Comparison of AI Heuristics in Traveling Salesman Optimization

Austin Hamilton

Contents

1 Introduction

1.1 Traveling Salesman Optimization

The Traveling Salesman Problem (TSP) is an example of a routing problem, i.e., a problem that involves finding a path that contains every node within a graph. Such a path is known as a Hamiltonian path. TSP is concerned with deciding if a cycle exists within a graph in which the weights of the edges of the cycle are less than some constant λ. A variation of TSP is TSP-OPT (Traveling Salesman Optimization). A solution to TSP-OPT is the minimum (or maximum) Hamiltonian cycle within a graph.

1.2 AI Heuristics

A heuristic is a piece of metainformation that is used to make decisions by an algorithm. Depending on the nature of the problem, heuristics leads to quicker solutions or better approximations. Artificial intelligence (AI) relies heavily on heuristics when searching for solutions. When a problem space is too large to efficiently search, heuristics are used to reduce the space an AI agent must explore to find a solution to a problem.

2 Motivation

2.1 Practical Value

Routing problems like the traveling salesman problem are considered intractable *when an optimal solution is sought*. However, heuristics can be applied to these problems to quickly find suboptimal, but acceptable solutions. The importance of being able to solve these problems efficiently becomes apparent when considering the real-world implications. Routing algorithms can be applied to various practical applications, ranging from wire planning on a circuit board to planning flight patterns. Corporations use routing algorithms to produce goods and services with a significant impact on the global economy and critical infrastructure.

2.2 Theoretical Value

TSP-OPT is classified as NP-hard. An NP-hard problem is a problem that is more complex than any other problem in NP and as such can be reduced into any NP problem in polynomial time. NP is a

class of complex problems that can be verified in polynomial time, but not necessarily solved in polynomial time. This means that a strategy for TSP-OPT can be applied to all NP problem. Heuristics for TSP-OPT can then be applied to a wide range of similar problems in computer science, such as graph isomorphisms, the clique problem, and the knapsack problem.

3 Research Problem and Approach

In this study, I compare three AI heuristics for solving TSP-OPT, Simulated Annealing (SA), Genetic Algorithms (GA), and Ant Colony Optimization (ACO). All three heuristics approximate solutions for a graph, but I could not find literature comparing these three specific heuristics for TSP-OPT. I was interested in finding which of the three heuristics is the most efficient, i.e., which one was the quickest and which one was the most accurate. I define a heuristic, g, to be quicker than a heuristic, h, if g approximates a solution within 5% of the optimal solution in less time than h does. On the other hand, g is more accurate than h if it produces a better approximation than h in a set amount of time.

I was also interested in the effects of a graph's properties on the efficiency of the heuristic. To reduce confounding variables, I chose to compare a heuristics efficiency to a graph's closeness. A close graph would be one whose nodes are generally near all its neighbors: it is locally dense. I use the average distance between two nodes as the closeness of a graph. This measurement would necessarily affect the accuracy of a heuristic (larger average distances would result in a longer overall tour), so I only measure the correlation between closeness and quickness.

To compare the efficiency of the heuristics, I devised two independent tests, a performance test and a time test. The performance test measures accuracy, and the time test measures quickness.

4 Literature Review

4.1 Simulated Annealing

In a 2016 paper, Zhan et al. proposed a method for using SA for TSP-OPT. The authors took strategies from other SA implementations and combined them. Zhan et al.'s algorithm uses path

representation to describe a tour. This means that *a* would represent the path that goes from node 1, to node 2, to node 3, …, to node 5.

a.) $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$

The authors implemented a hybrid successor choice that involves combining three common successor choosing strategies: the inverse operator, the insert operator, and the swap operator. The inverse operator randomly chooses two positions in the path and reverses the order of the nodes within the two positions. The insert operator selects two random nodes from the path. It then takes the first node, removes it from the list, then inserts it before the second node. The swap operator chooses two random nodes from the path and swaps them in the path. In each iteration of the authors' implementation, the algorithm uses all three strategies and chooses the path that has the shortest distance.

Once a successor has been chosen, if the successor is more fit (shorter) than the current path, it is already chosen. Otherwise, the successor is chosen with a probability of $e^{-\frac{d_i}{t_i}}$, where d_i is the difference between the successor and the current path, and t_i is the current temperature. The temperature is a value that correlates to how likely a less fit value is chosen. The temperature is initially a large value–which correlates to a higher probability–and is "cooled" throughout the algorithm, based on d_i and the random number, r_i, that was generated to check against the probability of choosing a successor.

$$t_i = -\frac{d_i}{\ln(r_i)}$$

Zhan et al. report convergence on the optimal solution for a 380-node graph within 500 iterations. Zhan et al. also show that their temperature schedule and geometric reduction reached a similar temperature at 500 iterations. The authors ran 25 tests with their implementation on different graphs. They found that their algorithm was able to achieve 5% of the optimal solution within 3 seconds on every test but one.

4.2 Genetic Algorithms

In 2017, Hussain et al. provided an implementation of a genetic algorithm for TSP-OPT in their paper, *Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator*. The

authors note that normal crossover methods do not work for TSP-OPT. Within TSP-OPT, every node must be visited exactly once. If a generic crossover method is used, it is likely that offspring produced by the crossover would have duplicate nodes in them. Several solutions have been proposed to get around this. In Hussain et al.'s paper, they compare three popular crossover methods, partially mapped crossover (PMX), ordered crossover (OX), and cycle crossover (CX).

4.2.1 Partially Mapped Crossover

PMX is achieved through a partial mapping of the genomes. The genome is represented in path representation, where each gene is a node in a path throughout the graph. Two crossover points are chosen at random, and a bidirectional mapping is made between respective genes. When two parents, P_1 and P_2 are selected, the first offspring, O_1 receives the genes from P_2's crossover section. Remaining values are filled from P_1 in their respective spots, given they don't reside in the crossover section of P_2.

$$P_1 = 3 \rightarrow 4 \rightarrow 8 \rightarrow |2 \rightarrow 7 \rightarrow 1| \rightarrow 6 \rightarrow 5$$

$$P_2 = 4 \rightarrow 2 \rightarrow 5 \rightarrow |1 \rightarrow 6 \rightarrow 8| \rightarrow 3 \rightarrow 7$$

$$O_1 = 3 \rightarrow 4 \rightarrow X \rightarrow |1 \rightarrow 6 \rightarrow 8| \rightarrow X \rightarrow 5$$

After this step, a partial mapping is used to fill in the remaining spots. The first X above would be filled in with a 2, and the second would be filled in with a 7. This is decided by taking the gene in P_1 directly opposite of the gene in the crossover section of P_2. If this value has also been taken, continue in this manner until you find a gene that works.

4.2.2 Ordered Crossover

In OX, the crossover takes a portion from one parent, and then fills in the remaining values with genes from the other parent, preserving relative path position. First, two random crossover points are selected. Then the genes from the second parent's (P_2's) crossover section is copied to the offspring, O_1. Then starting at the first gene of the first parent, P_2, each gene is copied over to O_1. If a gene already exists in O_1, then that gene is skipped and the next gene is copied to the current position instead.

$$P_1 = 1 \rightarrow 2 \rightarrow |3 \rightarrow 4 \rightarrow 5| \rightarrow 6$$

$$P_2 = 2 \rightarrow 4 \rightarrow |6 \rightarrow 1 \rightarrow 3| \rightarrow 5$$

$$O_1 = X \rightarrow X \rightarrow 6 \rightarrow 1 \rightarrow 3 \rightarrow X$$

$$O_1 = 2 \rightarrow 4 \rightarrow 6 \rightarrow 1 \rightarrow 3 \rightarrow 5$$

4.2.3 Cycle Crossover

In CX, the goal is to find a cycle through the genes, taking a gene from each chromosome at a time. Starting from the first gene from the first parent P_1, the gene is taken. Then the gene that is directly across from that gene–similar to PMX–is placed in the same position it is placed in P_1. This process is continued until the crossover ends in a cycle: the mapped gene is the first gene that was added.

$$P_1 = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$$

$$P_2 = 8 \rightarrow 5 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 4 \rightarrow 7$$

$$O_1 = 1 \rightarrow X \rightarrow X \rightarrow 4 \rightarrow X \rightarrow X \rightarrow 7 \rightarrow 8$$

After the genes from P_1 are taken, the remaining genes are taken in order from the second parent, P_2.

$$O_1 = 1 \rightarrow 5 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 8$$

4.2.4 Results

Hussain et al. found that the performance of the genetic algorithms depended on the type of graph being used, e.g., directed or undirected. However, in general cases, all three crossover methods performed similarly on average. PMX slightly outperformed the other two on average, with CX having the best of the best cases and OX having the best of the worst cases.

