An Empirical Evaluation of Approximation Algorithms for the Metric Traveling Salesman 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 salesman 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 derived from artificial intelligence. In this paper, we perform an evaluation and analysis of cornerstone algorithms for the metric TSP. We evaluate the nearest neighbor, greedy, Christofides, and genetic algorithms. We use several datasets as input for the algorithms including several small datasets, two medium-sized datasets representing cities in the United States, and a synthetic dataset consisting of 1,000 cities to test algorithm scalability. We discover that the nearest neighbor and greedy algorithms efficiently calculate solutions for smaller datasets. 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 longer runtimes.

1. INTRODUCTION

Known to be NP-hard, the traveling salesman problem (TSP) was first formulated in 1930 and is one of the most studied optimization problems to date [14]. 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 TSP has broad applications including: shortest-path for lasers to sculpt microprocessors and delivery logistics for mail services, to name a few.

The TSP is an area of active research. In fact, several variants have been derived from the original TSP. In this paper, we focus on the metric TSP. In the metric TSP, all distances between cities satisfy the triangle inequality. That is, for three cities, $A, B,$ and C:

$$dist(A, C) < dist(A, B) + dist(B, C)$$

This simplification allows us to survey several cornerstone algorithms without introducing complex scenarios, specifically for Christofides. The remainder of this paper is organized as follows. In Section 2, we briefly review the first solutions and survey modern approaches and variants to the TSP. We describe the algorithms used in our experiment and outline key implementation details in Section 3. A description of the benchmark datasets and results of the experiment are detailed in Section 4. A discussion in Section 5 explains the findings and compares the performance of the algorithms. We then conclude and describe future work in Section 6.

2. BACKGROUND

An example TSP is illustrated 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, problem complexity, and running time.

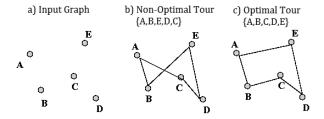Figure 1: Example TSP Input and Solutions

Figure 2 lists the number of edges and total possible number of tours for a specific dataset size. The number of possible tours is $(n-1)!/2$ since the same tour, with start point X and Y appears twice: once with X as the start node and once with Y as the start node.

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 William

---

Figure 2: TSP Complexity

Cities	Edges	Possible Tours
5	10	12
6	15	60
7	21	360
8	28	2,520
9	36	20,160
10	45	181,440
25	300	3.1e+26
50	1,225	3.04e+64
100	4,950	4.70e+155
n	$\sum_{i=0}^{n-1} i$	$(n-1)!/2$

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 for agricultural applications. The problem was then 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. The progression of solutions is shown in Figure 3. 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 applications to numerous fields. Examples among these applications include genome sequencing, air traffic control, supplying manufacturing lines, and optimization.

2.1 TSP Variants

Several variants of the original TSP exist. Some of these variants differ in their representation of cities and some differ by the constraints placed on city distances. We describe the Euclidean, Graphic, and Asymmetric TSP variants in this section.

2.1.1 Euclidean TSP

In the Euclidean TSP, the input is given as a list of coordinates describing the location of each city in $\mathbb{R}^2$ [1]. It is possible to extend this into higher dimensions as well. The alternative to giving a list of city coordinates is to give a distance/adjacency matrix describing the distances between each city pair. It is possible to derive the distance matrix given the city coordinates using Euclidean geometry. However, given the distance matrix, inferring the coordinates is not as straightforward and requires computation of a Gramian matrix and application of several matrix decomposition methods.

2.1.2 Graphic TSP

In recent years, the graphic TSP has attracted attention from the research community. The graphic TSP problem asks for the shortest tour that visits each vertex at least once. The current best approximation algorithm returns a solution within $13/9 \approx 1.444$ of the optimal [16]. For the past 30 years, Christofides had been the leading algorithm with a 1.5 approximation. Although the 13/9 approximation

is for a TSP variant, it is an important milestone for TSP approximations nonetheless.

