Analysis of Natural Metaphor Algorithms to Solve Traveling Salesman Problem

Krystal Lee
leex5444@umn.edu

May 11, 2016

Abstract

The traveling salesman problem is a popular combinatorial optimization touring problem that has an easy to understand nature, and hard to solve nature. Specifically, NP-hard. Its hard nature has made it a challenging problem that many are interesting in solving. Additionally, its easy to grasp nature has made it a natural testing ground for many different algorithms and heuristics. Among those algorithms are those inspired by the natural world. For example, the way ants obtain food, or populations evolve over time. These simple observations of the world around us has led to the creation of algorithms that give a unique perspective on difficult problems, such as TSP. In this paper we examine three of these algorithms: simulated annealing, genetic algorithm, and ant colony algorithm. We also create a new algorithm, 3NM, composed of different behaviors of the previously listed algorithms. Finally, we test all the algorithms on their ability to solve TSP. Our goal is to highlight the interesting way natural metaphor algorithms approach TSP.

1 Introduction

Image a salesman who wants to travel to n different cities before returning home. He only needs to visit each city once, and knows the distance between each pair of cities i and j. He wants to be as efficient as possible, and thus wants to figure out the shortest round-trip route, or tour, to take. How can he do this? This classic problem is known as the Traveling Salesman Problem (TSP), and although at first glance it may look straightforward and easy to solve, it has created many challenges for the mathematical and computer science communities!

An obvious way to approach the problem is to model the problem space as a graph problem. In this case each city would be a node, and every two nodes would be connected by an edge that represented the distance between them [8]. Therefore given a graph G, the nodes would form a set $c = \{c_1, c_2, ..., c_n\}$ that represents all the cities, and each pair of nodes $\{c_i, c_j\}$ would have a cost defined by the distance d_{ij} between them.

Depending on the type of graph G is, TSP can be represented as either symmetric or asymmetric. In the case of an asymmetric TSP the graph G is directed, and thus, the direction and cost of edges come into play. We have to consider possibilities where an edge only goes in one direction, or where two edges going in different directions have different costs. In other words, there exists the possibility that at least one edge in G has $d_{ij} \neq d_{ji}$ [2]. The more simpler case, the symmetric TSP, is where we say that G is undirected, and we can therefore assume that for all edges $d_{ij} = d_{ji}$. This

greatly reduces the possible orderings of the nodes, and will be the version of TSP we will focus on in this paper.

Now that we understand the basics of our problem we have to consider how to accomplish our goal. Well, essentially our goal is to find the minimum Hamilton cycle, that is, to find the shortest closed path that visits each node in c once [2]. To do this we want to find the ordering π of the nodes which minimizes equation 1:

$$\sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)} \tag{1}$$

This graphical model is a representation that creates an easy way to visualize the problem. There are several alternative representations such as the linear programming representation which is very well studied, and can be used with different constraints such as circuit packing and MTZ [16]. There are also many possible variations of the problem ranging from considering TSP within a time window to considering the nodes as black or white [16]. We can already see that to simply approach describing the problem we already have many different possibilities, and a lot of room for creative thinking. The problem description presented by equation 1 will be the one we will be focusing on in this paper since it serves an often used foundational basis to TSP.

As anyone can see the problem itself is very easy to describe and understand; there is a reason why TSP is a problem often introduced to college students in freshmen level mathematics and computer science courses! However, the problem is actually difficult to solve optimally since it belongs to the class of NP-complete problems. NP-complete problems are ones that have nondeterministic polynomial time, and are considered hard due to having no fast algorithms that solve them. This is not so much an issue with small problem instances, but as the problem instance grows the algorithm running time grows exponentially making it extremely difficult to find optimal solutions to large problem instances [15].

Due to their lack of optimal algorithms there has been a lot of interest in finding solutions to NP-complete problems. This allure is even more amplified by the fact that it is believed if we find a solution to one NP-complete problem it can be used for all other problems in the class. Even if we can never find a true solution to NP-complete problems there is still a great desire to find, and analyze, algorithms that work on these problems.

In the hunt for new algorithms, TSP is an appealing problem to focus on since it is so easy to understand and describe. We do not have to worry about keeping track of a problem description that has many complicated ideas therefore we can instead emphasis creating new, and unique, ways to approach the problem. TSP can essentially be boiled down to a problem where given a set of nodes, and the costs between each pair of nodes, find the shortest Hamilton cycle. This simple abstract view of the problem makes considering it in different viewpoints easy. For example, we could view it as a problem to minimize the route for ants to get to their food sources.