4.3 Ant Colony Optimization

Dorigo et al. proposed ACO for TSP-OPT in *Ant Colony Optimization*, published in 2004. In the book, they present a new optimization technique, ACO, and show several applications. ACO works by simulating how the collective behavior of ants can find optimal routes to food sources. Individual ants will make random walks until they find food. Once they find a food source, they follow a pheromone trail

back to the colony. These pheromone trails are deposited by each ant and evaporate over time. Consequently, the more ants follows a path, the more likely another ant is to follow that path. The converse is also true; the less ants follow a path, the less likely another ant is to follow the path. This collectivism leads to emergent behavior that optimizes paths as more ants are added to the environment.

Dorigo et al. propose a simulation of an ant colony called Ant System (AS). AS works by generating a "colony" of "ants" to make successive tours over a graph. After each ant has made a tour, it deposits pheromone to the graph. After deposition, evaporation takes place. Dorigo et al. suggest a pheromone deposit of $\frac{1}{C_k}$ where C_k is the total length of ant k's tour. This pheromone is added to the pheromone of each edge in the ant's tour. The authors also suggested a geometric reduction for the evaporation of pheromone. Every edge in the graph is reduced by a factor e at the end of an iteration. Dorigo et al. claim that any sufficiently small number can be used as an initial value for each edge, as long as the value is uniformly distributed among the edges.

Each ant has a probability of choosing an edge to advance in each step. The ants continue in this manner until all the nodes have been visited. Dorigo et al. derived a formula to determine the probability an ant chooses to advance to a node.

$$p_{ij}^{k} = \frac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{l \in N_i^k}[\tau_{il}]^{\alpha}[\eta_{il}]^{\beta}}, \text{ if } j \in N_i^k$$

τ and η are the pheromone and metaheuristic of an edge, $i \rightarrow j$. The metaheuristic of an edge is inversely proportional to the length of the edge, i.e., $\frac{1}{length}$. α and β are two constants used to exaggerate or dampen the effects of the pheromone or metaheuristic.

Dorigo et al. ran several tests on various implementations of ACO. They tested the implementations on a computer with a 700 MHz Pentium 3 processor and 512 GB of RAM. The implementations were tested on a dataset of graphs with optimal solutions. There were slight deviations in their implementations, but their performances quickly converged. For example, on a graph with 198

nodes, all of Dorigo et. al.'s implementations were within 12% of the optimum within 1 second. On a graph with 783 nodes, their implementations were within 17% of the optimum after 20 seconds.

5 Solution

5.1 Time Test

The time test needs a comparison to the optimal solution, so the test involves running a brute force approach to get the target cycle distance. As such, the time test must be run on a smaller graph. The test is run on the same graph for a brute force algorithm and each heuristic. The optimal solution distance is stored from the brute force and kept for the remainder of the test. Each heuristic is run until its solution distance is within 5% of the optimal.

After each solution is run, the solution's runtime is recorded. The graph's closeness is recorded to measure the effect of closeness on a heuristic and act as an identifier for a single graph.

5.2 Performance Test

The performance test compares the performance of the three heuristics on different sized graphs. Three graphs are generated with 50 nodes, 100 nodes, and 500 nodes, respectively. Each graph has an associated time with it–50 node graphs get 30 seconds, 100 node graphs get 60 seconds, and 500 node graphs get 300 seconds. Each heuristic is run on the three graphs for the associated time, and the final solution distance is recorded. The graph density is recorded to identify which instance of a graph the heuristic was ran on.

5.3 Heuristics

5.3.1 Simulated Annealing

For the simulated annealing algorithm, I used a path representation of a cycle. Each possible successor is chosen by swapping two nodes in the path. For example, *b* could be a possible successor of *a.*

a.) $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$

b.) $1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$

After a possible successor is chosen, its fitness (path distance) is compared against the current solution. If the successor is more fit than the current, then the algorithm sets the current to the successor. Otherwise, it accepts the successor with a probability of $e^{-\frac{E}{T}}$, where E is the difference between the successor's fitness and the current fitness and T is the current temperature. At the end of each iteration, the algorithm uses a geometric reduction cooling schedule.

$$t_{i+1} = t_i * 0.99$$

5.3.2 Genetic Algorithm

In the genetic algorithm, a chromosome of an individual is the path representation. Selection is performed by taking the top half of the population, carrying them over, and then breeding them. The crossover method is an ordered crossover. The ordered crossover works by first selecting two crossover points.

$$P_1 = 1 \rightarrow 2 \rightarrow |3 \rightarrow 4 \rightarrow 5| \rightarrow 6$$

$$P_2 = 2 \rightarrow 4 \rightarrow |6 \rightarrow 1 \rightarrow 3| \rightarrow 5$$

It then swaps the genes in the crossover section. After swapping, the rest of the chromosome is filled in with genes from the other parent, skipping over genes that already exist in the offspring.

$$O_1 = X \rightarrow X \rightarrow |6 \rightarrow 1 \rightarrow 3| \rightarrow X$$

$$O_1 = 2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 1 \rightarrow 3$$

After filling up a new generation with offspring from crossovers, each new member of the population is mutated. Mutation is implemented by looping through the genome and swapping the current gene with any other gene with a probability called the *mutation rate*.

5.3.3 Ant Colony Optimization

The ant colony optimization (ACO) algorithm is based on Ant System. The colony is typically comprised of 5 – 10 ants, depending on the size of the graph. Pheromones are initially set to 0.0001 for all edges. During each iteration, every ant makes a tour, then the pheromones are updated. An ant has a

probability proportional to the amount of pheromone on an edge and inversely proportional to the distance of the edge.

$$P(edge) = \frac{h(edge)}{\sum h(edge)}$$

$$h(edge) = \frac{\pi(edge)^{\alpha}}{\mu(edge)^{\beta}}$$

In the formulas above, P is the probability of an ant taking an edge, and h is a function that returns the heuristic of an edge. The pheromone on an edge, π, is raised to α, which can be changed to increase or decrease the importance of a strong pheromone. Likewise, the metaheuristic, μ, of an edge can independently be affected by β. μ is just $\frac{1}{distance(edge)}$ and can be calculated *a priori.*

At the end of each iteration, the pheromones are updated. First, they are evaporated by a geometric reduction of 0.95, then the pheromone of each ant is added to the colony's graph. Each ant produces $\frac{1}{distance(tour)}$ of pheromone per tour. Each edge in the respective tour is increased by this value.

5.4 Implementation and Dataset

I ran each test 15 times, resulting in 195 datapoints–60 points from the time test and 45 points from each graph in the performance test. At the end of each test, the solution type (e.g., Genetic Algorithm), graph density, solution performance, and solution runtime is recorded. These data are stored as comma separated values (CSV). I used Python to implement these heuristics and tested them on a Windows PC with an AMD Ryzen 9 6000 series processor with 16 GB of RAM.

6 Results

After completing the two tests, I received enough data to create several charts (see section 10). Both tests offered insight into various aspects of the implementations. The time test showed that the only heuristic that closeness significantly affected was SA. This may be explained by the fact that a denser graph would not have such a significant change when using the swap operator: two nodes would not be that far apart, so swapping them would not significantly change the distance.

ACO outperformed GA and SA in every test. During the time test, ACO was orders of magnitude quicker than the other two implementations, which were similar. ACO also outperformed SA, which outperformed GA, in the performance test.

7 Conclusion

ACO is the best unoptimized heuristic of the ones I compared. It offered quicker, more accurate approximations for TSP-OPT. ACO was able to provide 10 to 20 times better solutions for 500 nodes. When it comes to practical routing algorithms, even small decreases in solution distance can potentially save companies millions of dollars. This means that if an algorithm can produce solutions that are orders of magnitude better than its competitors, it would be an extremely useful tool.

8 Future Work

This project could be taken in several directions moving forward. It could be compounded by testing for different heuristics to compare additional strategies for solving TSP-OPT. Additional graphs could be added to the test data. The graphs I used were empty graphs (had no edges) and were filled using the heuristics. Adding graphs that had predefined edge could affect the results. While a problem like wire planning can easily be represented by an empty graph, something such as routes taken with roads would be better represented with a graph with edges: you can't drive directly from every city to every other city.