2.1.3 Asymmetric TSP

All TSP variants described thus far have assumed an undirected graph, resulting in $cost(A, B) = cost(B, A)$. The asymmetric TSP introduces directed edges. As a consequence, algorithms with assumptions about reflexive distances may fail when $cost(A, B) \neq cost(B, A)$. Real world applications of this includes roads and highways – specifically the case of one-way streets, detours, and alternate routes. Additionally when modeling vehicle energy usage, the geographical topology makes traveling uphill cheaper than downhill despite traveling between the same two points.

2.2 Related Heuristics

In this section, we briefly summarize current approaches for the metric TSP. We describe the 2-approximation algorithm and survey an advanced technique based on artificial intelligence.

2.2.1 2-Approximation Algorithm

A simple two-approximation solution can be achieved by constructing a minimum spanning tree (MST) for the input TSP graph and creating a list of vertices (no duplicates) from a pre-order walk of the MST [20]. This list of vertices becomes a Hamiltonian cycle and is a solution to the TSP. This algorithm completes in polynomial time and is is guaranteed to return a two-approximation solution (see [20] for proof).

2.2.2 Ant-Colony Approach

Classified under the umbrella of nature-inspired algorithms, the ant colony system (ACS) is a distributed swarm intelligence algorithm that has a set of cooperating agents called *ants* that attempt to solve the TSP. This method attempts to mimic how ants find the shortest path from their home to food in real life. In ACS, ants communicate by depositing *pheromones* on the graph edges as they build TSP solutions [7]. Over time, the shorter paths build larger amounts of pheromones. The solution to the TSP is the path with highest pheromone levels which visit every city.

This algorithm is able to run on both symmetric and asymmetric TSPs. For the symmetric TSP with 170 cities or less, the ACS algorithm finds the optimum solution. For the asymmetric TSP with 170 cities or less, it found a solution within 0.40% of the optimal and outperforms genetic algorithm approaches [7].

3. ALGORITHMS

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

3.1 Nearest Neighbor

The nearest neighbor algorithm is the simplest approach to TSP. The general idea is to start with any vertex v and continue to 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.

Figure 3: History of 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
2004	D. Applegate, R. Bixby, V. Chavatal, and W. Cook	24,978 Cities
2006	D. Applegate, R. Bixby, V. Chavatal, and W. Cook	85,900 Cities

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 algorithm produces a path that is on average, 25% longer than the optimal path [12].

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 a time limit for execution time. After each file is imported from a text file (see Section 4.1), an adjacency matrix is constructed and stored in memory for the remainder of the algorithm.

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 requires $O(n^2)$ 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 Kruskal's algorithm to give an approximate solution to the TSP [13]. The algorithm forms a tour of the shortest route and can be constructed if and only if:

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

2. The selected edge (before being appended to the tour) does not increase the degree of any node to be more than 2.

The algorithm begins by sorting all edges from least weight to most heavily weighted. After the edges are sorted, the least heavily-weighted edge is selected and it is added to the tour if it does not violate the above conditions. The algorithm continues by selecting the next least-cost edge and adding it to the tour. This process is repeated until all vertices can be reached by the tour. The result is a minimum spanning tree and is a solution for the TSP. The runtime for the greedy algorithm is $O(n^2 log(n))$ and generally returns a solution within 15-20% of the Held-Karp lower bound [17].

3.2.1 Greedy Implementation

The greedy solution to TSP differs from the nearest neighbor heuristic because it uses a Kruskal's approach to the problem. Instead of starting at a random node and building a tour using the nearest neighbor 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 the input datasets from the files and constructs a distance matrix corresponding to 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 doesn't 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, and returns the path cost of the tour.
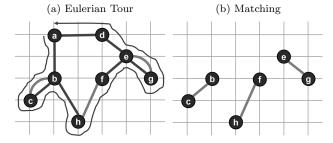
3.3 Christofides Algorithm

The Christofides Algorithm is a heuristic algorithm and finds a near-optimal solution. It is a polynomial 3/2 approximation algorithm for the TSP [15]. Its primary objective is to find a Hamiltonian cycle of minimum length.