The flexibility of TSP means that it has many different real world applications. Although the obvious application is for route planning, it can be applied to other areas in math and computer science. Additionally, it can be generalized to different fields of interest such as operations research, genetics, engineering, and electronics [16]. The fact that TSP is such an easy to understand problem, and its scope can be expanded to the natural world, such as with the priorly described ants metaphor, makes it an extremely appealing problem to analyze for many people. Thus, it is easy to see why TSP, and the many different algorithmic approaches to solving it, is interesting.

2 Literature Review

Although TSP has a somewhat unknown origin, its formal definition dates back to the 1930s. Since then it has served as the testing ground for many different algorithms and ideas, and is thus one of the more significant and interesting combinatorial optimization problems to study [12]. Among the purposed algorithms are exact solvers that guarantee an optimal solution, and non-exact solvers that do not guarantee an optimal solution but can perform faster [2]. Of the non-exact solvers there is an interesting set of meta-heuristic algorithms that take their inspiration from the natural world. These algorithms include simulated annealing, genetic, and ant colony algorithms. In this paper we examine these three different natural metaphor algorithms, and their unique approach to solving TSP.

Starting in the 1950s there was an increased desire to find solutions to TSP. This was largely due to efforts from the RAND corporation which contained a group of mathematicians focused on optimization problems [4]. In 1954 George Dantzig, Delbert Fulkerson, and Selmer Johnson published a paper where they presented a 49-instance TSP as a linear programming problem, and used a cutting plane method to solve it optimally [5]. This was truly a breakthrough in the history of TSP, and the paper spearheaded the movement towards finding exact solvers using linear programming. Exact solvers, compared to non-exact solvers, have the harder task of having to generate both a lower and upper bound on the problem instance [8]. This is often done by using a relaxation to replace the original problem with an easier one to solve. Some examples of relaxations used on TSP are assignment relaxation, n-path relaxation, and the linear programming relaxation [8]. These relaxations of TSP can then be solved using the earlier mentioned cutting plane method along with other exact solver methods, such as, interior point, branch-and-bound, and branch-and-cut [2]. One example of an exact solver that uses linear programming relaxation and the branch-and-cut method is Concorde, a program that in 2006 optimally solved the largest TSP instance of 85,900 cities [9].

Although being able to optimally solve TSPs with 85,900 cities or less is amazing it is still not ideal when considering the fact that the world contains millions of cities. Exact solvers unfortunately are not at the point where they can guarantee solutions for large instances of TSP with hundreds of thousands of cities or more. Instead, we must turn to non-exact solvers which also provide the ability to obtain faster solutions at the expense of not being able to guarantee optimal solutions [2]. One type of non-exact solvers are approximation algorithms which can find solutions to optimization problems that are relatively, and provably, close to optimal. The most popular algorithm of this type for solving TSP is Christofides' algorithm which was presented in 1976, and guaranteed a way to find a solution strictly less than 3/2 a factor within the optimal solution [3].

Another type of non-exact solvers are heuristic algorithms which, unlike approximate algorithms, only promise a feasible solution that may or may not be good. The appeal of heuristic algorithms are their speed, and their interesting ways of approaching problems. Since they do not have to guarantee a good solution, it is possible for anyone to come up with a heuristic and test it. This has led to the creation of many unique heuristic algorithms. Tour construction and tour improvement algorithms are two common types of heuristic algorithms. Tour construction algorithms stop when they find a solution, and include algorithms that use the nearest neighbor, greedy, and insertion heuristics which usually are within 10-15% optimality [15]. Tour improvement algorithms improve on tour construction solutions by often using 2-opt and 3-opt local searches. Examples of these algorithms include simulated annealing and genetic algorithms, both of which we will examine in this paper [15].

Simulated annealing is a meta-heuristic randomized local search algorithm, and performs as a slower version of the hill climbing algorithm with the added benefit of being able to deal with local optima in the search space [2]. It was developed in the early 1980s independently by Kirkpatrick et. al and Cerny, and takes its inspirational from metallurgy [1].

