Applying Ant Colony Optimization to the Traveling Salesman Problem

[**Introduction 2**](#_8xjqdbyef622)

[**The Traveling Salesman Problem 3**](#_cb7mmdtuoty)

[**Ant Colony Optimization 3**](#_mog8nfr0ttjb)

[Ant Movement 4](#_uvk447i7nqmd)

[Pheromone Calculations 6](#_wlscyan6e7v8)

[**Implementation and Data Collection 6**](#_o2lj5l69zfya)

[Tuning 7](#_n2efdeoocn47)

[Collection 8](#_o2g7jqjf0f55)

[Raw Data 8](#_rq9jvl80v24b)

[Algorithm iterations compared with shortest solution 8](#_7ofaqztcs3ih)

[ACO best solution compared with upper and lower bounds 9](#_gk1yuoy62n2r)

[**Analysis 10**](#_p6wdhaxs3u96)

[Modeling the Algorithm 10](#_g9uvv8kkho21)

[Using the Model 12](#_atwdqmb6u0dh)

[Derivation 12](#_w0bh3pu6emxd)

[Expanding the model 13](#_r2ycbb9q0cwx)

[Statistical Analysis 13](#_kqkl7kz2krgx)

[10 vertex graphs 14](#_a1ufr0xcy3zg)

[25 vertex graphs 15](#_k7q07ioewstt)

[40 vertex graphs 15](#_gapoto8u8h5m)

[**Conclusions 16**](#_dyhakmwygoo4)

[Limitations 17](#_n024euvbpvaz)

[**Works Cited 18**](#_p1zb9ebz0njd)

[**Appendix A: Data 19**](#_okd2x5saxwh1)

[Modeling The Algorithm 19](#_naua771zn5xk)

[Statistical Analysis 21](#_so4xjom934lc)

[10 Vertex Graphs 21](#_e1menin0vlra)

[25 Vertex Graphs 22](#_fymm643b7iom)

[40 Vertex Graphs 23](#_4dfav5fecmtz)

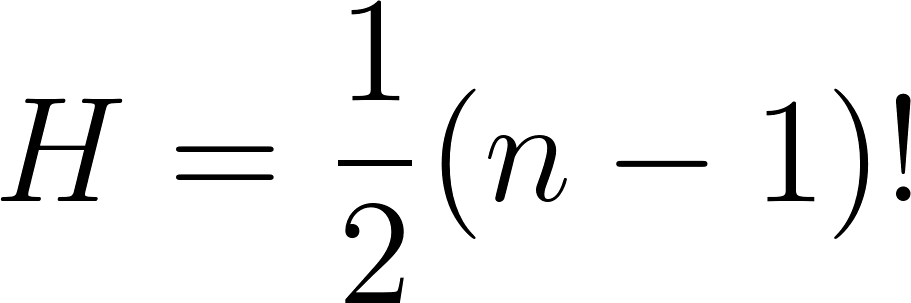
# Introduction

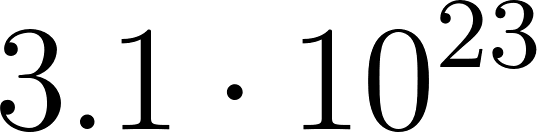
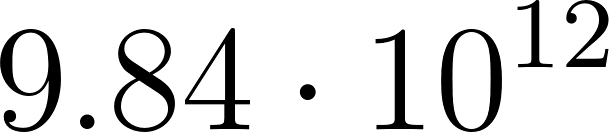
I have always been drawn to math because it is beautiful. There’s something so satisfying about describing our world with numbers and then applying math to those numbers to gain a deeper understanding about how things work. One night, when I was browsing YouTube, I came upon a video titled “Coding Adventure: Ant and Slime Simulations'' [[1]](#footnote-0). And what I found as I watched was a perfect example of how math can solve problems in beautiful and fascinating ways. In one part of the video, the creator explored the traveling salesman problem (TSP), and the part that interested me the most was the way that they solved it. The creator used a technique called ant colony optimization (ACO), which involves simulating a large number of ants walking between points on a graph, and measuring the lengths of the paths they take. When multiple generations of ants are simulated, they are able to find optimal solutions to the TSP. This fascinated me at the time, as I could not see the connection between ants and graph theory.

