for N/b points only where b is the number of buckets.

In the case of NN 2-opt, finding the path of N/b points requires a fewer number of iterations to reach a near optimum path and a much shorter run time. Accordingly, it is expected that the computational complexity of the Hashed Bucket algorithm will be of a much lower order of magnitude as we shall see in this paper.

The rest of this paper is organized as follows; Section II outlines the problem we are trying to address. Section III gives a briefing on TSPs and the current algorithms used for their resolution while Section IV presents a literature survey of related work. Section V describes our proposed solution. We then discuss the flow of our system in Section VII. Finally, our experimental results are presented in section VIII.

II. PROBLEM DEFINITION

The traveling salesman problem asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible path that visits each city exactly once and returns to the origin city? The complexity of such a problem is NP-hard making it extremely unrealistic to solve optimally.

The problem addressed here is how to improve Local Search Algorithms specifically the Nearest Neighbor 2-opt using a spatial Divide and Conquer method to obtain a new hybrid faster Heuristic algorithm. This poses the challenge of deciding the correct search space division and how these space divisions impact the performance of the NN 2-opt.

III. BACKGROUND

TSP is a very old problem with many references in literature as well as a long standing history. The first instance of the traveling salesman problem was documented by Euler in 1759. Euler wanted to address the problem of moving a knight to every position on a chess board exactly once as explained in [1]. The constraint set by Euler was that the knight must move according to the rules of chess and must visit each square exactly once.

A. Types of TSP

There are 2 main characteristics of TSPs. Depending on these, the problem representation may use different data structures and different calculations.

1) Symmetric vs. Asymmetric TSPs: a symmetric TSP is a problem where the distance from point A to point B is equal the distance from B to A. Asymmetric TSPs is when the distances

from A to B and vice versa are not equal. For example: if we consider the effort needed to go up a hill higher than the effort needed to go down then we have an Asymmetric problem.

2) Euclidean/Planer vs. 3 Dimensional: problems that consider only 1 constraint for the distance between the TSP points/cities can be considered planer. The most famous example of that would be the Euclidean distance. Once we start considering geographical distances, time or monetary costs we find that we have more constraints and hence, more dimensions for the TSP problem representation.

B. Complexity and Optimality

When we assess TSP algorithms, we look into optimality as well as complexity. Complexity is key given the number of permutations needed to calculate all possible paths; as we shall see in for exact algorithms in the following section. Yet in some cases, we can easily get an approximated path so it becomes necessary to measure the optimality of that path. By optimality, we mean how close it is to the real shortest path of the problem.

C. Exact (Non-Heuristic) TSP Algorithm

Simply put, there is only 1 way of finding the most optimal path for a TSP: comparing all possible paths and picking the shortest. Unfortunately, this Brute Force seraph method is not realistic as it means we must enumerate all possible permutations for the points in the TSP. In other words, for a problem with N points, we would need to look through N! – N Factorial – possible solutions. For example, a simple problem of 10 points would require passing through (and comparing) 3,628,800 possible paths. Such an approach would be impractical in real world situations where we would need to solve TSP for a huge number of points. The complexity of this approach is O (n!) where n is the number of points. Algorithms with such complexity are called NP-hard. In the case of NP-Hard problems, other means of reaching a solution are required as we shall see in the next section.

D. Heuristic TSP Algorithms

The traditional lines of attack for an NP-hard problem – when exact optimal methods are unfeasible – are the following:

- Devising "suboptimal" or heuristic algorithms, i.e., algorithms that deliver either seemingly or probably good solutions, but which may prove to be suboptimal.
- Finding special cases for the problem ("sub-problems") for which either better or exact heuristics are possible.

The TSP problem remains NP-hard even for the case when the cities are in the planer Symmetric Euclidean problem. Various heuristics and approximation algorithms have been devised specifically for TSP. Modern methods can find solutions for extremely large problems within a reasonable time and which are quite close to the optimal solution.