In metallurgy the process of annealing is used to heat materials at high temperatures, and then gradually cool them to alter their chemical properties [**?**]. This idea is adopted by the simulated annealing algorithm which uses a *temperature* control parameter to model the slow cooling process of annealing [12]. The *temperature* is initially set to a high value which mimics the behavior of atoms in high heat, and allows for more randomization at the beginning of the search. This high randomization at the start of the search increases the chance of making downhill movements, which can avoid local optimums. In this way, simulated annealing takes ideas from both random walk, and hill climbing, to allow for both efficiency and completeness [17].

Simulated annealing works great for large-scale optimization tasks, and in fact, one of the first problems it was applied to was TSP [12]. Unfortunately, the standard simulated annealing algorithm has some issues improving its quality-time trade-off, and can perform very slowly compared to simple hill climbing [7]. It is therefore often used with other heuristics to try and optimize its ability to find better solutions while minimizing its search time. One example of such an algorithm is list-based simulated annealing which uses a list-based cooling schedule to control the decrease in temperature [18]. It was presently recently in an article for its applications on TSP by Zhan et. al in February of 2016. The fact that simulated annealing, and its many variations, are still currently being testing on TSP more than 30 years after its creation shows how heuristics that adopt ideas from the natural can result in interesting, and beneficial algorithms.

Another example of an algorithm which adopts ideas from the natural world is the genetic algorithm. The algorithm takes cues from biological evolution, and was pioneered by Alex Fraser in 1957 [2]. It uses the simple ideas of natural selection via genetic materials to produce, ideally, fitter offspring. Instead of only keeping track of one possible solution, like simulated annealing, genetic algorithm keeps track of many solutions known as a *population* of individuals. Genetic algorithm also injects more randomization into its solutions than simulated annealing. During each generation, i.e. iteration of the algorithm, it evolves the population by modifying the individuals in it with the hope that this allows the population to eventually fit better with the environment, i.e. the problem space.

Genetic algorithms have a long history of being used for optimization problems that dates back to the 70s. Using a genetic algorithm with TSP is often done by implementing different types of crossover routines, measures of fitness, and mutation routines to try and obtains solutions that are close to optimal [2]. For crossover, two individuals with good fitness are chosen to become parents. The algorithm creates one or more children from their information which would hopefully increase the overall fitness of the population. Additionally, to avoid stagnation from lack of diversity the algorithm implements a mutation routine that modifies random individuals. Often times genetic algorithms also use local optimization algorithms on individual solutions to obtain more competitive results [12]. This is because, similar to simulated annealing, it has some run time issues.

Another natural metaphor algorithm with roots in biology is the ant colony algorithm, which quite literally uses the idea of ant colonies to solve problems. Real ants are capable of finding new shortest paths from a food source to their nest through the use of pheromone trails [6]. A similar process can be applied to solve TSP. The idea is to put artificial ants at random cities, and at each time step they move to new cities which creates a pheromone trail on the edges. Once the

ants have completed a trail the one with the shortest trail modifies the edges resulting in a higher rate of pheromone, and more attraction for other ants to go to the trail which in turn leads to an even higher rate of pheromone on the trail. This positive feedback process determines how the ants move. The algorithm also has the benefit that unlike real ants, artificial ants can be improved upon to determine how far cities are, and have memory of cities visited [6]. Since the algorithm uses trail mediated communication it increases chances of finding an optimal solution.

Ant colony algorithm is a newer algorithm compared to the others previously discussed, and was really only established in the 90s. However, its design naturally lends itself to TSP where the cities are the food sources, and the paths between them are pheromone trails. Ant colony has shown good performance with TSP, especially for small problems where it can quickly provide optimal solutions [15]. So once again, we see another example of a natural metaphor algorithm that takes an unique approach to its behavior, and is rewarded with positive results in solving TSP. All three of these natural metaphor algorithms show why heuristic algorithms are so appealing. They give a unique approach to solving TSP, and take ideas that have been proven in science. Seeing as the first solution to TSP presented way back in 1954 was discovered by viewing the problem in a unique way these sort of heuristic algorithms definitely provide an interesting direction that may lead to the next big breakthrough.

3 Project Approach

Of the proposed solutions to TSP are ones that use the idea of a natural metaphor to come up with behavior. For example, the simulated annealing, genetic, and ant colony algorithms. To solve TSP, we want to find the shortest tour between a set of cities. In this paper we will analysis the ability of natural metaphor algorithms to solve TSP, and test a new algorithm inspired by their behavior.