Thus, the aim of this paper is to explore the ACO algorithm, and compare its effectiveness and efficiency to traditional methods. This will use concepts from graph theory and algorithms within as well as calculus. First, I will introduce the steps of the algorithm. Then, I will gather data with a custom implementation of the algorithm written in Python. Finally, I will use calculus to estimate the point at which diminishing returns impact the algorithm’s performance, and make conclusions about ACO’s usefulness in everyday life.

# The Traveling Salesman Problem

The traveling salesman problem aims to find the cycle that passes through all of the vertices on the graph once (also called a Hamiltonian cycle) of least weight for a given complete graph. And herein lies the challenge, the number of Hamiltonian cycles on a given graph increases with the factorial of the number of vertices on the graph. Specifically, the number of cycles for a graph is defined by the following equation

[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%20H%3D%5Cfrac%7B1%7D%7B2%7D(n-1)!%20#0)

Where [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=n#0) is the number of vertices on the graph and [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=H#0) is the number of possible Hamiltonian cycles in said graph. This makes a brute force approach infeasible for graphs with larger vertex counts. For example, a fully connected graph with 5 vertices would have only 12 cycles for an algorithm to check. While a fully connected graph with 25 vertices would have [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=3.1%20%5Ccdot%2010%5E%7B23%7D#0) cycles for an algorithm to check. Even if a brute force algorithm could check one cycle every millisecond, it would take [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=9.84%20%5Ccdot%2010%5E%7B12%7D#0) years to check them all ().

While no real world application of this problem would use a brute force algorithm, other exhaustive algorithms are subject to the same issue, and running these computationally intensive algorithms has both environmental and financial ramifications.

# Ant Colony Optimization

As discussed above, I am going to be applying an algorithm called ant colony optimization (ACO) to attempt to solve this problem. At the highest level, this algorithm revolves around placing virtual agents at different vertices on the graph, and simulating their movement between the other vertices on the graph. These agents are called ants, as an analogy to the behavior of ants in the real world. This algorithm is also genetic, which means that after one group of ants have completed their walks around the graph, a new group is run through the graph, with their decisions of where and when to move between certain vertices influenced by the previous group. This is accomplished by associating a pheromone value with each edge on the graph that describes how many agents have traversed that edge in the past.

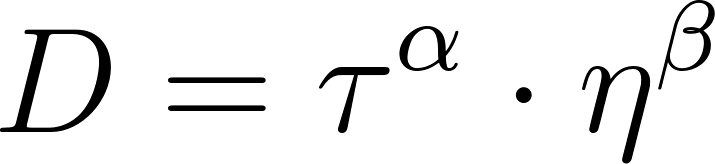
Here’s a breakdown of the steps involved:

1. First, we place a set number of ants at random starting vertices on the graph.
2. Next, we let each ant wander between vertices using a set of rules that describe how the ant decides which vertex to move to.
3. After all ants have completed a Hamiltonian cycle, we compute the total weight of the path that each ant took.
4. Finally, we use those weights to lay a “pheromone” trail on the graph for future generations of ants to follow.
5. Finally, we repeat the process from step 1 with a new generation of ants.

## Ant Movement

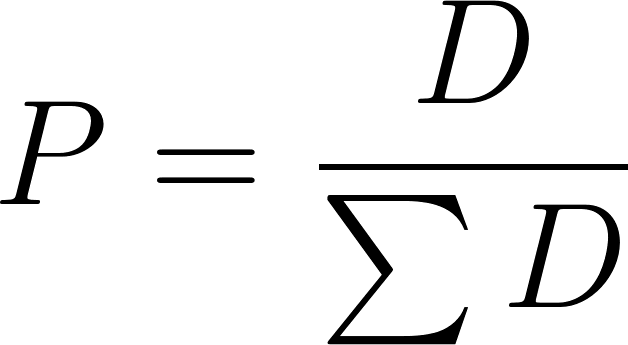
An ant’s decision about what vertex to move to next is influenced by two factors: the weight between its current position and a possible vertex, and the concentration of pheromone on the edge connecting the two vertices. This value is called a desirability, and each ant in the algorithm will calculate a desirability for each vertex it could possibly travel to. The ant will make its final decision at random, but this allows us to have a set of tuning constants that define the relative importance of distance versus pheromone concentration when an ant is making a decision.

