An Empirical Evaluation of Approximation Algorithms for the Symmetric Traveling Salesperson Problem

Albert Haque, Jay Shah, Faisal Ejaz, Jia Xing Xu *
Department of Computer Science
University of Texas at Austin
{ahaque,jayshah,faisalej,jiaxing}@cs.utexas.edu

ABSTRACT

With applications to many disciplines, the traveling salesperson problem (TSP) is a classical computer science optimization problem with applications to industrial engineering, theoretical computer science, bioinformatics, and several other disciplines. In recent years, there have been a plethora of novel approaches for approximate solutions ranging from simplistic greedy to cooperative distributed algorithms. In this paper, we perform an evaluation and analysis of cornerstone TSP algorithms. We examine the theoretical bounds and empirical performance of the nearest neighbor, greedy, Christofides, and genetic algorithm approaches. We use several datasets as input for the algorithms including small datasets, a medium-sized dataset representing the United States capital cities, and a synthetic dataset consisting of 1,000 cities. We discover that the nearest neighbor and greedy algorithms efficiently calculate solutions for small datasets while Christofides has the best performance for both optimality and runtime for medium to large datasets. Genetic algorithms can occasionally find near-optimal solutions but have no guarantee and generally have long runtimes.

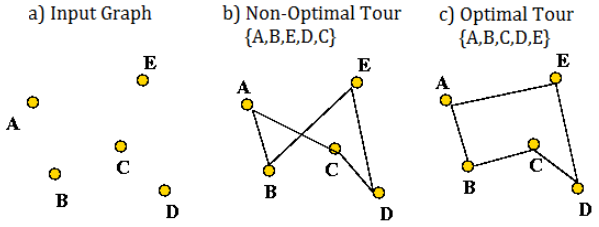
1. INTRODUCTION

Known to be NP-hard, the traveling salesperson problem was first formulated in 1930 and is one of the most studied optimization problems to date [12]. The problem is as follows: given a list of cities and a distance between each pair of cities, find the shortest possible path that visits every city exactly once and returns to the starting city.

The remainder of this paper is organized as follows. In Section 2, we briefly review the first algorithms developed and survey modern approaches to TSP. We describe our algorithms and key implementation details in Section 3. Results of the experiment and a description of the benchmark datasets are detailed in Section 4. A discussion in Section 5 explains the findings of our evaluation followed by a conclusion in Section 6.

2. BACKGROUND

Figure 1: Example TSP Input and Solutions



We demonstrate the TSP in Figure 1. The input is shown in subfigure (a) as a collection of cities in the two-dimensional space. This input can be represented as a distance matrix for each pair of cities or as a list of points denoting the coordinate of each city. In the latter method, distances are calculated using Euclidean geometry. A non-optimal tour is shown in subfigure (b). Although not shown in the figure, each edge will have some, non-negative edge weight denoting the distance between two nodes or cities. Due to the computational complexity of the TSP, it may be necessary to approximate the optimal solution. The optimal tour is shown in subfigure (c). For small graphs, it may be possible to perform an exhaustive search to obtain the optimal solution. However, as the number of cities increases, so does the solutions space.

Figure 2: TSP Complexity

Cities	Edges	Possible Tours
5	10	60
6	15	360
7	21	2,520
8	28	20,160
9	36	181,440
10	45	1,814,400
25	300	7.75e+24
50	1,225	1.52e+64
100	4,950	4.66e+157
n	$\sum_{i=0}^{n-1} i$	$n!/2$

Figure 2 lists the number of edges and total possible number of tours for a specific dataset size. The number of possible tours is $n!/2$ since we must adjust for the double-counting of a single path ($n!$ counts the same path twice: once with the start node appearing first and one with the end node appearing first).

Mathematical problems similar to the Traveling Salesman Problem date back to the 18th century. The basis of the problem was first discussed by Irish mathematician Sir William Rowan Hamilton and by a British mathematician Thomas Penyngton Kirkman.

The TSP problem itself was first formulated in the 1930s by Karl Menger in Vienna and Harvard. It was later studied by statisticians Mahalanobis, Jessen, Gosh, and Marks to use as an agricultural application, and the problem was popularized by mathematician Merill Flood with his colleagues at Research and Development Corporation in the 1940s. By the mid-1950s, solutions for TSP began to appear. The first solution was published by Dantzig, Fulkerson, and Johnson using a dataset of 49 cities. In 1972, Richard M. Karp proved that the Hamiltonian cycle problem was NP-Complete, which proves that the TSP is NP-Hard.