There are many types of Heuristic TSPs in the literature. Following is an overview of the main categories:

1) Tour Construction Algorithms: these algorithms gradually build a tour by adding a new city at each step. This approach is always quiet simple, but often too greedy. The first distances in the construction process are reasonably short, whereas the distances at the end of the process usually will be rather long. The most popular algorithm in this family is the Nearest Neighbor (NN). NN starts at some random city and then visits the city nearest to the starting city and then keeps visiting the nearest city that has not been visited so far until all cities are covered. It is a poor heuristic with the only simplicity as an advantage so it is normally used for small size problems.

2) Iterative Local Search (ILS) Algorithms: these start out with a complete solution at a certain optimality and iteratively try to change the features of the solution until a more optimal solution is found. For TSP, the initial complete solution can be a random tour through the problem with total cost S. The iterative changes would involve exchanging edges or paths between 2 or more cities and comparing the resulting tour of cost S' to S. If S' is a more optimal tour, we start iterating on that. If S' is worse than S, we discard that tour and begin iterating on other city pairs. A stopping criteria must be set in advance so that the algorithm doesn't iterate endlessly on all cases. There are many variations on the ILS, for example:

a) 2-opt Heuristic algorithm: this is the most basic of the ILS algorithms:

- Start with a given tour.
- Replace 2 links of the tour with 2 other links in such a way that the new tour length is shorter.
- Repeat until no more improvements are possible.

*b) 3-opt Heuristic Algorithm:*this is the same as the 2-opt but we pick 3 edges or links to replace instead of just 2 edges.

K-opt or Lin–Kernighan Heuristic Algorithm: this a generalization on the 2-opt and 3-opt algorithms that allows k-opt moves. It has many different constraints and modifications in an attempt to improve optimality and complexity. As explained in [2], the original algorithm as implemented by Lin and Kernighan in 1971, had an average running time of order $N^{2.2}$ and was able to find the optimal solutions for most problems with fewer than 100 cities. However, this algorithm is not simple because the number of operations to test all k-exchanges increases rapidly as the number of cities increases. In a naive implementation, the testing of a k-exchange has a time complexity of $O(N^k)$. Furthermore, there is no upper bound of the number of exchanges. Accordingly, the usefulness of general k-opt sub-moves usually depends on the candidate TSP. Unless it is sparse, it will often be too time consuming to choose k larger than 4. Another drawback is that k must be specified in advance and it is difficult to know what k to use to achieve the best compromise between running time and quality of solution. To overcome the drawbacks of the traditional LK algorithm, Lin and Kernighan introduced a powerful variable-opt algorithm: at each iteration, the algorithm examines – for ascending values of k – whether an interchange of k-links may result in a shorter tour. This continues until some stopping conditions are satisfied. Many other variantions and enhancements can be found in [3].

3) Evolutionary Algorithms: As the name implies, evolutionary algorithms follow nature in an attempt to reach

the best solution for optimization problems. Genetic algorithms (GAs) are one of the most popular evolutionary techniques. Taken from nature, GAs use crossover and mutation to solve optimization problems. GAs are loosely based on natural evolution and use a "survival of the fittest" technique, where the best solutions survive and are varied until we reach a good result. The incorporation of the survival of the fittest idea provides a means of searching the problem space without enumerating every possible solution. A GA works by first 'guessing' a set of solutions and then combining the fittest solutions to create a new generation of solutions which should be better than the previous generation. We may also include a random mutation element to account for the occasional 'mishap' in nature. As [4] explains, the main disadvantages of GAs are premature convergence and poor local search capability. In order to overcome these disadvantages, evolutionary adaptation algorithms based on the working of the immune system have been devised. The interested reader can refer to [5], and [6] for more samples.

IV. Related Work

Being able to solve large scale TSPs has been of great interest to many. In this section, we give an overview of some proposed solutions and their usefulness for different types of TSP sizes.