The algorithm begins by constructing a minimum spanning tree M on $G = (V, E)$. 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 will take polynomial time. 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. This is demonstrated in Figure 4a.

---

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

Figure 4: Christofides Algorithm [20]

(a) Eulerian Tour   (b) Matching

We pick any initial vertex in the Eulerian tour while noting the order of visited vertices. Call this list of vertices S. While traversing the Eulerian tour, if we arrive at a node that has already been visited, we do not add it to S. Once we reach the starting vertex again, S now describes a Hamiltonian path and as a result, S is a solution to the TSP.

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 [21].

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 finding a Hamiltonian cycle of minimum length. The implementation by Bjorn Harald Olsen and Oscar Tackstrom has been proven to successfully achieve a minimum cost path solution to the TSP. We implement this algorithm in Java for this experiment.

The source code for this implementation can be found on Github[2]. The algorithm begins by reading the input 2-dimensional adjacency matrix with city-distances at each matrix value. 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 by vertex matching. Our implementation finds a minimum cost perfect matching of all odd-degree vertices in polynomial time. An example matching is shown in Figure 4b. We then union the MST and the perfect matching. Call this graph G^*. A Eulerian tour is then constructed on G^* and subsequently traversed while constructing a *short-cut* list of vertices by ignoring previously seen vertices. This *short-cut* version is a Hamiltonian cycle. We then sum the distances return the total tour cost. Additionally, the path is returned as an ordered list of vertices.

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

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

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 salesman problem [2]. We define a fitness function $\mathcal{F}$ as the length of the tour. 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 $d(x, y)$ denote the distance from x to y.

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

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. For genetic crossover, the space requirement remains $O(n)$. The best genetic algorithms can find solutions within 2% of the optimal tour for certain graphs [10].

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 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. Lower tour costs imply better solutions and higher fitness scores.

Given our starting population, we evolve the population. This requires selecting two parents from the population of individuals and crossing them. To cross two parents, we take some attributes from parent 1, some attributes from parent 2, and create a new child with the combined set of attributes. This child is then scored and placed back into the population. Parents with higher fitness scores are more likely to reproduce and thus the next generation of children should have better fitness scores than the parents. Ideally, the children should be better solutions to the TSP. We can introduce *mutations* that randomly change each individual by swapping the order of two cities in the tour. Some mutations will be beneficial while some will lower the fitness score.

Several parameters are used which drive the genetic algorithm:

- Population Size: This is number of individuals in our population. For this experiment, we used 100 individuals.

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

- Number of Evolution Iterations: Number of times to advance the population and create offspring/mutations.

- Forward Progress Epsilon: The threshold for the genetic algorithm to terminate (see Equation 3).

- 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 classified as elite individuals. Elite individuals have very high fitness scores (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.

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, let d be the cost function for calculating distance, and let $\bar{x}_i$ denote the average fitness score of the population after the ith iteration:

$$\bar{x}_i = \sum_{b=1}^{p} \frac{1}{p} \left[\left(\sum_{a=1}^{n-1} d(x_a, x_{a+1}) \right) + d(x_n, x_0) \right] \qquad (2)$$

The genetic algorithm terminates once the average fitness score $\bar{x}$ between two iterations does not significantly change. Formally, we terminate when:

$$|\bar{x}_i - \bar{x}_{i+1}| \leq \epsilon, i \in \mathbb{Z} \qquad (3)$$

We call epsilon in Equation 3 the *forward progress epsilon*. If the population is no longer evolving, we end the algorithm. Alternatively, if the running time reaches 2 minutes, the algorithm terminates. Once the algorithm terminates, 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. Additionally, to test the scalability of the algorithms, we generated a synthetic dataset consisting 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, 19]. The ATT48 and SGB128 datasets represent real-data consisting of locations of cities in the United States. A visual representation of the SGB128 dataset in the 2D plane can be found in Figure 5.