My implementations could be improved, optimizing the heuristics to get better approximations. This could potentially give better approximations than my ACO implementation, which was the goal of the project. Further using my implementations, it would be possible to create a hybrid heuristic, such as a genetic ant colony.

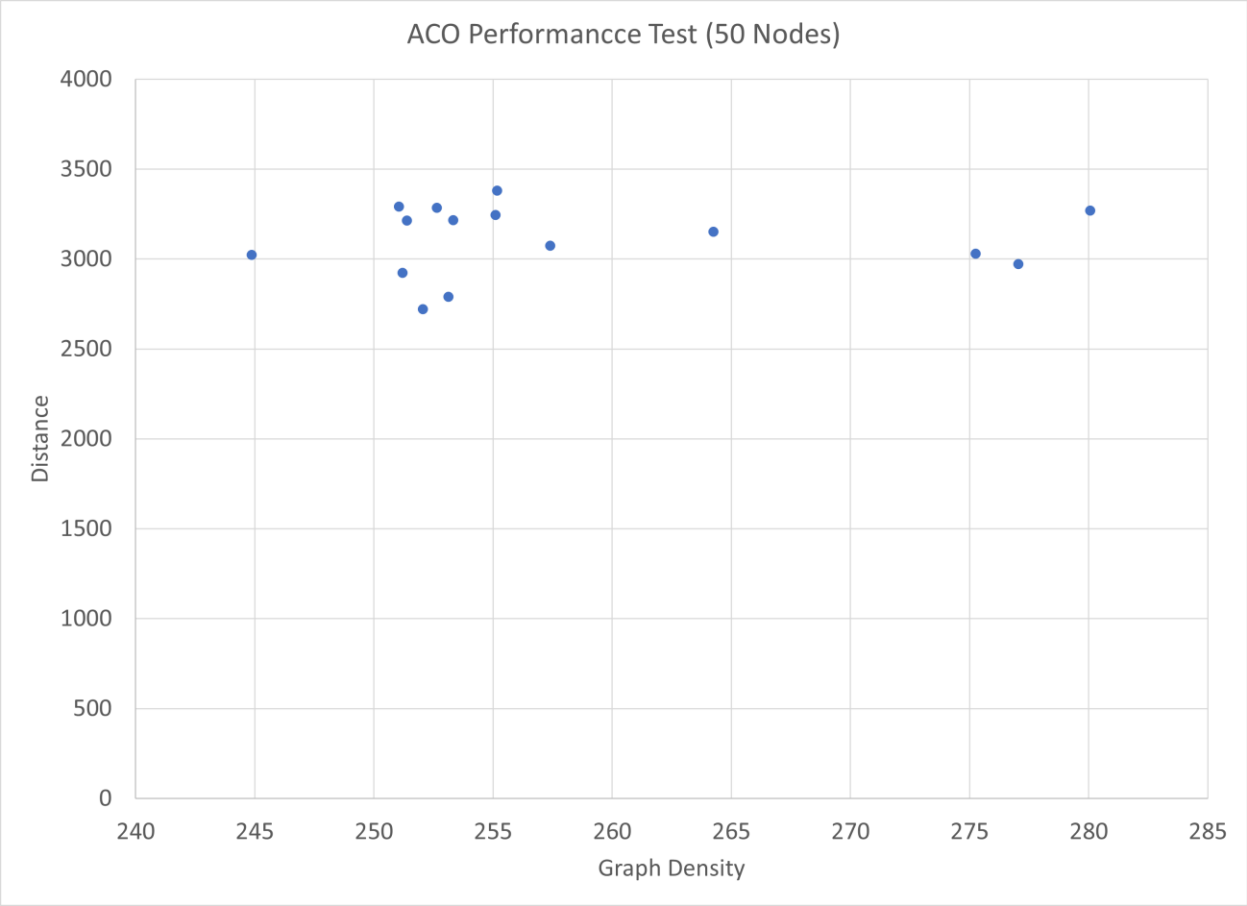
9 References

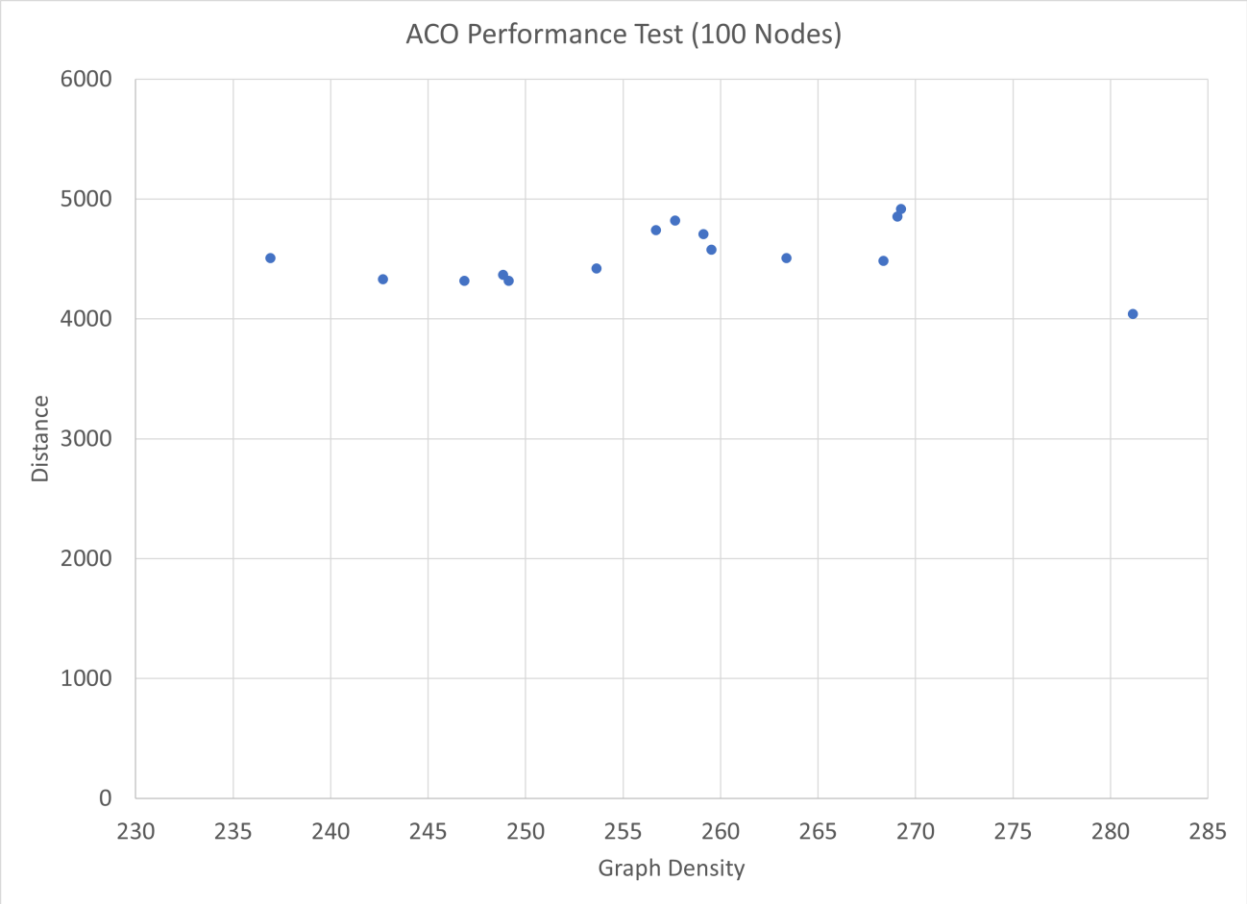
Dorigo, M., & Stuzle, T. (2004). *Ant Colony Optimization.* Cambridge: MIT Press.