In modern day, the Traveling Salesman Problem has a variety of different applications to numerous fields. Examples among these applications include genome sequencing, air traffic control, supplying manufacturing lines, scan chain optimization, and many more.

2.1 Variants

Euclidean space, asymmetric TSP are modifications of the original problem. In the Euclidean space, $\mathbb{R}^2$, the input is typically given as a list of coordinates describing the location of each city [1]. Some variants extend this problem into the three-dimensional space. However, both the classical and Euclidean TSP symmetric edge weights. In the asymmetric variant, the edge $weight(a, b) \neq weight(b, a)$ [7]. This introduces additional complexities that are outside the scope of this paper. Some people are also researching multiple traveling salesperson problem with multiple travelers.

3. ALGORITHMS

We now move to a discussion of the algorithms used in the evaluation. First we describe the traditional nearest neighbor and greedy approaches in Sections 3.1 and 3.2. We then outline Christofides' Algorithm and then discuss the genetic algorithm being implemented in Sections 3.3 and 3.4.

3.1 Nearest Neighbor

The Nearest Neighbor algorithm is the simplest approach to TSP. It simply states to start with a vertex v, and continued add the neighbor closest to the most recently added vertex. It is important to consider the performance of this algorithm as the Nearest Neighbor heuristic is applied to many problems in computer science and serves as an overall benchmark.

The algorithm runs in $O(n^2)$ time but is not guaranteed to be optimal. As a matter of fact, there exists a subclass of TSPs in which the Nearest Neighbor algorithm consistently performs the worst among other solutions [8]. In randomized tests, the Nearest Neighbor produces a path that is an average of 25% longer than the optimal path [10]. We include Nearest Neighbor to provide a relative measure of optimality for other algorithms and heuristics.

3.1.1 Nearest Neighbor Implementation

The Nearest Neighbor heuristic was implemented in Java and included capability to import data as adjacency lists as well as time the duration of each calculation. After each file was imported from a text file (see Datasets, section 5.1), the City class stored the name of each city and its distance from the other n-1 cities available on the map.

After randomly choosing the first city, the algorithm looks for the shortest distance between any city we have already visited to any city we have not visited (this is $O(n^2)$ calculations). After checking all possible cities, the algorithm visits the city with the cheapest cost associated with it and starts the same loop again. This loop is broken once all cities on the map have been visited. Finally, it adds the cost of going from the last city to the first and thus, completing the tour.

The program outputs the cost as well as the running time of the operation.

3.2 Greedy

The Greedy heuristic is based on Kruskals algorithm to give a suboptimal solution to the Traveling Salesman Problem [11]. The algorithm forms a tour of the shortest route and can be constructed if and only if:

1. The edges must not form a cycle unless the selected number of edges is equal to the number of vertices in the graph.

2. The selected edges does not increase the degree of any node to be more than 2.

The algorithm starts by sorting all of the edges from least weight to most heavily weighted. After the edges are sorted, choose the least heavily weight edge in the graph, and add it to the tour if it doesnt violate the above rules. Continue by selecting the next least heavily weighted edge, and add it to the tour. Repeat this step until the number of selected edges in the tour is equal to the number of vertices in the graph. Thus, the result is a minimum spanning tree for the Traveling Salesman Problem.

The runtime for this greedy algorithm is $O(n^2 log(n))$ and generally returns a solution within 15-20% of the Held-Karp lower bound [14].

3.2.1 Greedy Implementation

The Greedy solution to TSP differs from the Nearest Neighbor heuristic because it uses a Kruskals approach to the problem. Instead of starting at a random node and building a tour suing the nearest beighbor of the selected node, the Greedy algorithm selects the least weighted edge and adds it to the tour regardless of if it is connected or disconnected to the current tour.

The Greedy algorithm was implemented in Java and is available on Github[1]. The program reads in datasets from a file and uses createMatrix to create a matrix weightedMatrix of the distances between cities. The algorithm first sorts all of the weights of the edges from lowest to most heavily weighted, and selects the lightest edge to begin the tour with. It then selects the next lightest edge and adds it to the tour as long as it doesnt create a cycle or make any vertices have a degree of more than 2. The algorithm keeps performing the loop until the number of edges in the tour is equal to the number of vertices in the graph. It then prints out all of the edges added to the tour, the running time of the operation in nanoseconds, and returns the path cost of the tour.