A desirability for a possible vertex, given a current vertex is calculated with the following formula:

[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=D%3D%5Ctau%5E%7B%5Calpha%7D%20%5Ccdot%20%5Ceta%5E%7B%5Cbeta%7D#0)

Where [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%5Ctau#0) is the concentration of pheromone on the edge connecting the vertices, and [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%5Ceta#0) is the inverse of the distance between the vertices. I used the inverse of the weight because we want closer, and therefore shorter, paths to be more desirable. [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%5Calpha#0) and [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%5Cbeta#0) are the tuning constants discussed above. Increasing [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%5Calpha#0) will make the pheromone concentration more influential in the ants’ decision making process. Increasing [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%5Cbeta#0) will make the distance between vertices more influential in the process.

Once an ant has calculated the desirability for every vertex it could possibly move to, it divides the desirability for each vertex by the sum of all desirabilities to obtain a probability of moving to that vertex. Thus, the probability of an ant moving to a given vertex can be defined with:

[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=P%3D%5Cfrac%7BD%7D%7B%5Csum%20D%7D#0)

Finally, the ant simply chooses a vertex to move to with the probabilities found and moves there. This is repeated until all ants in the algorithm have completed a Hamiltonian cycle.

## Pheromone Calculations

As well as having a weight associated with it, each edge in a graph will also have a pheromone concentration associated with it. When the algorithm starts, these values are all initialized to 0.

After all ants have completed their cycles, and before the next generation of ants is created, these concentrations are updated.

First, a deposit concentration is calculated for each ant, which is simply the inverse of the total length of the cycle that the ant completed. As with above, I used the inverse here because I wanted to reward paths of shorter length.

Then, for each ant, its deposit concentration is deposited on all of the edges that it traversed in its cycle.

There is one more important step here, we have to “evaporate” the pheromones. Over multiple generations of ants, pheromone build up can be very detrimental to the performance of the algorithm, so it is necessary to evaporate the pheromones on the graph periodically. This is a simple process: for each edge on the graph, simply multiply its current pheromone concentration by an evaporation constant, which will be represented by [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%5Crho#0).

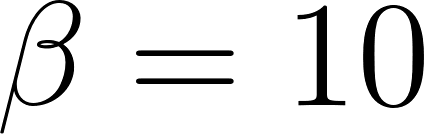
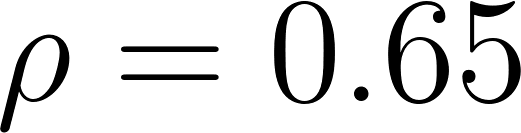
# Implementation and Data Collection

To start, I implemented the ACO algorithm in the Python programming language according to the rules discussed above. The full implementation can be found on my github page[[2]](#footnote-1). In addition to the ACO algorithm, I also implemented the nearest neighbor algorithm to represent the traditional approach for computing solutions to the TSP. I also chose to implement the deleted vertex algorithm to provide another comparison point. These two traditional algorithms give the upper and lower bounds for possible solutions to the traveling salesman problem for that particular graph, making them useful comparisons for the length of solutions found by ACO. Although, it should be mentioned that the comparison will be between the nearest neighbor algorithm and ACO because the nearest neighbor algorithm finds a cycle while deleted vertex finds a weight but not a cycle.

I also used MatPlotLib[[3]](#footnote-2) to generate graphs for analysis and visual representation of the algorithms.