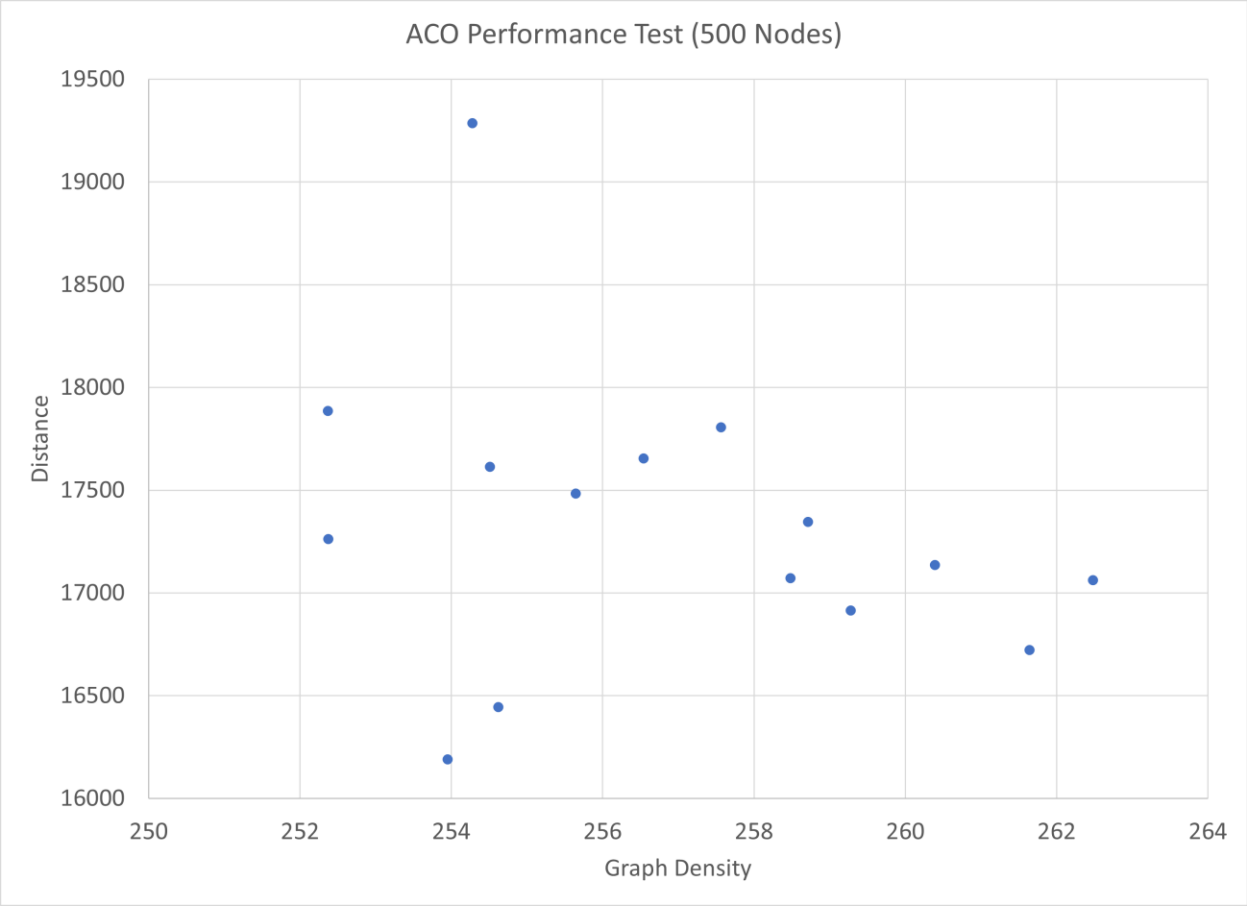
Hussain, A., Muhammad, Y. S., Sajid, M. N., Hussain, I., Shoukry, A. M., & Gani, S. (2017, October 25). Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator. *Computational Intelligence and Neuroscience*.

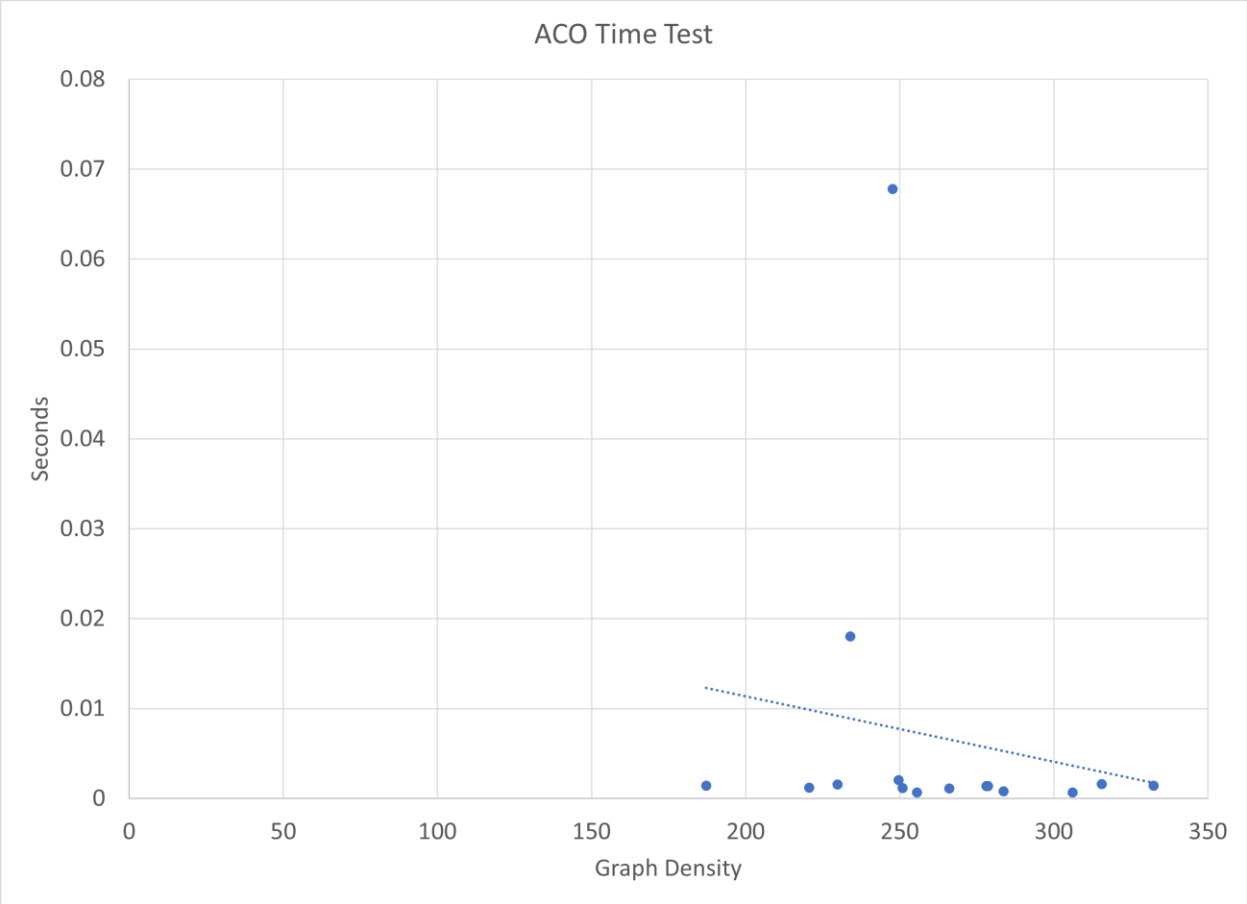
Zhan, S. L. (2016, March 13). List-Based Simulated Annealing Algorithm for Traveling Salesman Problem. *Computational Intelligence and Neuroscience*.