Simulated Annealing Algorithm

The algorithm starts off with a random best tour. While *temperature* is above a *limit* value it checks if a neighboring tour is a better tour. It does this by examining the energies, i.e. lengths, of the tours. If the energies indicate the neighbor is better, it sets it to the current best tour. Otherwise, it only sets based on a probability of the *temperature* and energies. The lower the *temperature*, or worse the energies, the less likely the neighbor will be picked. The algorithm then decreases *temperature* based on *coolRate*, so as it proceeds it is less likely to accept worse tours.

SIMULATED-ANNEALING
1. Set *temperature*, *limit*, and *coolRate*
2. Generate a new current tour
3. **while** *temperature* > *limit* **do**
   4. Get neighbor tour
   5. **if** ENERGY(neighbor) - ENERGY(current) > 0 **then** $prob = 1$
   7. **else** $prob = e^{ENERGY(neighbor) - ENERGY(current)/temperature}$
   8. **if** *prob* > random value **then** set current to neighbor tour
   9. Lower *tempearture* based on *coolRate*

end

Genetic Algorithm

The following algorithm starts with a new population of a given *size*. As long as we have not

reached some max *generation* value, it will keep on evolving the population. In this way it is similar to simulated annealing which starts off with a new tour, and keeps on going until the *limit* value. However, this is where genetic algorithm gets more complicated. To evolve the population the algorithm checks for elitism, performs crossover, and possibility mutates members of the population. This is all done in the hopes of gaining more diversity without introducing too much randomization.

GENETIC
1. Set *size*, *max*, *rate*, *elitism*
2. Create new population of *size*
3. **for** *1 to max generations* **do**
   4. **if** *elitism* **then** save best individual as first tour
   5. **for** *rest of population* **do**
      6. Select two parent tours randomly
      7. Crossover parents and to get child tour
      8. **if** random value $<$ *rate* **then** mutate child
   end
   9. Add child to population

end

Ant Colony Algorithm

The following ant colony algorithm starts off with a new ant colony, i.e. "population", of a given *size* similar to genetic algorithm. It then initializes the pheromone trails of the ants to be some default value. As long as a consistent best tour hasn't been found, the algorithm updates the ants movements based on the pheromone trails, which are then similarly updated to reflect the ants movements. In this way the pheromones on ideal paths should become stronger over time, and pheromones on bad paths should become weaker till they evaporate.

ANT-COLONY
1. Set *size*, *value*, *count*
2. Create new ant population of *size*, and randomize their starting tours
3. Get best tour from ant population
4. Initialize pheromone trails
3. **while** *count* $<$ *value* **do**
   4. Update ant population based on pheromone trails
   5. Update pheromone trails
   6. Get best tour of updated ant population
   7. **if** best tour is same **then** increment *count* **else** *count* $= 0$

end

3NM Algorithm

The following algorithm describes one I devised off of the three previously described natural metaphor algorithms. I designed it trying to keep in mind what behavior in each algorithm makes it interesting. Starting off I noted many similarities between the two biology metaphor algorithms. Both genetic and ant colony algorithm start with a "population" that goes off, and tries to collectively find the best tour. It does this by "mutating" to the attributes of the individuals in the population, i.e. the genes in genetic algorithm or pheromones in ant colony. This is the main basis

of 3NM which starts with an ant population that does crossover with the pheromone trails of its parents. It also incorporates ideas of simulated annealing in that instead of evolving the current generation it probabilistically replaces one of the parents based on a *temperature* and *coolRate*.

3NM
1. Set *size*, *max*, *temperature*, *coolRate*
2. Create new ant population of *size*, and randomize their starting tours
3. Get best tour from ant population
4. Initialize pheromone trails
5. **for** *1 to max generation* **do**
   6. Select two parent ants randomly
   7. Crossover parent pheromone trails and get child ant pheromone trail
   8. **if** energy of child is better than either parents **then**
   replace $minParent = \min(\text{ENERGY(parent1), parent(2))}$ with child
   9. **else** $prob = e^{ENERGY(minParent)-ENERGY(child)/temperature}$
   10. **if** $prob >$ random value **then** replace $minParent$ with child
   11. Update pheromone trails
   12. Lower *temperature* based on *coolRate*

end

4 Experiment

The experiment was conducted by running C# code on a Windows 8 machine through Microsoft Visual Studio. To set up the TSP problem space, online code published by Tony R. Martinez for Brigham Young University was used [13]. It provided the basic framework for the experiment, and an useful GUI interface which can be seen in **figure 1**.