1) Medium Scale TSPs (500 to 3000 points): In [7], the authors look into solving TSP problems with hybrid, iterative extended crossover operators for GA. The objective of the hybrid algorithm is to efficiently search for the optimum solution while maintaining the diversity of the cyclic paths composing the population. It is a kind of hybrid method which combines Edge Assembly Crossover (EAX) with Ant Colony Optimization. The algorithm was verified on test data of size up to 1173 cities. The optimal path was obtained but required 109 hours to calculate! In fact, the computational time increases exponentially with the increase in number of cities.

2) Large Scale TSPs (5000+ points): In [8], the authors consider a k-means partitioning algorithm to divide the initial TSP problem into multiple partitions to be solved separately then merging. The partitioned sub-problems are merged using Lin Kernighan algorithm. To partition the TSP problem, the authors represented the problem as a graph and used multilevel graph partitioning. Multilevel k-means graph partitioning reduces the size of the graph by collapsing vertices and edges as explained by [9]. It divides the graph into smaller graphs and then refines the partition during an "un-coarsen" phase to construct a partition for the original graph. For solving each sub-problem a greedy tour construction heuristic is used to get a good solution of individual small partitions. After solving each partition, step by step recreation of graph is carried out by simply adding each solved partition back to the graph. The algorithm was tested on TSPLIB and provided quite optimal TSP tours but no time complexity was clarified. It is known that the average LK complexity is $O(N^{2.2})$; by clustering and using LK in the coarsening phases of merging the partitions, it is clear that the complexity of this algorithm is definitely more than that of the proposed Divide and Conquer NN 2-opt.

Many other researchers have attempted to enhance the complexity of LK implementations and have reached O (N^2) yet the tradeoff is extra memory of O(N) making it again impractical for large scale problems.

3) HW Parallelization of Large Scale TSPs: Given the complexity of TSP algorithms, the speed up and execution time gained from increasing HW resources cannot be expected from normal software solutions so will not be compared with the algorithm proposed in this paper. It does show though that partitioning the problem still allows us to get relatively optimal tour solutions. The authors of] 10[(2007) introduced the notion of "symmetrical 2-Opt moves" which allowed them to uncover fine-grain parallelism when executing the 2-opt local search optimization algorithm. Once the parallelism is apparent they use an FPGA (or FPGA simulator) to resolve each sub problem gaining an average speed up of 600%.

V. Proposed Approach

The proposed algorithm depends on the theory that "the addition of shortest set of paths will yield shortest total path". Accordingly, if we have N points getting the shortest path for N/b points then consolidating the set of b paths will give us the shortest path through the N points. We divide the N points according to their proximity to each other in the search space using x and y dimensions.

For example, a square space of area A^2 will be divided into smaller areas called Buckets of area A^2 /b where b is the number of Buckets. All points in the same bucket are considered close in proximity and a heuristic TSP algorithm is used to get their shortest path. Given that the number of points in area A^2/b is much less than the total area, the heuristic algorithm has a good chance of finding the optimal path in a much shorter execution time. Once all b paths have been obtained, merging them into a single path for the N points should yield the shortest total path. Fig. 1 shows a simple example where the search space/plane has been divided into 4 buckets.

This approach was inspired by the work done in [11] that is based on the Fixed-Radius Nearest-neighbor problem. The authors of [11] show that bucket hashing is very effective in the domain of electronic design automation specifically in chips of *millions* of transistors as it breaks the problem into manageable pieces for quicker resolution.