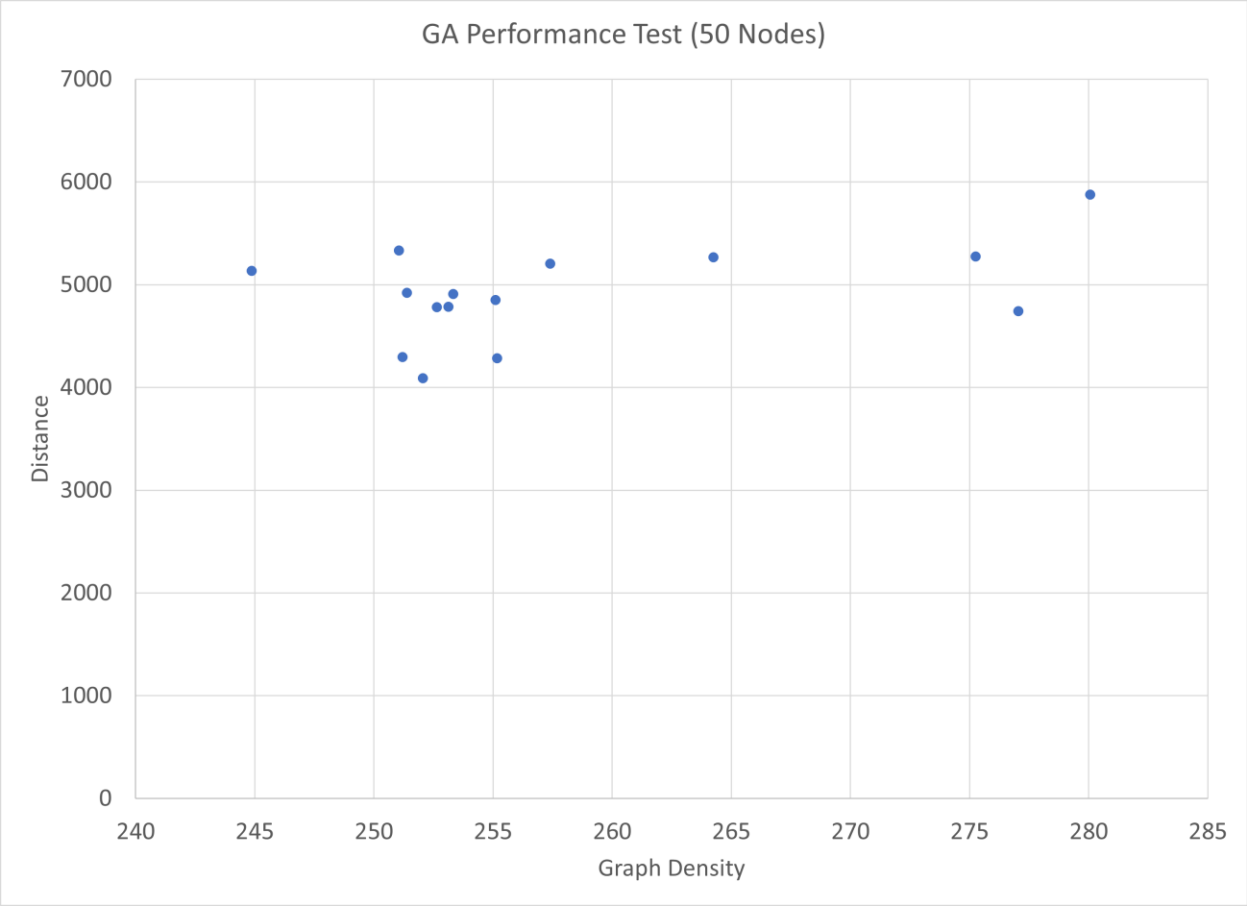
10 Figures

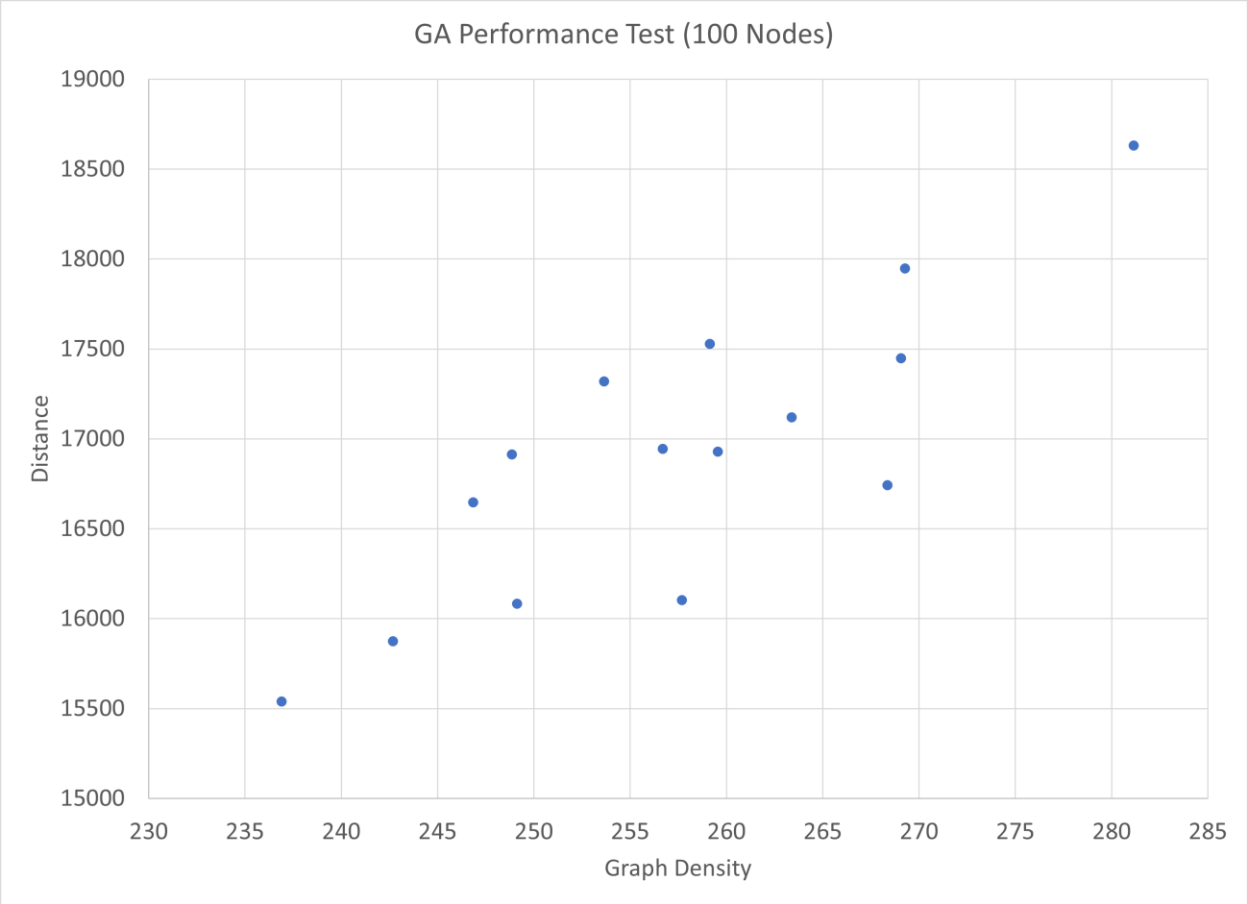

ACO Performancce Test (50 Nodes)
Distance
4000
3500
3000
2500
2000
1500
1000
500
0
240
245
250
255
260
265
270
275
280
285
Graph Density