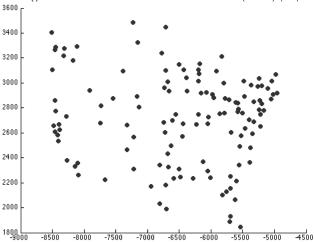Figure 5: SGB128 Dataset Visualization (USA) [19]

Figure 6: Dataset and Optimal Path Cost: Inferred optimal costs are denoted by an asterisk (*)

Dataset	Lower Bound	True Optimal	Upper Bound
GR17	1529	2085	2293
FRI26	730	937	1095
ATT48	8850	10628	13275
WG59	817	817*	1225
SGB128	18538	18538*	27807
G1000	18548	18548*	27822

$$\frac{\text{Christofides}}{1.5} \leq \text{True Optimal} \leq \text{Christofides} \qquad (4)$$

Not all datasets have a known optimal tour. When this is the case, we use Christofides algorithm to infer a lower bound of the optimal tour since Christofides 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. Additionally, 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 as shown by Equation 4. The known and inferred optimal solutions are listed in Figure 6.
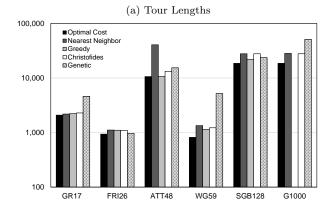
4.2 Dataset Generation

The G1000 dataset was generated by plotting 1,000 random, uniformly distributed 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. The output (and subsequently the input for each algorithm) is an adjacency matrix with city distances at each matrix entry.

5. DISCUSSION

As we can see in Figure 7a, the nearest neighbor and greedy algorithms have similar solution costs. In Figure 7b, we can see that most algorithms return a solution under 20%

Figure 7: Experimental Results: Tour Length vs. Optimal

(a) Tour Lengths

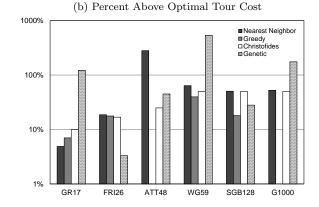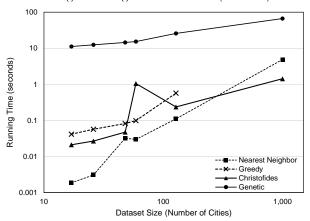(b) Percent Above Optimal Tour Cost

Figure 8: Algorithm Runtime (Seconds)

over the optimal for small datasets and are often twice the optimal for larger datasets.

In terms of running time (Figure 8), 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 the 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 larger traveling salesman problems. However, at that scale, the route for the solution was almost double the cost of the optimal solution. Thus, the nearest neighbor is a poor heuristic choice 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 in Figure 7a, Christofides algorithm performs fairly consistently, in comparison to the nearest neighbor and greedy algorithms, across all datasets. Highlighted in Figure 8, the running time of Christofides is generally a fast algorithm for small to medium datasets and is the fastest algorithm for the G1000 dataset. This suggests that for larger datasets, if running time is a concern, then the Christofides algorithm should be used. Figure 7b further demonstrates that Christofides algorithm maintains a smaller *percent above optimal* than the other algorithms. From this, we can see

that Christofides algorithm has high accuracy and faster runtimes than other heuristics, especially for larger datasets.

Since the genetic algorithm does not have an optimality guarantee, its results vary across datasets. It can perform very well – as demonstrated on the FRI26 dataset where it returned a solution with 3.31% of the optimal. On the other hand, it returned the worst solution for three out of the seven datasets. For both the GR17 and WG59 datasets, the genetic algorithm returned a solution of an order of magnitude greater than the optimal. Due to the randomized nature of genetic algorithms, it may be possible to become stuck at a local optimum. This may have been the case for the G1000 dataset. The other three algorithms were within $2\times$ the optimal.