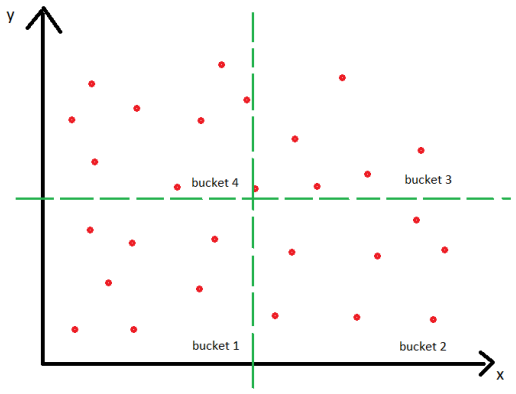

Fig. 1. Search Space division using Buckets

VI. System Flow

The system is comprised of a set of functions that interact with each other. The flow of processing can be seen in the chart in fig. 2. The first step of the process is to parse the input file to get the points that make up the TSP problem. Using the input criteria for bucket size, a decision is made regarding the division of the search space. The following functions then handle the creation of the buckets and hashing the TSP points into the different buckets. Once the buckets are ready, we can then consider each bucket as a separate TSP problem for the regular heuristic NN 2-opt TSP algorithm and thus, the algorithm is run for each bucket. The final step in our flow is to merge the smaller bucket TSP solutions into 1 solution for the original input point thus providing 1 single path and its total cost.

The implementation explained above makes the assumption that all points are connected (in case the TSP doesn't fit this constraint, setting the distance between the unconnected points to infinity should automatically eliminate the path but this theory has not been tested here). We also assume that the input TSP is a Symmetric TSP and that the distance used is Euclidean.

The algorithm depends on finding the minimum path for each bucket and then merging the result. The sum of these local minima may not in fact result in the global minimum. This is one of the disadvantages of the Heuristic algorithms in general yet given that the hashed algorithm complexity is significantly lower we can run the hashed algorithm with different averages or bucket sizes and choose the minimum depending on the original problem size.

A. Bucket Size Decision

The algorithm complexity depends heavily on the number of buckets used. Accordingly, we need a simple decider to use for dividing our search space. Heuristic TSP algorithms normally have an Average number of points that they can optimally get the shortest path for. Assuming the points are equally distributed, we use this average to divide the space according to the following equations:

1) Get the minimum possible number of buckets by dividing number of points on the input average.

2) Calculate the search space area A (maximum of x * maximum of y).

3) Get bucket width using eq. 1:

$$bucket\ Width\ = ceil(\frac{(maximum\ of\ x)^2}{minimum\ no.of\ Buckets}) \quad (1)$$

4) Get bucket length using eq. 2:

$$bucket\ Length = ceil(\frac{A}{minimum\ no.of\ Buckets * bucketWidth}) \quad (2)$$

B. Path Merging

The other important step in the algorithm is the merging of the separate bucket solutions to form a single final path. An example of the bucket path merging is shown in fig. 3.

We have Start point "S" for the bucket and a Transition Forward point "TF" for moving to the next bucket in each individual bucket path. When we merge, we remove the path between the TF and the point following it in the bucket; instead we merge it with the start point of the successor bucket. On the way back, we remove the last leg of the path back to the bucket