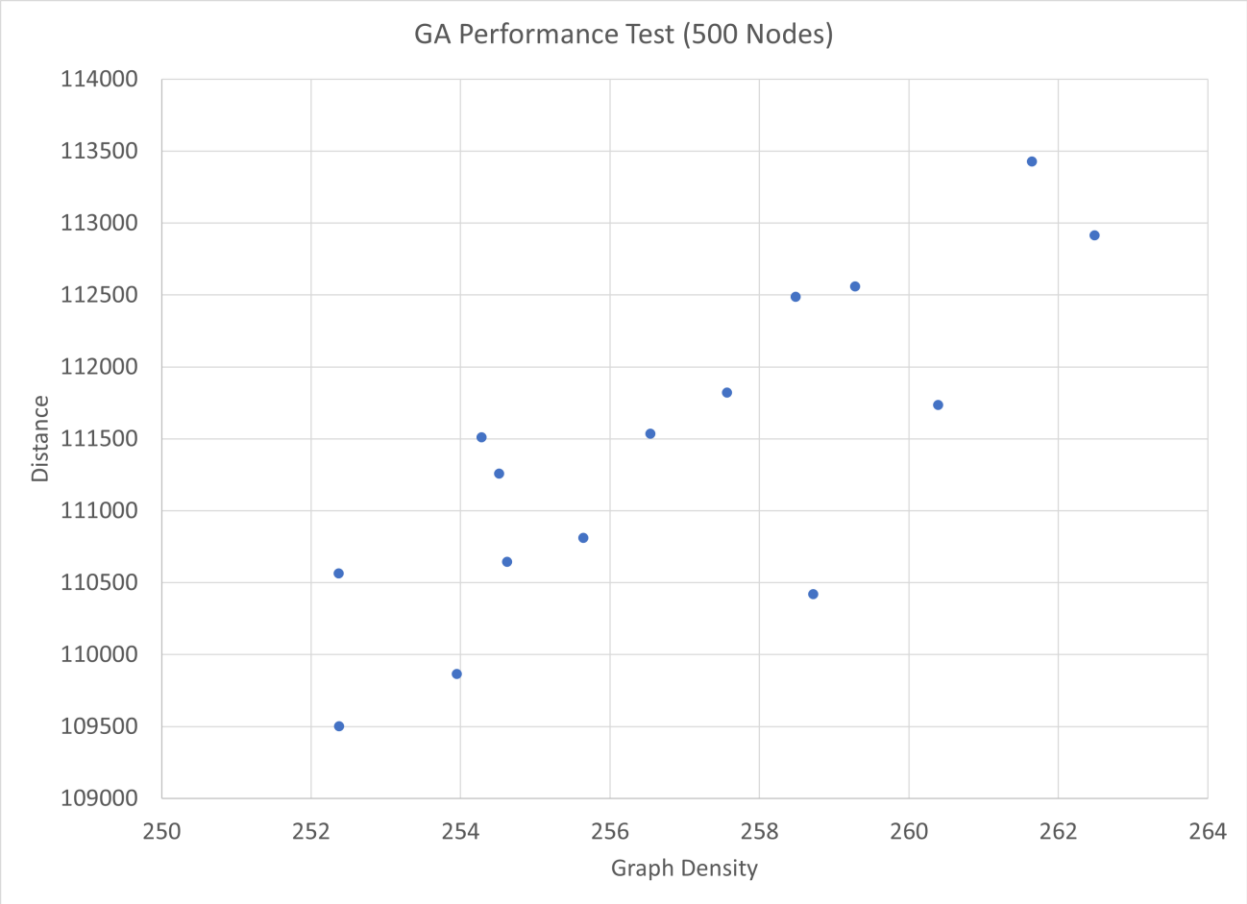

ACO Performance Test (100 Nodes)
Distance
6000
5000
4000
3000
2000
1000
0
230
235
240
245
250
255
260
265
270
275
280
285
Graph Density

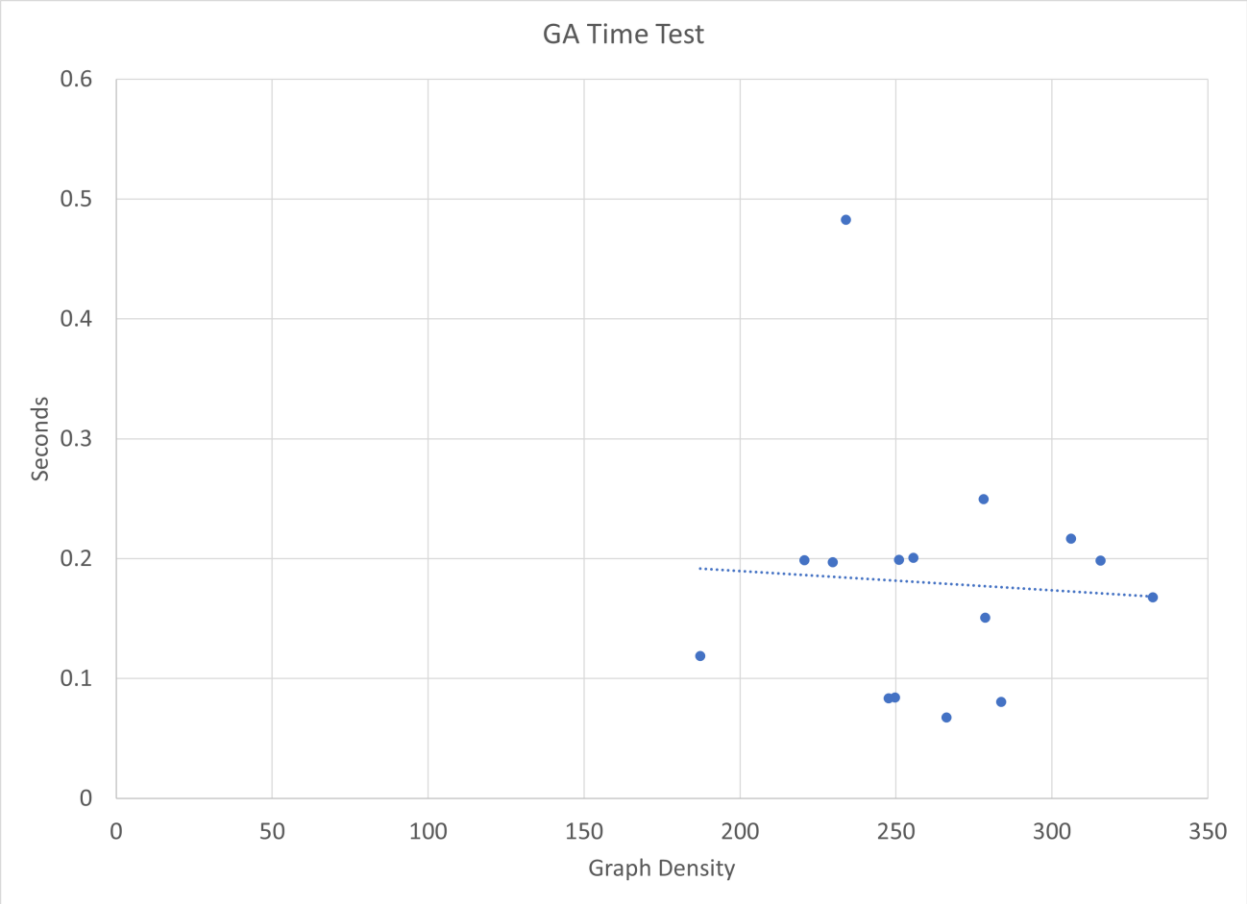

ACO Performance Test (500 Nodes)
19500
19000
18500
18000
17500
17000
16500
16000
Distance
250
252
254
256
258
260
262
264
Graph Density