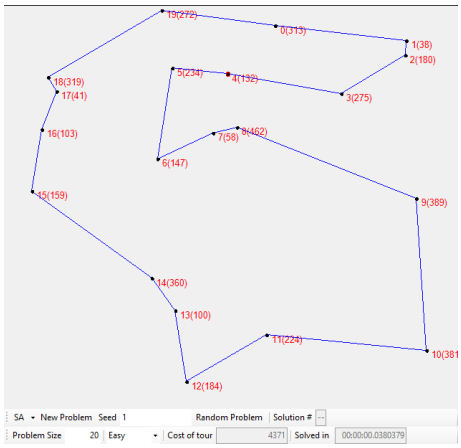

Figure 1: Example output of program using simulated annealing algorithm

The code already came implemented with the idea of cities, defined by their coordinates, and tours, defined as a list of cities. It also included pre-generated seed problems, the ability to alter

the problem size by increasing number of cities, and ability to calculate cost of the tour. I added additional functionality to calculate the time of algorithms, and of course the algorithms themselves.

For the experiment I first ran "initialization" tests on each of the three natural metaphor algorithms to determine what parameter values would be ideal to use. I defined being ideal as a value that minimizes cost without impacting time too much. I only ran 3 different levels (low, medium, high) for each parameter value so naturally choosing which value to use ended up being a rather subjective process. For each of the algorithms I obtained results by using a controlled problem seed of 1, and size 20, with differing values for the different control parameters.

Simulated Annealing Algorithm (SA)

I used the SIMULATED-ANNEALING pseudocode in **section 3** along with a C# adapted version of the SimulatedAnnealing.java code provided online by Lee Jacobson for this experiment [10]. Since simulated annealing keeps track of one solution at any time, the only structures used were tours, i.e. lists of cities, to keep track of the order of the cities. In simulated annealing the main parameters that control the performance of the algorithm are *temperature*, *coolRate*, and *limit*, which have all been defined in the provided SIMULATED-ANNEALING pseudocode. Essentially, *temperature* impacts how close to hill climbing or randomized walk simulated annealing performs, and *coolRate* controls how quickly it changes behavior. Finally, *limit* tells the algorithm how long to continue.

For the initial value of *temperature* I wanted it to be decently high to allow for more randomization at the start, and for the both *coolRate* and *limit* I set them decently low so that the algorithm would run for awhile, and cool slowly. To compare performance I tried both increasing and decreasing the control values by a factor of 10. An example of one of the trial results is seen in **figure 1**, and the results of the experiment in **table 1**. In the table the average cost and time values were obtained over 3 trials. Additionally the minimum average cost and time are in green font, while the maximums are in red.

Simulated Annealing	**Control Parameters**				
	temperature	cooling rate	limit	**avg. cost**	**avg. time**
control experiment	1000	0.001	0.01	**4471.33**	**0.063**
change in	-	0.0001	-	**4049.00**	**0.613**
cooling rate	-	0.01	-	**5080.00**	**0.009**
change in	100	-	-	**4469.33**	**0.043**
temperature	10000	-	-	**4380.33**	**0.062**
change in limit	-	-	0.001	**4367.33**	**0.063**
	-	-	0.1	**4316.33**	**0.039**

Table 1: Simulated annealing algorithm performance from change in constraints

From **table 1** we can see that changes in the cooling rate seem to have the biggest impact on resulting cost and time with inverse minimizing and maximizing of cost vs. time. However, from this limited experiment we cannot say that cooling rate always impacts performance of simulated annealing the most. It might have been the results of the initial control values with *coolRate* having the largest change of impact at the specific range created by the factor of 10 above and below it.

Regardless, the information is enough for simply trying to pick decent parameter values. Although it might be appealing to pick 0.001 or 0.01 for *coolRate* it simply isn't worth the cost-

time tradeoff, in either case. Instead, I stuck with $coolRate = 0.001$. Additionally, I picked $temperature = 100$ since the average cost is slightly better than the control, without negative impact to time. Lastly I picked $limit = 0.1$, another easy choice as it lowers both cost and time.