[1] `https://github.com/jenny-xu/Greedy-TSP`

Figure 3: Historical TSP Solution Milestones [5]

Year	Research Team	Size of Instance
1954	G. Dantzig, R. Fulkerson, and S. Johnson	49 Cities
1971	M. Held and R.M. Karp	64 Cities
1975	P.M. Camerini, L. Fratta, and F. Maffioli	67 Cities
1977	M. Grotschel	120 Cities
1980	H. Crowder and M.W. Padberg	318 Cities
1987	M. Padberg and G. Rinaldi	532 Cities
1987	M. Grotschel and O. Holland	666 Cities
1987	M. Padberg and G. Rinaldi	2,392 Cities
1994	D. Applegate, R. Bixby, V. Chavatal, and W. Cook	7,397 Cities
1998	D. Applegate, R. Bixby, V. Chavatal, and W. Cook	13,509 Cities
2001	D. Applegate, R. Bixby, V. Chavatal, and W. Cook	15,112 Cities
2004	D. Applegate, R. Bixby, V. Chavatal, and W. Cook	24,978 Cities

3.3 Christofides Algorithm

The Christofides Algorithm is a heuristic algorithm and finds a near-optimal solution. It is a polynomial 3/2 approximation algorithm for the TSP [13]. It's primary objective is to find a Hamiltonian cycle of minimum length.

We first start by constructing a minimum spanning tree M on G. Let O be the set of the odd-degree vertices of M [6]. O now contains leaves and potentially some internal vertices. Note, $|O|$ is even since the sum of the degrees of all the nodes must be even since it is exactly twice the number of edges. This implies that the graph on O has a perfect matching and finding a minimum-weight, perfect-matching P in polynomial time of O. Next take a graph on the edges E consisting of the edges P and M. This graph may have duplicate edges, since P and M might intersect [6]. All nodes now have even degree because an edge of P is incident to each node of O. Therefore a Eulerian Tour can now be found by traversing each edge exactly once. Start with the initial point and keep traversing using minimal paths and skipping vertices that have already been visited.

In various research it has been noted that the Christofides Algorithm can be extended for the k-depot TSP and its variants. The worst case analysis of this variant is known to be difficult to be calculate since this requires bounding the length of the minimum perfect matching for vertices having odd degrees that are found [16].

3.3.1 Christofides Implementation

Christofides Algorithm is a 1.5-approximation algorithm for the Traveling Salesman problem.This is achieved by reducing the problem to ultimately finding a Hamiltonian cycle of minimum length. The implementation by Bjrn Harald Olsen and Oscar Tckstrm has proved to successfully achieve a minimum cost path solution to the TSP. We implement this algorithm in Java for this experiment.

The TSP is reduced first to finding a minimum spanning tree. The source code for this implementation can be found on Github[2]. The main class `Christofides.java`, is initiated by calling the method solve which takes in a 2D matrix. Given this matrix, we calculate the MST by using Prim's algorithm. This completes in $O(n^2)$ time. The most important step is to achieve the 1.5-approximation is by graph matching. `greedyMatch` is called with the MST, and the input matrix which finds matching with the minimum set of odd degree vertices in polynomial time. `buildMultiGraph` builds an array of GraphNodes and we find an Eulerian circuit in the graph by calling `getEulerCircuit`. The final steps required are to sum up the weight of the minimum cost path, and to finally return the path itself.

The Java class `ChristofidesMain.java`, which is a driver program for Christofides algorithm, accepts an input file specified from the keyboard. The format of this input file should be an adjacency matrix with values delimited by a single space. The file is read to a 2D matrix using the singleton class `ChristofidesManager.java`. The output is written to standard out which runs in supported verbose mode. This mode prints all distances, parent nodes, matchings found, the sum of the minimum cost path, the time it took, and the shortest path itself.

[2] `https://github.com/faisal22/Christofides`

3.4 Genetic Algorithm

Genetic algorithms (GA) are search heuristics that attempt to mimic natural selection for many problems in optimization and artificial intelligence [4]. In a genetic algorithm, a population of candidate solutions is evolved over time towards better solutions. These evolutions generally occur through mutations, randomization, and recombination. We define a fitness function to differentiate between better and worse solutions. Solutions, or individuals, with higher fitness scores are more likely to survive over time. The final solution is found if the population converges to a solution within some threshold. However, great care must be taken to avoid being trapped at local optima.