From a running time standpoint, the genetic algorithm takes the longest time to reach a solution. For most datasets, the algorithm required ten seconds to one minute to complete. This can be attributed to the *forward progress epsilon*, that is the threshold at which we declare a population as "no longer evolving." The smaller the epsilon, the longer the runtime. Although genetic algorithms have longer runtimes, it may be possible to find a solution better than the other three algorithms.

6. CONCLUSION

Most of our algorithms attempt to solve the TSP in a linear fashion. Originating from artificial intelligence, the genetic algorithm is very different compared to greedy, nearest neighbor, and Christofides. Literature suggests that the best algorithms focus on iteration and convergence to find optimal tours – something genetic algorithms attempt to achieve. For example, the Large Step Markov Chain [11] relies on Markov chains to find convergence of many paths to form a global optimum and several papers cite Markov Chains as the best known solution to TSP. Recent studies include using adaptive Markov Chain Monte Carlo algorithms [18]. Many of these extend the Metropolis algorithm [9], a simulated annealing algorithm which attempts to mimic randomness with particles as the temperature varies. This further supports our conclusion that algorithms inspired from artificial intelligence perform well for finding solutions for the TSP. However, these may not be suitable when a guarantee is required.

As future work, we believe it would be an interesting exercise to add an iterative component to our basic algorithms.

For example, in nearest neighbor, it would be interesting for the algorithm to evaluate its path efficiency in real time and backtrack if the path exceeds a certain threshold of costs – similar to graph search algorithms. Of course, for this kind of iteration, it would be required to know the optimal cost. Our work relied on the Christofides algorithm to assume a lower bound and a similar algorithm could be used to infer an optimal solution.

In this paper, we surveyed several key cornerstone approaches to the traveling salesman problem. We selected four well-known algorithms and tested their performance on a variety of datasets. Our results suggest that genetic algorithms (and other approaches from artificial intelligence) are able to find a near-optimal solution. However, these approaches do not provide guarantees like Christofides and the two approximation algorithms.

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] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 1(1):53–66, 1997.

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

[9] H. Haario, E. Saksman, and J. Tamminen. An adaptive metropolis algorithm. *Bernoulli*, pages 223–242, 2001.

[10] 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.

[11] I. Hong, A. B. Kahng, and B.-R. Moon. Improved large-step markov chain variants for the symmetric tsp. *Journal of Heuristics*, 3(1):63–81, 1997.

[12] 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.

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

[14] 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.

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

[16] M. Mucha. \ frac {13}{9}-approximation for graphic tsp. *Theory of Computing Systems*, pages 1–18, 2012.

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

[18] F. Qiu, J. Zhang, and H. Yan. An adaptive markov chain monte carlo algorithm for tsp. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 1, pages 439–442. IEEE, 2008.

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

[20] K. Wayne. Approximation algorithms. Princeton University COS 423 Lecture Notes, 2001.

[21] 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 9: 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 10: Algorithm Solutions: Tour Costs

Dataset	Opt	Near	Greed	Christo	Genetic
GR17	2085	2187	2231	2293	4628
FRI26	937	1112	1103	1095	968
ATT48	10628	40551	10654	13275	15397
WG59	817	1339	1142	1225	5201
SGB128	18538	27885	21877	27807	23734
G1000	18548	28284		27822	51042

Figure 11: Algorithm Solutions: Percent Above Optimal

Dataset	Near	Greed	Christo	Genetic
GR17	4.89%	7.00%	9.98%	121.97%
FRI26	18.68%	17.72%	16.86%	3.31%
ATT48	281.55%	24.91%	24.91%	44.87%
WG59	63.89%	39.78%	50.00%	536.60%
SGB128	50.42%	18.01%	50.00%	28.03%
G1000	52.49%		50.00%	175.19%