Genetic Algorithm (GA)

For the genetic algoirthm I used the GENETIC pseudocode in **section 3** along with a C# adapted version of the TSP-GA.java code provided online by Lee Jacobson for this experiment [11]. The genetic algorithm was more involved than simulated annealing. Along with the tour structures to keep track of cities I created a population list structure to keep track of tours, e.g. individuals in population. For this part, along with the rest of the experiment, I allowed elitism. Thus, the best individual of each generation is saved. Additionally, crossover was achieved by picking parents from the 5 most fit individuals, and mutation was achieved by swapping two cities in a random tour.

Genetic algorithm had more potential parameters to test than simulated annealing, however, I decided to focus on picking values for initial population size, number of generations, and mutation rate. For the initial control experiment I chose population and generation to both be 100 so that there would be a decent amount diversity, and time to evolve. I also chose the mutation rate to be relatively low so as not to create too much diversity in the population. The results of the experiment can be seen in **table 2.**

Genetic Algorithm	**Control Parameters**				
	population	generations	mutation rate	**avg. cost**	**avg. time**
control experiment	100	100	0.015	**4147.00**	**0.177**
change in	10	-	-	**5723.33**	**0.019**
population	1000	-	-	**4035.00**	**1.803**
change in	-	10	-	**5945.67**	**0.012**
generations	-	1000	-	**4437.67**	**1.691**
change in	-	-	0.0015	**4329.33**	**0.177**
mutation rate	-	-	0.15	**5805.33**	**0.180**

Table 2: Genetic algorithm performance from change in constraints

Again we note that the results show values that produce minimum costs and times also produce maximum times and cost, respectively. However, unlike in simulated annealing, this is due to different parameters indicating that more control parameters impact performance for genetic algorithm. I ended up deciding to pick the initial control values for all the parameters since the only better performances cost wise or time wise weren't worth the trade-offs.

Ant Colony Algorithm (AC)

To create an ant colony algorithm I used the ANT-COLONY pseudocode in **section 3** along with a modified version of the C# code for ant colony optimization provided online by James McCaffrey [14]. By far ant colony was the most complex of the three natural metaphor algorithms. To simplify things I reused the population structure to represent a colony of ants. Also, most of the structures used were temporary lists for updating the pheromone trails which were themselves represented via a 2D array of doubles initialized to 0.01.

The ant colony algorithm had even more control parameters than genetic algorithm so I really had to narrow it down on what I wanted to test. I first chose the number of ants, as they are the ones that do the work. This algorithm takes longer than genetic algorithm so I used a smaller

population of ants as an initial control value. I also decided to test the pheromone increase and decrease factors, i.e. the amount the pheromones are updated. The results are shown in **table 3**.

Ant Colony	**Control Parameters**				
	number of ants	increase factor	decrease factor	**avg. cost**	**avg. time**
control experiment	40	2	0.01	**12676.0**	**2.396**
change in	4	-	-	**11562.3**	**0.395**
number of ants	400	-	-	**13019.3**	**24.36**
change in	-	0.2	-	**12594.0**	**2.556**
increase factor	-	20	-	**12433.0**	**1.953**
change in	-	-	0.001	**12280.3**	**3.321**
decrease factor	-	-	0.1	**12339.0**	**4.039**

Table 3: Ant colony algorithm performance from change in constraints

Unlike before, there is no inverse relationship. It is clear that picking a smaller number of ants for my implementation of the algorithm is best. This probably has to do with how the pheromones were updated, and is supported by the fact that having a large increase factor seems to also be better. Therefore I chose to use 4 ants, with an increase factor of 20, and decrease factor of 0.01.

Algorithm Performance on TSP

For 3NM since its control parameters were derived from the other algorithms, I decided to use the same values as them for parameters. I tested the different algorithms abilities to solve TSP by running all four of them on different problem sizes. The results are shown in **table 4**.

	avg. cost (length)				**avg. time (seconds)**			
problem size	**SA**	**GA**	**AC**	**3NM**	**SA**	**GA**	**AC**	**3NM**
5	**2603**	**2166**	**2603**	2594	0.008	**0.051**	**0.006**	0.012
10	**3150**	**3150**	**5977**	4374	**0.020**	**0.087**	0.071	0.086
15	**3513**	**3513**	8457	**8777**	**0.029**	0.128	0.091	**0.242**
20	**3998**	5560	**11533**	9786	**0.038**	0.170	0.183	**0.569**
25	**5099**	5336	**13622**	12574	**0.045**	0.220	0.474	**1.065**
50	**7707**	9294	26342	**32096**	**0.084**	0.558	5.418	**8.246**
100	**15771**	26008	**55673**	50446	0.169	**1.637**	22.86	**66.49**