## Tuning

The next step was to tune the algorithm. As discussed above, the ACO requires [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%5Calpha#0), [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%5Cbeta#0), as well as a constant to represent how fast pheromones evaporate, but there are a few more constants that need to be accounted for: [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=n#0), the number of generations of ants that are simulated over the course of running the algorithm, and [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=a#0), the number of ants simulated per generation. After a little bit of testing, I settled on the following constants to collect data with:

[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%5Calpha%20%3D%201%20#0)  
[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%5Cbeta%20%3D%2010%20#0)  
[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%5Crho%20%3D%200.65%20#0)  
[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=n%20%3D%2050#0)

The constant for the number of ants is a little bit different, however, as it is dependent on the number of nodes in the graph. From my testing, it seemed like a relation of 1 ant for every 3 nodes on the graph was sufficient.

## Collection

Because ACO becomes more useful as the size of a graph increases (remember back to how the number of cycles increases with the factorial of vertices), I decided that to fully test the algorithm, I needed to run tests at different graph sizes. I decided on 10, 25, and 40 vertex graphs. For each graph size, I ran 25 tests with randomly generated graphs, and recorded the result from the nearest neighbor algorithm as well as the best solution found by the ACO algorithm. The full data tables can be found in Appendix A: Data.

### Raw Data

#### Algorithm iterations compared with shortest solution

| Number of Iterations | Shortest Solution |
| --- | --- |
| 1 | 662 |
| 2 | 629 |
| 3 | 628 |
| 4 | 628 |
| 5 | 616 |
| 6 | 616 |

This dataset was used to model the algorithm.

#### ACO best solution compared with upper and lower bounds

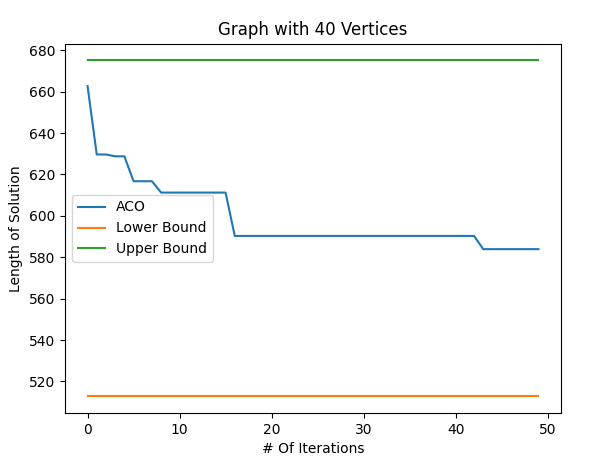
| Nearest Neighbor | ACO |
| --- | --- |
| 286 | 280 |
| 449 | 374 |
| 326 | 280 |
| 272 | 272 |
| 336 | 284 |
| 343 | 310 |
| 348 | 298 |

This data was collected for graphs with 10 vertices, and was used for comparing the performance of the two algorithms for finding solutions.

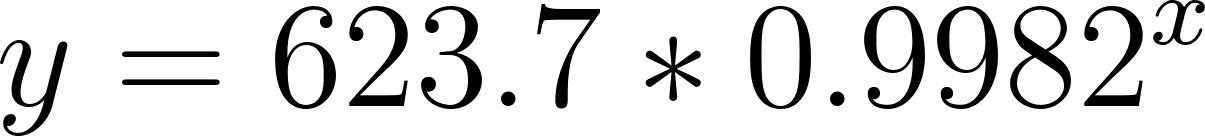
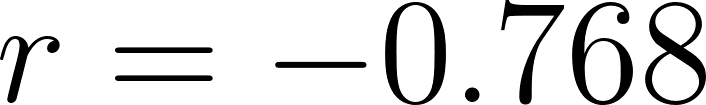
# Analysis

## Modeling the Algorithm

The first thing I had to do was choose a run to analyze, and I ended up going with the following run with 40 vertices:



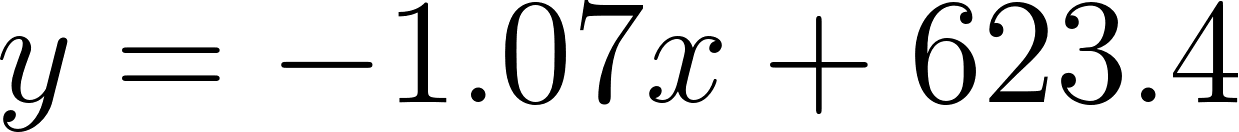
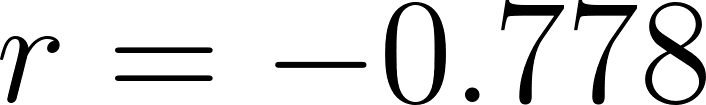
Just by looking at the graph, my intuition was that this data would be exponential, but I didn’t know for sure, so I ran some regressions. The first one I tried was an exponential regression, which had the following results