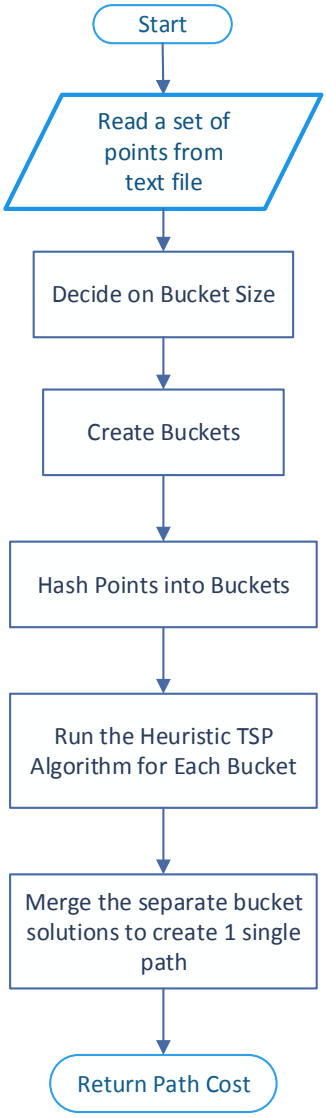

Fig. 2. System Workflow

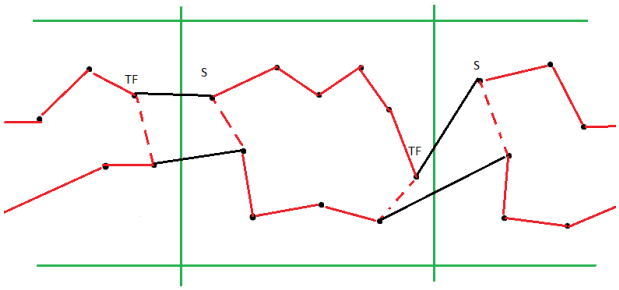

Fig. 3. Path Merging

start point and instead move to the point following the TF in the previous bucket. In other words, we delete the "dashed" red lines and add the solid black lines.

VII. Results and Analysis

The complexity of the simple NN 2-opt algorithm is O (N^2) where N is the input number of points of the TSP problem. In the hashed approach, we divide the space into buckets. Assuming that we have "b" buckets and that the points are evenly distributed on the buckets as "N/b" then the complexity of a single bucket is O($(N/b)^2$). To get the complexity of the entire Hashed algorithm we consider that we need to calculate the NN 2-opt for b buckets i.e. a complexity of O ($b*(N/b)^2$) which is equal O ($(N/\sqrt{b})^2$). As we shall see in the results, the above complexity is tangible numerically in the following example. If N = 493 and b = 9, the normal NN 2-opt would require 4932 = 243,049 calculations/computations while the Hashed technique would provide $(442)^2/9$ = 27,005 computations which is 11% of the simple algorithm computations.

To show the effectiveness of the proposed algorithm and its ability to solve all general cases, some test cases from TSPLIB were used with focus on large samples. The results obtained are from running the system under Windows 7 and using Matlab 7.0. The system specifications are: Intel Core CPU (1.6GHz) and a 4GB RAM.

We show the test results for 3 different test samples. For each sample, we list the results of the normal NN 2-opt algorithm as well as the Hashed algorithm with the different values of the bucket decider. We can conclude that using 20% of the problem size as the bucket decider tends to give the least error %.

The error is calculated using the eq. 3.

$$E\% = \frac{Calculated\ Path\ Value - Optimal\ Path\ Value}{Optimal\ Path\ value} * 100 \quad (3)$$

For the results the execution time is provided in seconds. This documented execution time does not include the time spent in parsing the input file because this is the same function in both hashed and un-hashed algorithms. It is important to note that the NN 2-opt requires an input of the number of iterations to be used. We have kept this at a constant of 4 for both the NN 2-opt and the hashed buckets algorithm. More iterations should theoretically decrease the error but after some test runs we found that 4 iterations are a suitable average as the added optimality is not proportional to the increase in time.

A. Sample 1: File d493.txt

- Number of Points: 493 with optimal path cost as per TSPLIB: 35,002.
- NN 2-opt without hashing has path cost = 36,099 and execution of 1.476s thus an Error % of 3.13%
- Results for the Hashed algorithm are in Table 1.

B. Sample 2: File rl5915.txt

- Number of Points: 5,915 with optimal path cost as per TSPLIB: 565,530.
- NN 2-opt without hashing (average results of 2 runs) has path cost = 591,715 giving an Error % of 4.63%. Its execution was 2004.5s which is equal to 33.5 minutes!
- Results for Hashing algorithm are in Table 2.

C. Sample 3: File rl11849.txt

- Number of Points: 11,849 with optimal path as per TSPLIB: 923,288.