We will now apply a genetic algorithm to the traveling salesperson problem [2]. We define a fitness function $\mathcal{F}$ as the length of the path. Supposed we have an ordering of the cities $A = \{x_1, x_2, ..., x_n\}$ where n is the number of cities. The fitness score for the TSP becomes the cost of the tour:

$$\mathcal{F}(A) = \left(\sum_{i=0}^{n-1} w(x_i, x_{i+1}) \right) + w(x_n, x_0)$$

The genetic algorithm begins with an initial, P_0, random population of candidate solutions. That is, we have a set of paths that may or may not be good solutions. We then move forward one time step. During this time step, we perform a set of probabilistic and statistical methods to select, mutate, and produce an offspring population, P_1, with traits similar to those of the best individuals (with the highest fitness) from P_0. We then repeat this process until our population becomes homogeneous.

The running time of genetic algorithms is variable and dependent on the problem and heuristics used. However, for each individual in the population, we require $O(n)$ space for storage of the path and during genetic crossover, the space requirement increases to $O(n^2)$. The best genetic algorithms can find solutions within 2% of the optimal tour for select graphs [9].

3.4.1 Genetic Implementation

A genetic algorithm was implemented, in Java, for this experiment. Genetic algorithms generally are parameter rich and as such, we explain critical settings before running the experiment. This implementation is also freely available on Github[3].

Genetic algorithms attempt to mimic real life evolution and are commonly used in artificial intelligence and optimization problems. Consider our traveling salesperson problem (TSP). An *individual* is a single solution to the TSP. A *population* consists of several individuals, all of which are different and need not be unique. Each individual is given a score based on a *fitness function*. In the TSP, the score is the length of the tour/path. The lower the score an individual has, the better of a solution it is.

Given our starting population, we evolve the population. This requires selecting two parents (from the population of individuals) and crossing them. We take some attributes from parent 1 and some from parent 2 to create a new child. This child is then scored and then placed back into the population. The more fit parents are more likely to reproduce and thus the next generation of children should be better off than the parents, that is, they are better solutions to the TSP. We can introduce *mutations*, that is, random events that change each individual (with regards to the TSP, it swaps the order of two cities at random).

Several parameters are used which drive the genetic algorithm:

- Population Size: This is number of individuals in our population
- Number of Evolution Iterations: Number of times to advance the population and create offspring/mutations
- Tournament Size: To create a child, we must have two parents. To select these two parents, we create a tournament consisting of *TOURNAMENT_SIZE* individuals. We then select the most fit individual from this tournament to become the parent.
- Mutation Rate: The probability that a given individual will incur a single mutation.
- Clone Rate: The probability that a child will be an exact copy of one of its parents.
- Elite Percent: The percent of the population to deem as elite individuals. Elite individuals have very high fitness scores (or low path costs) and have a higher reproduction rate.
- Elite Parent Rate: The probability that one of the parents (when creating a child) is elite.

[3] `https://github.com/ahaque/Genetic-TSP`

We continue to evolve the population until: (i) our population becomes the homogeneous or (ii) we reach a time limit. A population becomes homogeneous when there is not significant evolutionary progress. Let n denote the number of cities in the tour, let p denote the population size, and let $\bar{x}_i$ denote the average fitness score of the population after the i^{th} iteration.

$$\bar{x}_i = \sum_{b=1}^{p} \frac{1}{p} \left[\left(\sum_{a=1}^{n-1} w(x_a, x_{a+1}) \right) + w(x_n, x_0) \right]$$

The genetic algorithm terminates once the average fitness score between two iterations does not significantly change. Formally, we terminate when:

$$|\bar{x}_i - \bar{x}_{i+1}| \leq \epsilon, \forall i \in \mathbb{Z}$$

Our implementation terminates if all individuals are within one standard deviation of the mean fitness score or if the time reaches two minutes, whichever comes first. Once we terminate, the genetic algorithm returns the lowest cost solution from the population.

4. EVALUATION

4.1 Datasets and Results

We benchmark our algorithms using publicly available datasets. In addition, to test the scalability of the algorithms, we generated a synthetic dataset of 1000 cities. In all dataset names, the numeric digits represent the number of cities in the dataset. The datasets are as follows: GR17, FRI26, ATT48, WG59, SGB128, and G1000. All datasets except G1000 can be found online [3, 15].