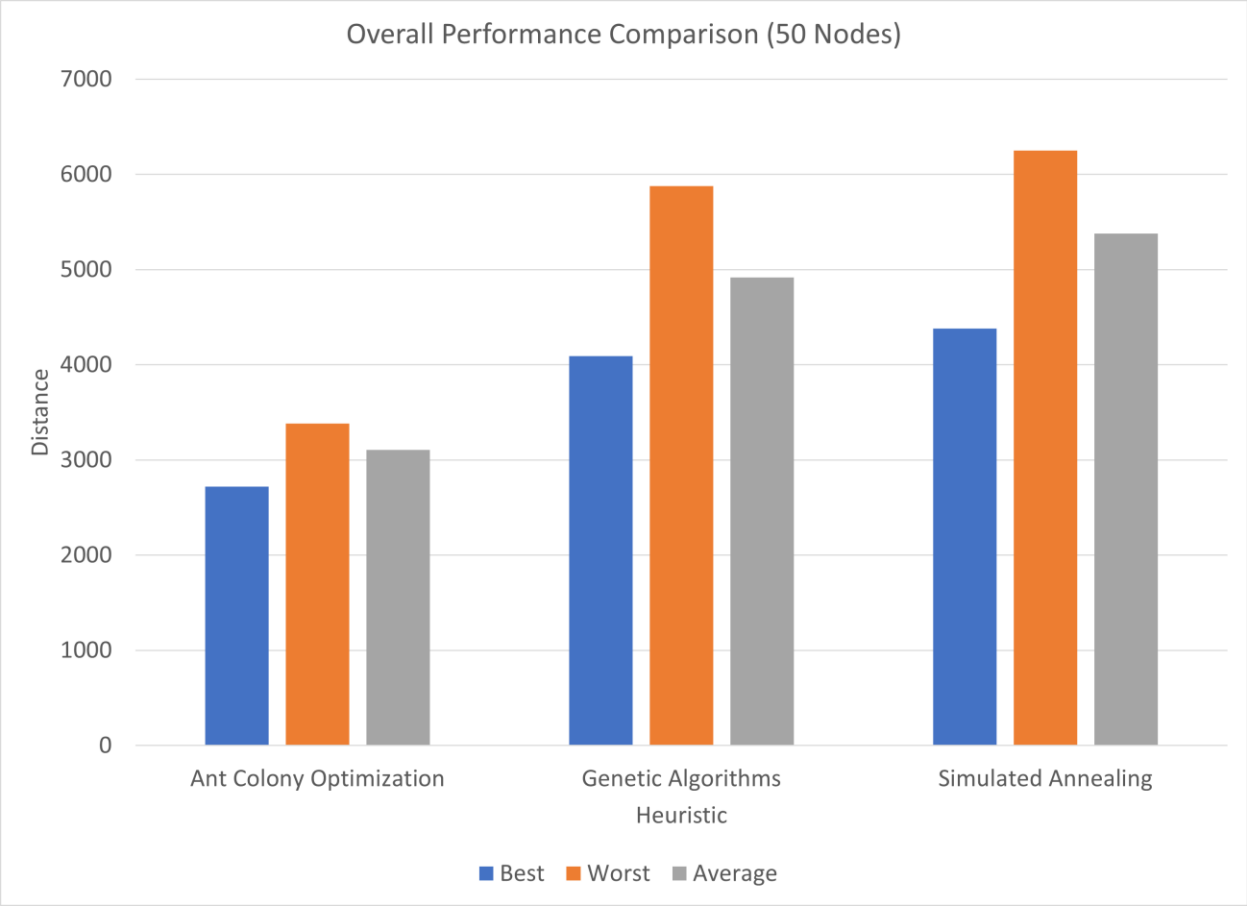

ACO Time Test
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
Seconds
0
50
100
150
200
250
300
350
Graph Density

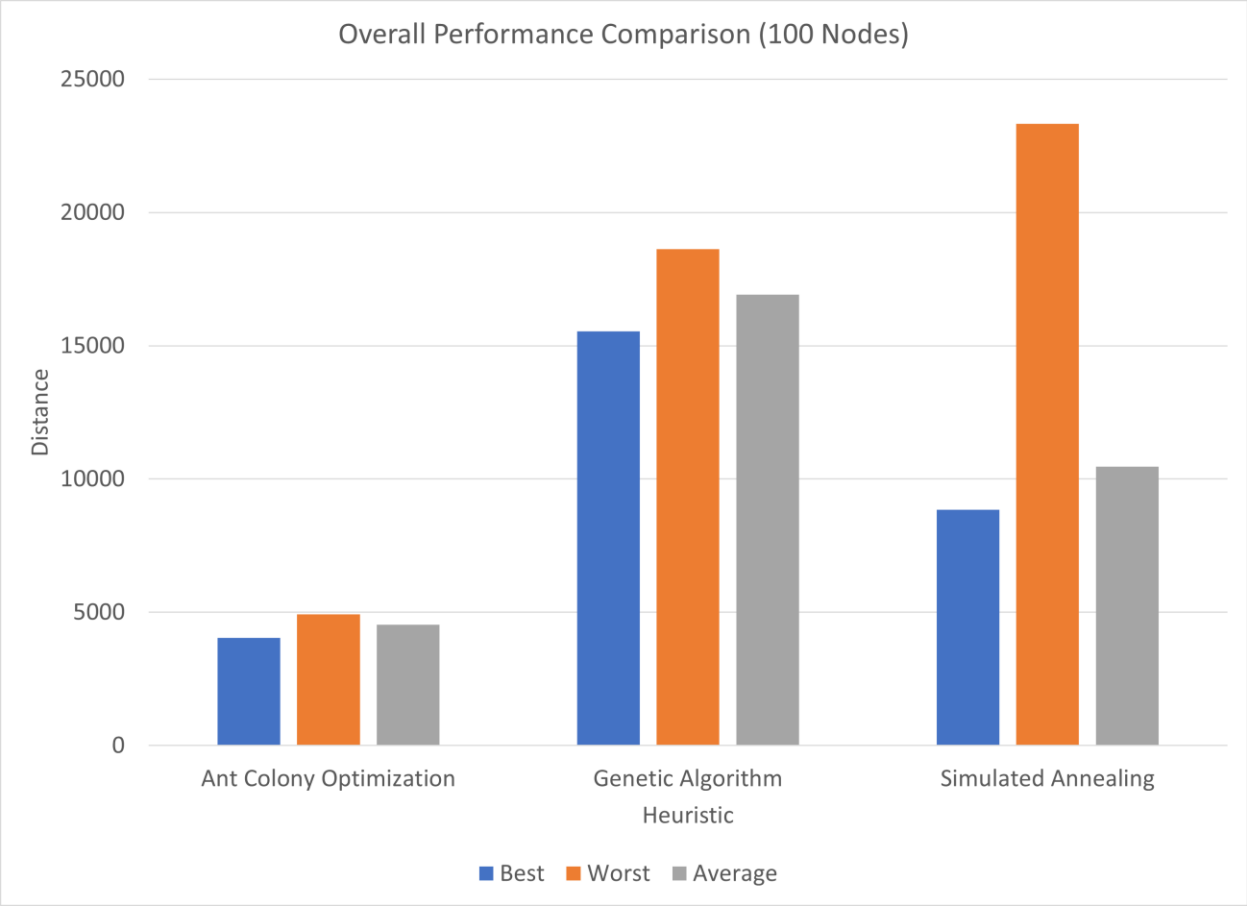

GA Performance Test (50 Nodes)
Distance
7000
6000
5000
4000
3000
2000
1000
0
240
245
250
255
260
265
270
275
280
285
Graph Density

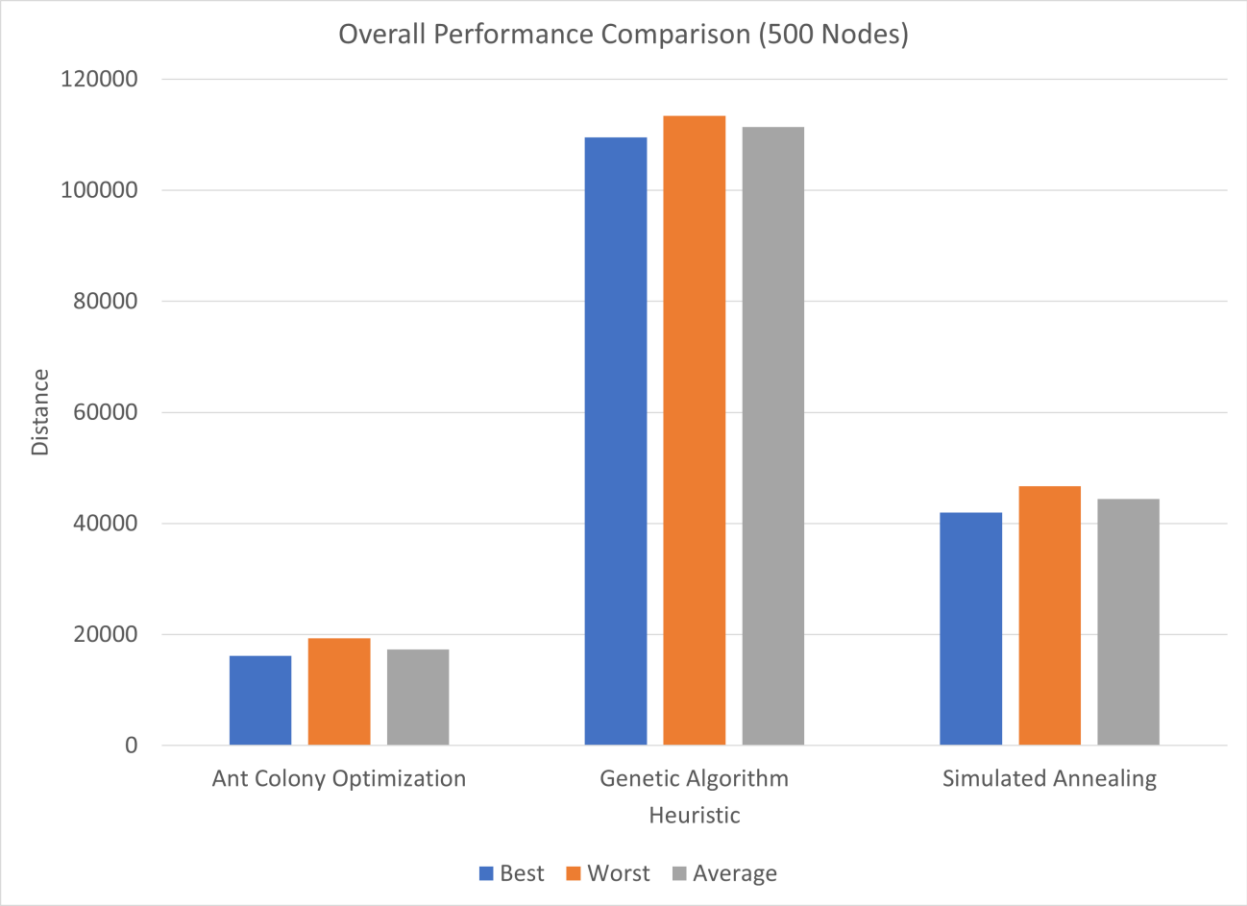

GA Performance Test (100 Nodes)
Distance
19000
18500
18000
17500
17000
16500
16000
15500
15000
230
235
240
245
250
255
260
265
270
275
280
285
Graph Density