Table 4: Comparison of algorithm performance on different problem sizes

5 Analysis of Results

Right away we can see some strong patterns in **table 4**. Mainly, simulated annealing seems to perform by far the best out of all the algorithms. Asides from with a problem size of 5, it obtained the smallest cost for each problem size. Additionally, it regularly took the least, or second least, average time. This is probably due to the fact that it injected the least amount of randomization into its algorithm, and the randomization it did inject was extremely controlled.

The fact that simulated annealing did well in minimizing cost is not that surprising. As alluded to in **section 2**, simulated annealing is often known to be able to find better solutions with it's main weakness being long computing time. Therefore, it is somewhat surprising that simulated annealing also performed the best time wise. This may be because the tested problem sizes of the experiment did not get to be big enough to strain it, after all we do see for the last problem size genetic algorithm performing slightly faster than simulated annealing. However this unexpected good performance in time may also be due to the city configurations which could have been more optimal for simulated annealing verses the other algorithms. It is also important to note that the algorithms ran for this experiment were adapted from basic versions, and therefore if we were to improve more upon the behaviors of the genetic and ant colony algorithms we may have seen different performances.

For example we see hints of ant colony algorithm's ability to provide fast results for small problem sizes with a problem size of 5 where it had the minimal cost. Additionally, it's not until I doubled the problem size from 25 to 50 that the average cost started exponentially increasing. Despite its potential to possibly have good running times, ant colony still had the worst performance cost wise. This is surprising since it seems that the ant colony algorithm I implemented had even worse performance than my own algorithm, 3NM. Since 3NM was largely based off of my ant colony algorithm this indicates that its poor performance is probably due to behaviors inherited from ant colony, but it luckily had some behaviors from simulated annealing and genetic algorithms that offset it.

The costs for all four algorithms are graphed in **figure 2**, and what is interesting about the performance of 3NM is how it seems to have a wave like behavior. In the beginning it performs on par with with simulated annealing and genetic algorithm, but then at a problem size of 15 its cost double! Again it maintained a steady increase in cost until again at a problem size of 50 it doubled. In fact, as the graph ends it looks like 3NM is leveling off again. These results indicate that we should look for perhaps some edge cases that create this extreme behavior in 3NM.

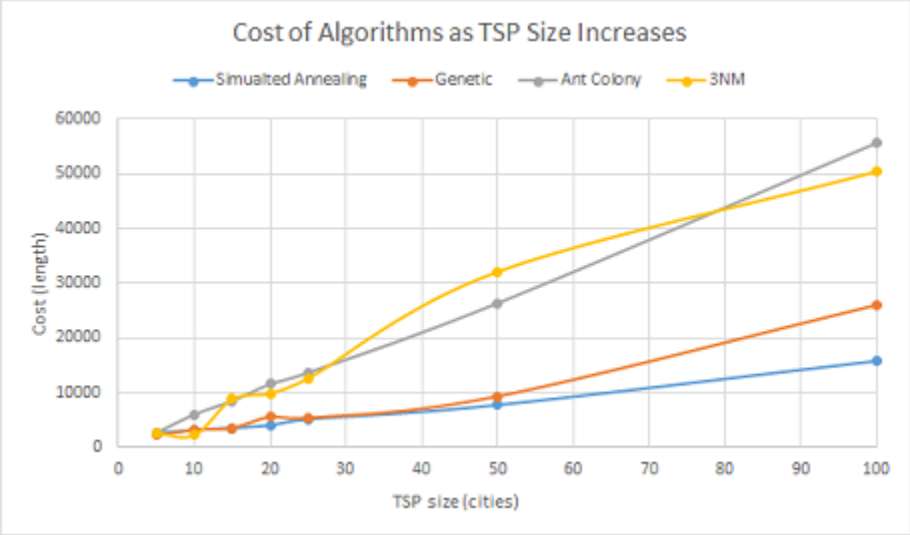

Figure 2: Graph of experiment costs

From **figure 2** we can see even more clearly the cost behaviors of the four algorithms. Mainly,

simulated annealing clearly outperformed the others with genetic algorithm doing well too. In fact, it was on par with simulated annealing for awhile before it started sloping upwards. It is worth noting that even though genetic algorithm had similar costs to simulated annealing at small problem sizes it had much higher times, and actually had the maximum times for the first couple problem sizes. This implies that it had to take a lot of effort to find the low costs, and as the problem size increased it wasn't able to keep up. This fits with the way the experiment was designed since genetic algorithm had a cut off generation level, and therefore had to stop after 100 generations regardless of the cost of the tour. If the generation level was specified to be higher it might have stuck on par with simulated annealing, of course though it would have taken a long time.