Figure 5: Dataset and Optimal Path Cost: Inferred optimal costs are denoted by an asterisk (*)

Dataset	Cities	Optimal Cost
GR17	17	2085
FRI26	26	937
ATT48	48	10628
WG59	59	817*
SGB128	128	18538*
G1000	1000	18548*

Figure 6: Algorithm Runtime (Seconds)

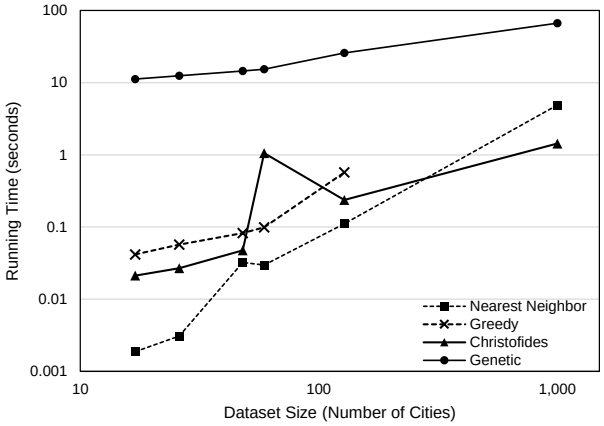
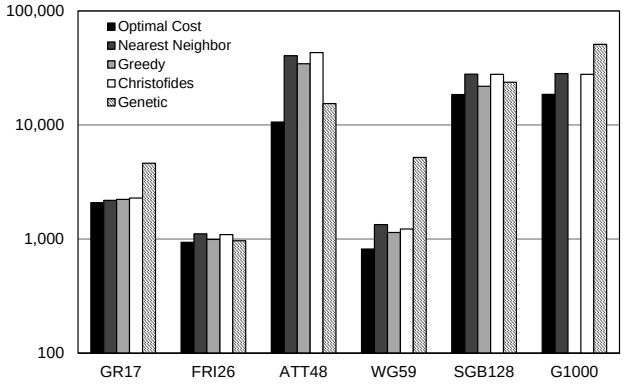
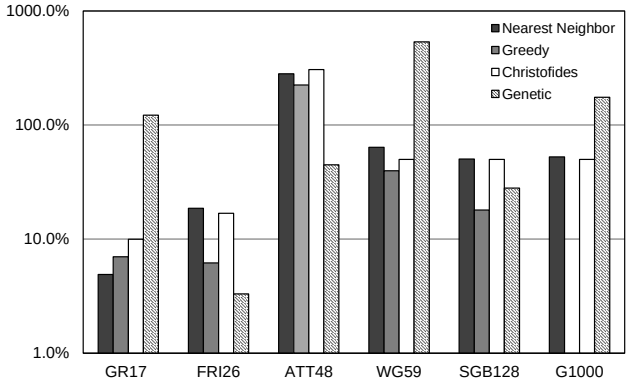


Figure 4: Experimental Results: Tour Length vs. Optimal

(a) Tour Lengths



(b) Percent Above Optimal Tour Cost



Not all datasets have an optimal tour. We use Christofides algorithm to infer a lower bound of the optimal tour since it is guaranteed to give us a 1.5 approximation. We assume Christofides gives us a worst case solution of 1.5 times the optimal. Then we divide the solution by 1.5 and arrive at an inferred optimal. The true optimal may be higher than our inferred optimal. In fact, the true optimal may not exceed the solution returned by Christofides algorithm. As a result, we have both a lower and upper bound on the optimal solution.

The G1000 dataset was generated by plotting 1,000 random points (x, y), in $\mathbb{R}^2$ with $x, y \in [0, 1000]$. Each point is then compared with each other point and the Euclidean distance is calculated. As a result, all distances satisfy the triangle inequality and this dataset can be classified as a metric TSP dataset. The running time for creating the distance matrix is $O(n^2)$ since we have n comparisons for n cities.

5. DISCUSSION

As we can see in Figure 4a, Nearest neighbor and greedy have similar solution costs. In Figure 4b, we can see that most algorithms return a solution under 20% over the optimal for small datasets and are often twice as much for larger datasets.

In terms of running time as shown in Figure 6, the best algorithm is Nearest Neighbor. However, in terms of optimal cost of solution, the best algorithm is Greedy. This is in line with our expectations, and alludes to the fact that different heuristics are better suited for different situations.

We found that Nearest Neighbor calculated a route for even the largest dataset (1000 cities) in 4 seconds, proving that this algorithm can quickly calculate a solution for even the largest traveling problems (for a salesmans purposes, that is). However, at that scale, the route for the solution was almost double the cost of the optimal solution. Thus, Nearest Neighbor is a poor choice of heuristic for extremely large datasets. Furthermore, this heuristic is best suited for smaller routes when any optimization gains are marginal in comparison to time complexity of other, more sophisticated heuristics.