[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%20y%3D623.7*0.9982%5Ex#0)  
[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=r%3D-0.768%20#0)

I found this very interesting, as it was nowhere near as good of a fit as I would have expected. I was especially interested in the fact that the base was very, very close to 1. This prompted me to run some other types of regressions to see if any fit better. First was a natural log regression

  
[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=r%3D-0.84%20#0)

Already this was a much better fit. Just to make sure, though, I ran a linear regression as well. Which had the following results

[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%20y%3D-1.07x%2B623.4#0)  
[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=r%3D-0.778%20#0)

This was not super surprising after seeing the results from the natural log regression, but I just wanted to make sure that I wasn’t missing any obvious patterns in the data.

From this analysis I could say that the progression of this algorithm was strongly correlated to that of the following natural log function



Looking back at the original graph, and at the design of the algorithm itself, a logarithmic model actually makes a lot of sense. As the algorithm progresses, and more generations of ants run through the graph, the best pathways will become more and more defined in contrast to the disadvantageous paths, which means that more and more ants will settle into the same cycle. So, as the algorithm runs, it becomes less and less likely that a shorter solution will be found.

## Using the Model

The ACO algorithm is not designed to solve the traveling salesman problem. Instead, it aims to find an optimal solution, a solution that is of acceptable length. This, combined with its genetic nature, means that it could be run forever. Given enough time, it is possible that ACO could find a solution shorter than the one it currently has. However, at a certain point, the resources involved with continuing computation outweigh the benefit of a possible shorter solution.

In other words, the law of diminishing returns applies in this situation, where there is a discrete point in the algorithm at which it makes logical sense to stop computation and accept the current shortest cycle as optimal.

To estimate this stopping point, we can use the model we developed in the previous section, along with some calculus.

### Derivation

I took the derivative of the model as follows:

Logically, the point of diminishing returns for this algorithm will be once the slope of the progression becomes greater than negative one, as after that it will just get closer and closer to zero forever. With this in mind, it makes sense that the point of diminishing returns for this run would be at 18.9, or 19 runs rounded to the nearest whole run. This means that, for this size of graph, with these algorithm parameters, it would only make sense to run it for 19 iterations before declaring the current shortest solution as the optimal solution.

### Expanding the model

This model will of course only work best with the specific graph, and parameters that were used to calculate the regression. For a better understanding of the amount of time an ACO solver should run for a graph with 40 vertices, averages can be taken over many different test graphs. Running 10 regressions on different algorithm evolutions with 40 vertices, I collected the following data:

| 11 | 13 | 13 | 4 | 22 | 18 | 19 | 13 | 12 | 18 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

With this, an average of , or 14 iterations rounded to the nearest whole iteration.

This means that, for graphs with 40 vertices, ACO will find a solution that we can deem optimal in around 14 iterations of the model.

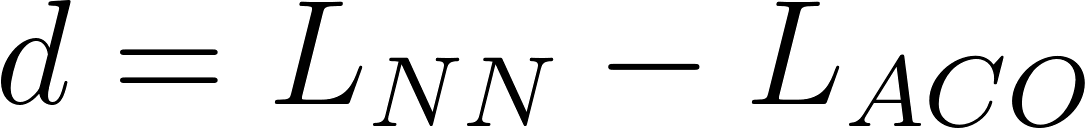
## Statistical Analysis

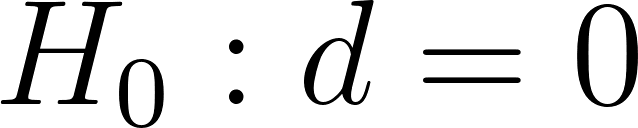
Next, I wanted to prove that my algorithm was better able to find solutions to the TSP than traditional methods. While it is somewhat obvious from the raw data alone that ACO performs better than nearest neighbor, I wanted to confirm that there was indeed a significant difference in the results. To do this, I conducted a series of statistical tests.