6 Conclusion

In this paper we compared the performance of different natural metaphor algorithms on TSP. Our goal was to look at the unique behavior of these sort of algorithms, and analyze their abilities. We did this by examining the impact of the different parameter controls in each algorithm, and found the results to be pretty straightforward. Additionally, the experiment really highlighted the natural metaphors used to create these algorithms. For example, it made a lot of sense that a large population size for genetic algorithm decreased the cost as a large population usually implies a diverse population. Additionally it made metaphorical sense that the same large population size led to a sharp increase in running time since in nature it is hard to see very concrete evolution in large population size, such as with us humans!

We used the information gathered from analyzing the three algorithms to create a new algorithm, 3NF, that takes the idea of ants creating pheromone trails from the ant colony algorithm along with the ideas of evolution across generations from genetic algorithm, and probabilistic accepting of solutions from simualted annealing. Finally, we performed our main experiment which was comparing the average cost and time performance of the four algorithms on TSPs of differing size. The results showed simulated annealing performing the best, with genetic algorithm doing similarly well for small problem sizes. It also showed that the ant colony algorithm performed extremely bad, and even worse than 3NF in cost.

In the future, looking more closing at the different control parameters and their impact on solving TSP can be useful. It was interesting that 3NF seemed to have a pattern of performing well in some problem sizes followed by poor performance in others. This indicates that combining the different behaviors from the three algorithms worked in some instances and didn't in others. Figuring out where those specific instances are, and why they are there could be a useful experiment.

References

[1] E. Aarts, J. Korst, and W. Michiels. Simulated annealing. In *Search methodologies*, pages 187–210. Springer, 2005.

[2] C. Chauhan, R. Gupta, and K. Pathak. Survey of methods of solving tsp along with its implementation using dynamic programming approach. *International Journal of Computer Applications*, 52(4), 2012.

[3] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, DTIC Document, 1976.

[4] V. Chvátal, W. Cook, G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large-scale traveling-salesman problem. In *50 Years of Integer Programming 1958-2008*, pages 7–28. Springer, 2010.

[5] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

[6] M. Dorigo and L. M. Gambardella. Ant colonies for the travelling salesman problem. *BioSystems*, 43(2):73–81, 1997.

[7] X. Geng, Z. Chen, W. Yang, D. Shi, and K. Zhao. Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search. *Applied Soft Computing*, 11(4):3680–3689, 2011.

[8] K. L. Hoffman, M. Padberg, and G. Rinaldi. Traveling salesman problem. In *Encyclopedia of Operations Research and Management Science*, pages 1573–1578. Springer, 2013.

[9] K. Hornik and B. Grün. Tsp-infrastructure for the traveling salesperson problem. *Journal of Statistical Software*, 23(2):1–21, 2007.

[10] L. Jacobson. Applying a genetic algorithm to the traveling salesman problem, 2012.

[11] L. Jacobson. Cs 312: Algorithm analysis, 2014.

[12] D. S. Johnson and L. A. McGeoch. The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*, 1:215–310, 1997.

[13] T. R. Martinez. Cs 312: Algorithm analysis, 2014.

[14] J. McCaffrey. Test run - ant colony optimization, 2012.

[15] C. Nilsson. Heuristics for the traveling salesman problem. Technical report, Tech. Report, Linköping University, Sweden, 2003.

[16] A. P. Punnen. The traveling salesman problem: Applications, formulations and variations. In *The traveling salesman problem and its variations*, pages 1–28. Springer, 2007.

[17] S. Russell, P. Norvig, and A. Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27, 1995.

[18] S.-h. Zhan, J. Lin, Z.-j. Zhang, and Y.-w. Zhong. List-based simulated annealing algorithm for traveling salesman problem. *Computational Intelligence and Neuroscience*, 2016, 2016.