TABLE I. D493 Results (Average of 100 runs)

Decider	Hashed Time	Hashed Cost	Hashed Error
10% = 50 (15 buckets)	0.168s	44,788	27.96%
20% = 100 (9 buckets)	0.235s	36,364	3.89%
40% = 200 (4 buckets)	0.261s	38,774	10.78%

TABLE II. RL5915 Results

Decider	Hashed Time	Hashed Cost	Hashed Error
10% = 600 (10 buckets)	39.64s	640,430	13.24%
20% = 1200 (6 buckets)	66.376s	617,670	9.22%
40% = 2400 (4 buckets)	155.015s	609,920	7.85%

TABLE III. RL11849 Results

Decider	Hashed Time	Hashed Cost	Hashed Error
5% = 600 (30 buckets) 20 runs average:	51.92s	1,116,000	20.87%
10% = 1200 (15 buckets) 20 runs average:	251.14s	1,053,300	14.07%
20% = 2400 (6 buckets) 3 runs average:	917.97s	1,002,600	8.59%

- NN 2-opt without hashing: the machine ran out of memory and thus no results were gained.
- Results for Hashing algorithm are in Table 3

VIII. Conclusion

As shown in the results section, the hashed bucket algorithm is very effective in reducing the overall execution time of large scale TSPs. Fig. 4 and fig. 5 show the comparison between algorithm methods and different decider values quite clearly.

We are able to reach a path in less than 10% of the time required for the original NN 2-opt. We understand that the tradeoff is in optimality yet a 9%~15% error is considered an acceptable margin for such a gain in execution speed.

We would also like to note that original NN 2-opt algorithm was unable to run on the limited specs of the machine after a certain size due to its memory consumption. Accordingly, another advantage of the algorithm is the possibility to reach results using limited memory and execution power. This begs the possibility that the algorithm would be useful in robots and applications that run on batteries (reduced power consumption) and limited size.

REFRENCES

[1] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, Springer-Verlag, 2nd edition, 1994

[2] Keld Helsgaun, "An effective implementation of the Lin-Kernighan traveling salesman heuristic", European Journal of Operational Research, 2000

[3] Keld Helsgaun, "General k-opt submoves for the Lin–Kernighan TSP heuristic", Springer and Mathematical Programming Society, 2009

[4] Donald Davendra, Traveling Salesman Problem, Theory and Applications, InTech, 2010

[5] Hirotaka Itoh, The Method of Solving for Travelling Salesman Problem Using Genetic Algorithm with Immune Adjustment Mechanism, 2010

[6] Oloruntoyin Sefiu Taiwo, Olukehinde Olutosin Mayowa & Kolapo Bukola Ruka, "Application Of Genetic Algorithm To Solve Traveling Salesman Problem", International Journal of Advance Research (IJOAR), Volume 1, Issue 4, April 2013

[7] Ryouei Takahashi, "Solving the Traveling Salesman Problem through Iterative Extended Changing Crossover Operators", 10th International Conference on Machine Learning and Applications, 2011

[8] Atif Ali Khan, Muhammad Umair Khan, & Muneeb Iqbal, "Multilevel Graph Partitioning Scheme To Solve Traveling Salesman Problem", Ninth International Conference on Information Technology- New Generations, 2012

[9] Chris Walshaw, "A Multilevel Lin-Kernighan-Helsgaun Algorithm for the Travelling Salesman Problem", Computing and Mathematical Sciences, University of Greenwich, Old Royal Naval College, Sep. 2001

[10] Ioannis Mavroidis, Ioannis Papaefstathiou, & Dionisios Pnevmatikatos, "A Fast FPGA-Based 2-Opt Solver for Small-Scale Euclidean Traveling Salesman problem", International Symposium on Field-Programmable Custom Computing Machines, 2007

[11] Hoda A. Darwish, Hoda N. Shagar, Yasmine A. Badr, et al., "A Hashing Algorithm for Rule-Based Decomposition in Double Patterning Photolithography", IEEE 22nd International Conference on Microelectronics (ICM), 2010