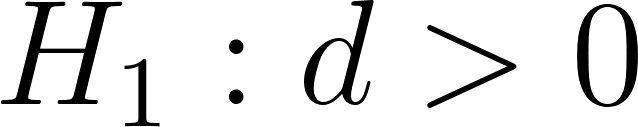
### 10 vertex graphs

Before performing any tests, I wanted to look at some basic statistics from the data. The statistic I was most interested in was the means of the datasets. I was not as concerned with other statistics like standard deviation as much because those depend on the graphs the algorithms are being run on, which in this case were randomly generated every time. For the 10 vertex data, I calculated the following statistics:

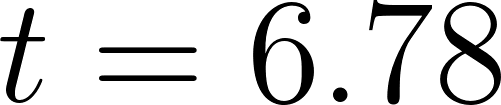
The mean length solution found by the nearest neighbor algorithm was 329.16 units, while the mean length solution found by the ACO algorithm was 297.08 units. After seeing this, I was very confident in my algorithm’s ability to outperform traditional methods for small vertex count graphs, but to be sure, I ran a paired T-test.

[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%20d%3DL_%7BNN%7D-L_%7BACO%7D#0)

[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=H_0%3A%20d%20%3D%200#0)

[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=H_1%3A%20d%20%3E%200#0)

Where d is defined as the difference between the data points of the nearest neighbor and ACO algorithms. I decided on a significance level of 1% as I was already very confident in my algorithm’s ability to outperform the nearest neighbor algorithm. The results of test were as follows:

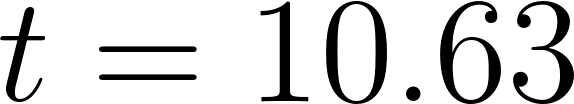
[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%20t%3D6.78#0)

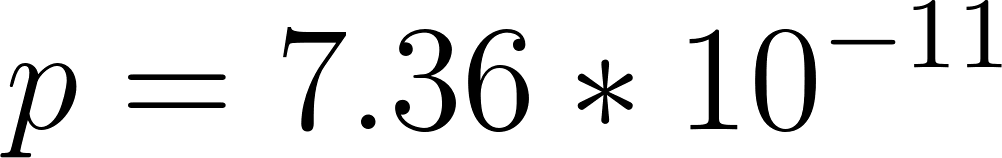
[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=p%3D2.61*10%5E%7B-7%7D%20#0)

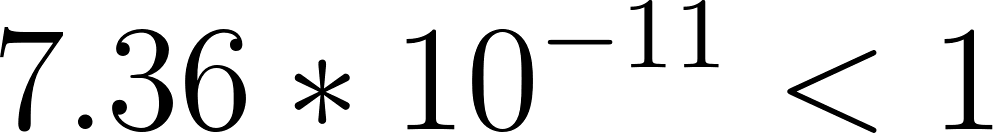
This [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=p#0) value was smaller than the set significance level for the test, which means that I was able to reject the null hypothesis and say with confidence that: On graphs with small vertex counts, my algorithm was able to outperform traditional methods for computing solutions to the TSP.

### 25 vertex graphs

With the results from the analysis on the smaller graphs, I was expecting to see similar results for the 25 vertex graphs, running the same paired T-test as before produced the following results

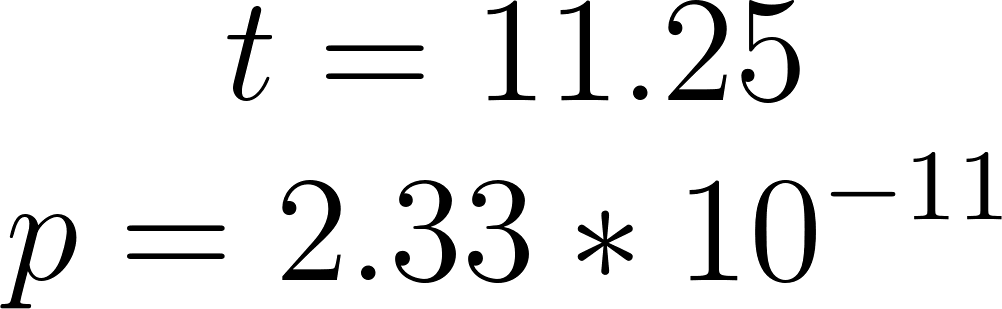
[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%20t%3D10.63#0)