As shown by Figure 4a, Christofides algorithm preforms fairly consistently in comparison to Nearest Neighbor and Greedy algorithms across all datasets. Where Christofides Algorithm really shines, however, is highlighted in Figure 6 when comparing the running time of the heuristics. On larger datasets, Christofides algorithm completes successfully and in noticeably shorter time than all other heuristics. Figure 4b further demonstrates that the percent above optimal is kept minimal as well for Christofides algorithm on larger datasets. Thus, Christofides heuristic is the desired algorithm to be used with larger datasets due to its advantages in accuracy and success rate in finishing quicker than other heuristics.

6. CONCLUSIONS AND FUTURE WORK

7. REFERENCES

[1] S. Arora. Polynomial time approximation schemes for euclidean tsp and other geometric problems. In *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on*, pages 2–11. IEEE, 1996.

[2] K. Bryant and A. Benjamin. Genetic algorithms and the traveling salesman problem. *Department of Mathematics, Harvery Mudd College*, 2000.

[3] J. Burkardt. Data for the traveling salesperson problem. `http://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html`, July 2011.

[4] S. Chatterjee, C. Carrera, and L. A. Lynch. Genetic algorithms and traveling salesman problems. *European journal of operational research*, 93(3):490–510, 1996.

[5] W. Cook. The traveling salesman problem. `http://www.tsp.gatech.edu/index.html`, 2009.

[6] L. Cowen and D. Rice. The traveling salesman problem. Course Notes for COMP 260 - Advanced Algorithms, 2009.

[7] F. Glover, G. Gutin, A. Yeo, and A. Zverovich. Construction heuristics for the asymmetric tsp. *European Journal of Operational Research*, 129(3):555–568, 2001.

[8] G. Gutin and A. Yeo. The greedy algorithm for the symmetric tsp. 2005.

[9] A. Homaifar, S. Guan, and G. E. Liepins. Schema analysis of the traveling salesman problem using

genetic algorithms. *Complex Systems*, 6(6):533–552, 1992.

[10] D. S. Johnson and L. A. McGeoch. The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*, pages 215–310, 1997.

[11] B. I. Kim, J. Shim, and M. Zhang. Comparison of tsp algorithms, 1999.

[12] E. L. Lawler, J. K. Lenstra, A. R. Kan, and D. B. Shmoys. *The traveling salesman problem: a guided tour of combinatorial optimization*, volume 3. Wiley Chichester, 1985.

[13] J.-S. Lin. Christofides' heuristic. Course Notes for IEOR 251 - Facility Design and Logistics, 2005.

[14] C. Nilsson. Heuristics for the traveling salesman problem. *Linkoping University*, pages 1–6, 2003.

[15] G. Reinelt. `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`.

[16] Z. Xu, L. Xu, and B. Rodrigues. An analysis of the extended christofides heuristic for the k-depot tsp. *Operations Research Letters*, 39(3):218–223, 2011.

APPENDIX

A. DETAILED EVALUATION TABLES

Figure 7: Algorithm Running Times (Milliseconds)

Dataset	Near	Greedy	Christofides	Genetic
GR17	1.87	41.71	21.20	11214.15
FRI26	3.08	57.20	26.85	12477.52
ATT48	32.19	82.13	47.61	14541.29
WG59	29.66	98.80	1058.20	15340.86
SGB128	111.03	570.42	237.01	25791.27
G1000	4887.68		1434.22	66619.82

Figure 8: Algorithm Solutions: Tour Costs

Dataset	**Opt**	**Near**	**Greed**	**Christo**	**Genetic**
GR17	2085	2187	2231	2293	4628
FRI26	937	1112	995	1095	968
ATT48	10628	40551	34490	43174	15397
WG59	817	1339	1142	1225	5201
SGB128	18538	27885	21877	27807	23734
G1000	18548	28284		27822	51042

Figure 9: Percent Above Optimal

Dataset	**Near**	**Greed**	**Christo**	**Genetic**
GR17	4.89%	7%	9.98%	121.97%
FRI26	18.68%	6.19%	16.86%	3.31%
ATT48	63.89%	39.78%	49.94%	536.6%
WG59	281.55%	224.52%	306.23%	44.87%
SGB128	50.42%	18.01%	50%	28.03%
G1000	52.49%		50%	175.19%