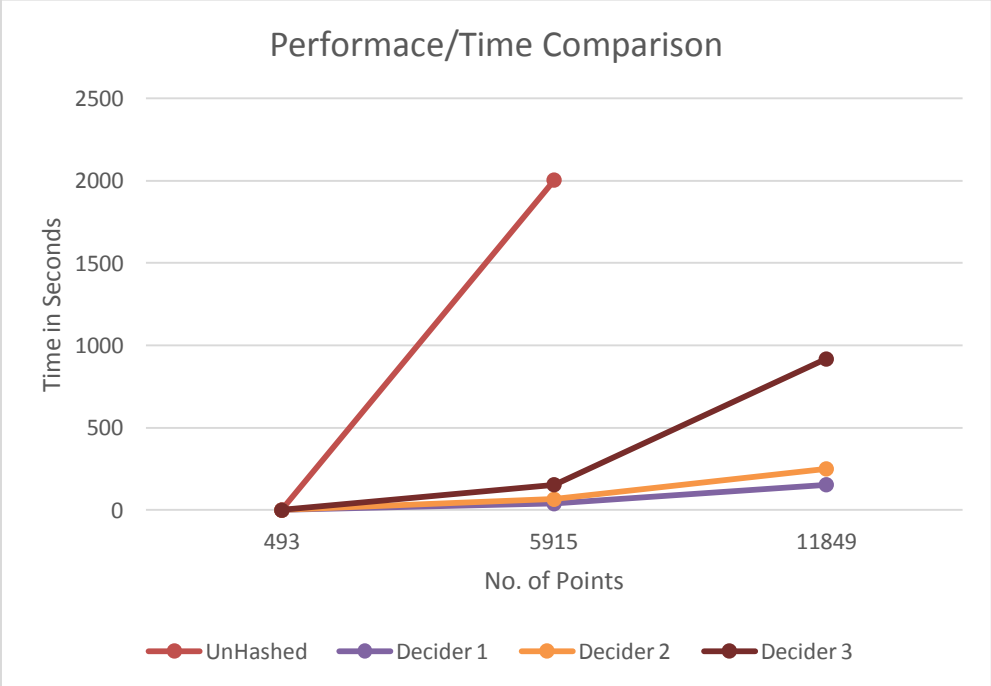

Fig. 4. Performance/Time Comparison

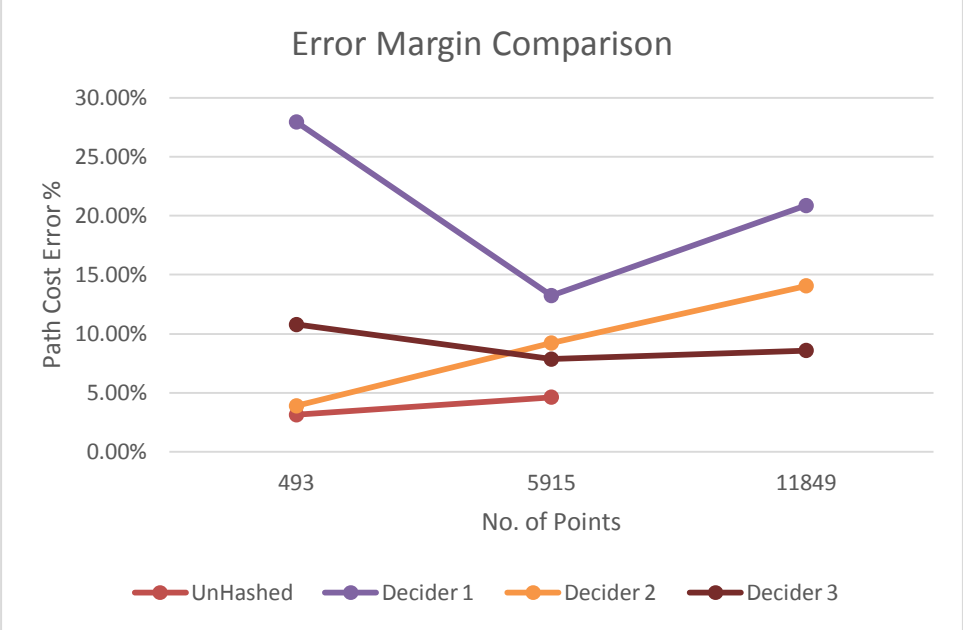

Fig. 5. Error Margin Comparison