[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=p%3D7.36*10%5E%7B-11%7D%20#0)

[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=7.36*10%5E%7B-11%7D%20%3C%201#0) Therefore I was able to reject the null for this test.

### 40 vertex graphs

At this point I was very confident about what the results would be on the largest set of graphs that were tested, running the same test as previous yielded the following results

[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%20t%3D11.25%5C%5C%20p%3D2.33*10%5E%7B-11%7D%20#0)

For the third and final time, this [](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=p#0) value was lower than the 1% significance level set by the test, and therefore I could now say with confidence that my algorithm was able to outperform traditional methods on graphs with large vertex counts.

# Conclusions

The traveling salesman problem can be applied in a myriad of different ways in the real world, which means that there is a lot of potential for improvement for real world systems. A good example is in “last mile delivery.” This is the step of the delivery process where a package goes from a warehouse to a customer’s desired delivery location. Last mile delivery is the leading cost driver in the supply chain[[4]](#footnote-3). Let’s look at it this way: Say you are a delivery driver. Your job every day is to deliver 50 packages from a warehouse to their final locations and then come back to the warehouse when you are done. This is a classic application of the traveling salesman problem, where each of the destinations you have to deliver to can be represented by a vertex, and the edges connecting those vertices have weights that can represent things like the amount of time it takes to move between the vertices or the amount of fuel it takes to go from one to the next. Because your delivery business does not have unlimited money for fuel, or unlimited time to deliver packages, you have to find a Hamiltonian cycle with a sufficiently small weight to make your business profitable. This is the exact kind of real world problem that genetic algorithms like ACO are good at solving.

The ability to optimize routes like these in the real world has many benefits. In the case of the delivery driver problem, reductions in route length can lead to lower costs for shipping due to less fuel used. These optimizations reduce environmental impact for these processes as well.

## Limitations

While my exploration did yield results to the questions that I asked earlier, it did have some important limitations. The first lies in the very nature in the comparison that I made. While the nearest neighbor algorithm is a valid method for finding a solution to the traveling salesman problem, it is a very inefficient way to do so. In any real world application of this problem, a more efficient algorithm would be used. Therefore, in the context of real world solutions, this comparison may not be applicable in a meaningful sense. However, in the scope of my own learning and personal interests, this comparison is much more apt, as it allowed me to make comparisons against an algorithm that I was already familiar with through my math class.

# Works Cited

Bouchard, W. ACOpy [Computer software]. <https://github.com/Westly-Bouchard/mathIA>

Dorigo, Marco. “The Ant System: Optimization by a colony of cooperating agents.” Unibo.it. Accessed March 3, 2022. <http://www.cs.unibo.it/babaoglu/courses/cas05-06/tutorials/Ant_Colony_Optimization.pdf>

Hunter, J. D., “Matplotlib: A 2D graphics environment,” Computing in Science \& Engineering, vol. 9, no. 3, pp. 90-95, 2007, doi: 10.1109/MCSE.2007.55.

Ma, Suzanne. “Solving the Travelling Salesman Problem for Deliveries.” Routific, January 2, 2020. <https://blog.routific.com/travelling-salesman-problem>.

Sebastian, Lague. “Coding Adventure: Ant and Slime Situations.” *Youtube* video, 17:53. March 25, 2021. <https://www.youtube.com/watch?v=X-iSQQgOd1A>

# Appendix A: Data

## Modeling The Algorithm

| Number of Iterations | Shortest Solution |
| --- | --- |
| 1 | 662 |
| 2 | 629 |
| 3 | 628 |
| 4 | 628 |
| 5 | 616 |
| 6 | 616 |
| 7 | 616 |
| 8 | 611 |
| 9 | 611 |
| 10 | 611 |
| 11 | 611 |
| 12 | 611 |
| 13 | 611 |
| 14 | 611 |
| 15 | 611 |
| 16 | 590 |
| 17 | 590 |
| 18 | 590 |
| 19 | 590 |
| 20 | 590 |
| 21 | 590 |
| 22 | 590 |
| 23 | 590 |
| 24 | 590 |
| 25 | 590 |
| 26 | 590 |
| 27 | 590 |
| 28 | 590 |
| 29 | 590 |
| 30 | 590 |
| 31 | 590 |
| 32 | 590 |
| 34 | 590 |
| 35 | 590 |
| 36 | 590 |
| 37 | 590 |
| 38 | 590 |
| 38 | 590 |
| 39 | 590 |
| 40 | 590 |
| 41 | 590 |
| 42 | 590 |
| 43 | 583 |
| 44 | 583 |
| 45 | 583 |
| 46 | 583 |
| 47 | 583 |
| 48 | 583 |
| 49 | 583 |
| 50 | 512 |

## 

## Statistical Analysis

### 10 Vertex Graphs

| Nearest Neighbor | ACO |
| --- | --- |
| 286 | 280 |
| 449 | 374 |
| 326 | 280 |
| 272 | 272 |
| 336 | 284 |
| 343 | 310 |
| 348 | 298 |
| 304 | 289 |
| 315 | 275 |
| 336 | 336 |
| 300 | 281 |
| 301 | 286 |
| 280 | 269 |
| 322 | 285 |
| 319 | 275 |
| 359 | 322 |
| 234 | 234 |
| 252 | 259 |
| 349 | 320 |
| 391 | 362 |
| 400 | 335 |
| 354 | 304 |
| 331 | 300 |
| 355 | 267 |

### 25 Vertex Graphs

| Nearest Neighbor | ACO |
| --- | --- |
| 448 | 415 |
| 424 | 329 |
| 493 | 485 |
| 561 | 428 |
| 473 | 420 |
| 545 | 404 |
| 538 | 432 |
| 504 | 451 |
| 511 | 439 |
| 487 | 436 |
| 443 | 429 |
| 598 | 485 |
| 520 | 416 |
| 492 | 449 |
| 477 | 424 |
| 505 | 404 |
| 491 | 406 |
| 501 | 424 |
| 578 | 444 |
| 489 | 426 |
| 564 | 461 |
| 467 | 440 |
| 593 | 482 |
| 503 | 447 |

### 40 Vertex Graphs

| Nearest Neighbor | ACO |
| --- | --- |
| 633 | 540 |
| 576 | 520 |
| 615 | 575 |
| 668 | 556 |
| 611 | 487 |
| 624 | 521 |
| 599 | 527 |
| 626 | 540 |
| 687 | 555 |
| 743 | 608 |
| 650 | 550 |
| 648 | 557 |
| 639 | 553 |
| 622 | 516 |
| 772 | 578 |
| 652 | 586 |
| 691 | 545 |
| 630 | 547 |
| 723 | 55 |
| 719 | 590 |
| 542 | 531 |
| 711 | 566 |
| 675 | 583 |
| 610 | 591 |

1. Sebastian Lague. “Coding Adventure: Ant and Slime Situations.” *Youtube* video, 17:53. March 25, 2021. <https://www.youtube.com/watch?v=X-iSQQgOd1A> [↑](#footnote-ref-0)
2. Bouchard, W. ACOpy [Computer software]. <https://github.com/Westly-Bouchard/mathIA> [↑](#footnote-ref-1)
3. Hunter, J. D., “Matplotlib: A 2D graphics environment,” Computing in Science \& Engineering, vol. 9, no. 3, pp. 90-95, 2007, doi: 10.1109/MCSE.2007.55. [↑](#footnote-ref-2)
4. Suzanne Ma, “Solving the Travelling Salesman Problem for Deliveries,” Solving the Travelling Salesman Problem for deliveries (Routific, January 2, 2020), https://blog.routific.com/travelling-salesman-problem. [↑](#footnote-ref-3)