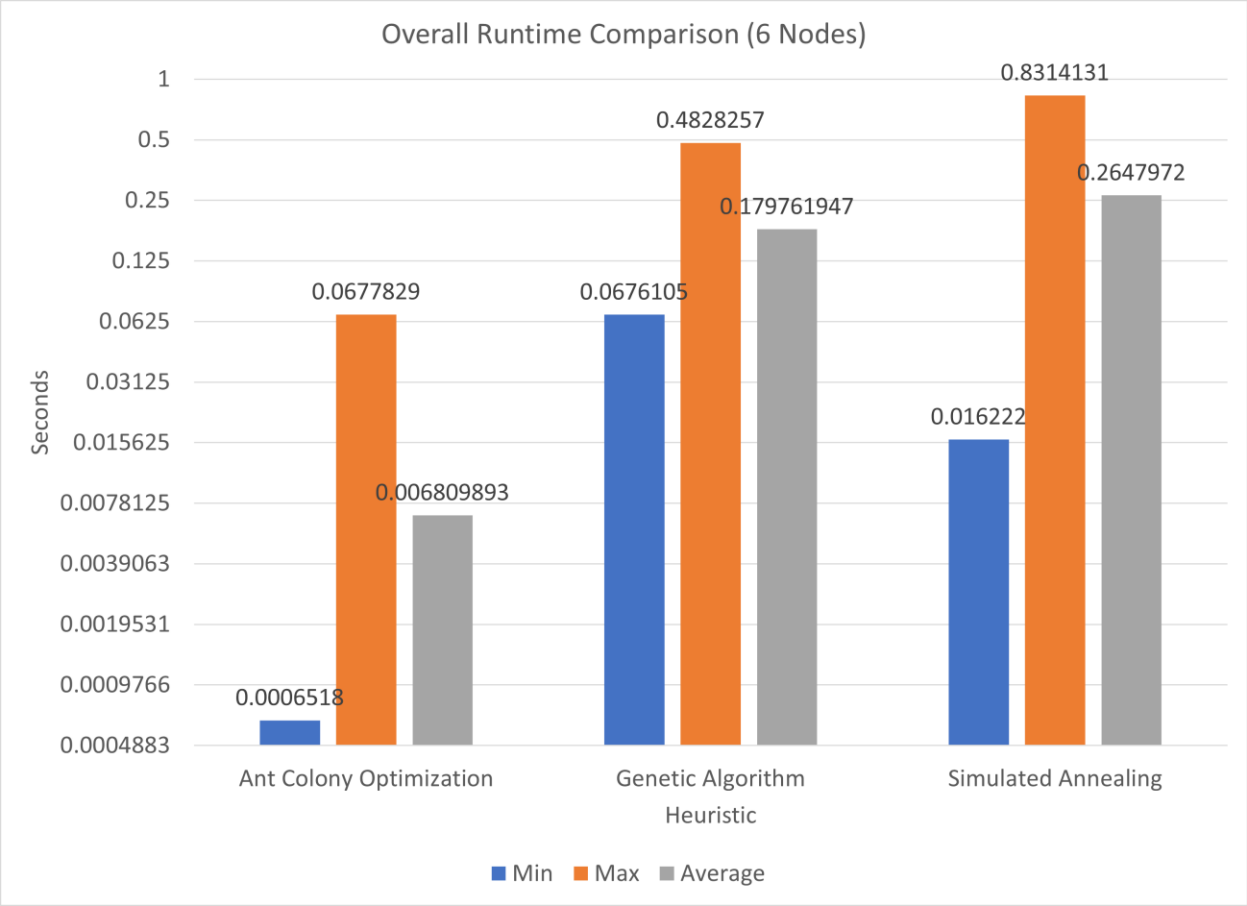

GA Performance Test (500 Nodes)
Distance
114000
113500
113000
112500
112000
111500
111000
110500
110000
109500
109000
250
252
254
256
258
260
262
264
Graph Density

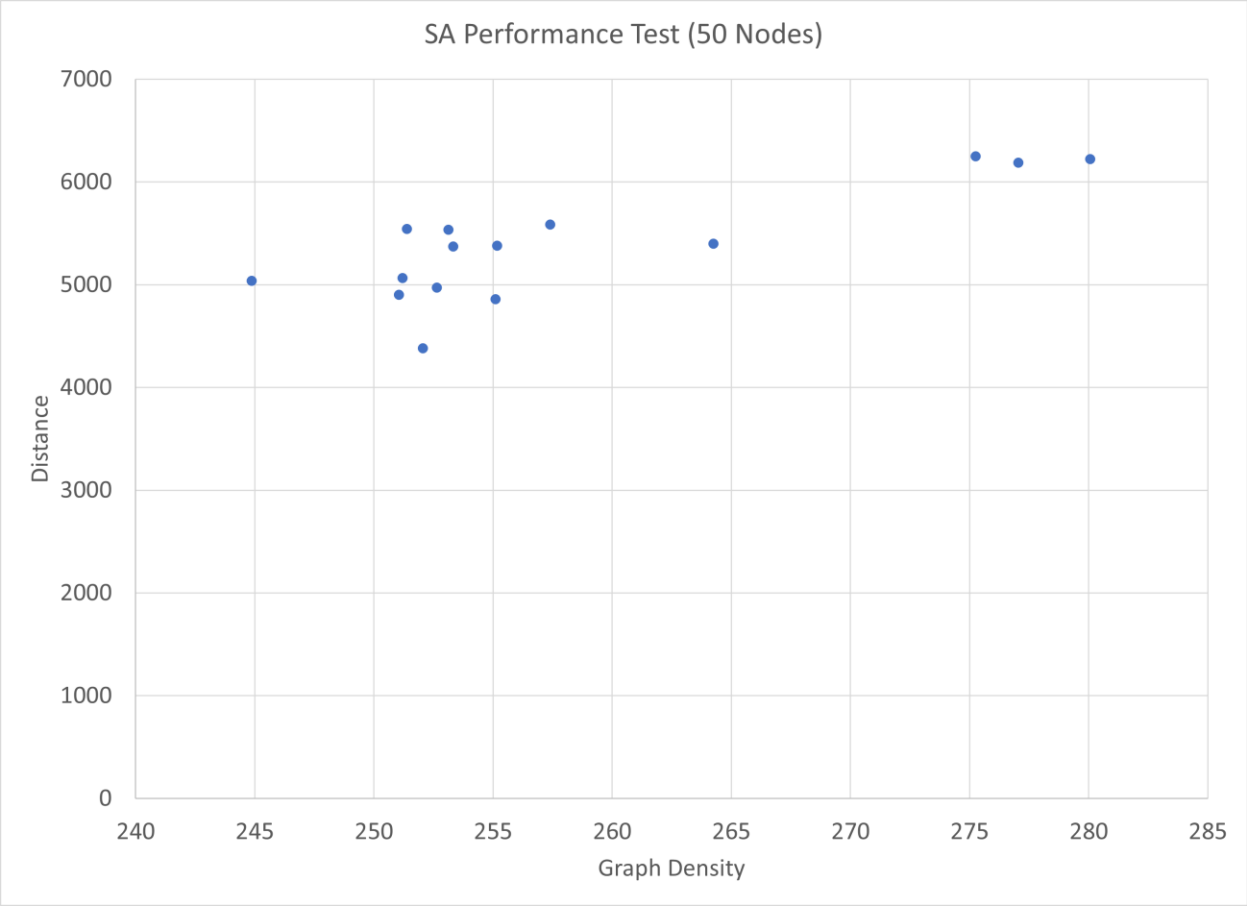

GA Time Test
0.6
0.5
0.4
0.3
0.2
0.1
0
Seconds
0
50
100
150
200
250
300
350
Graph Density

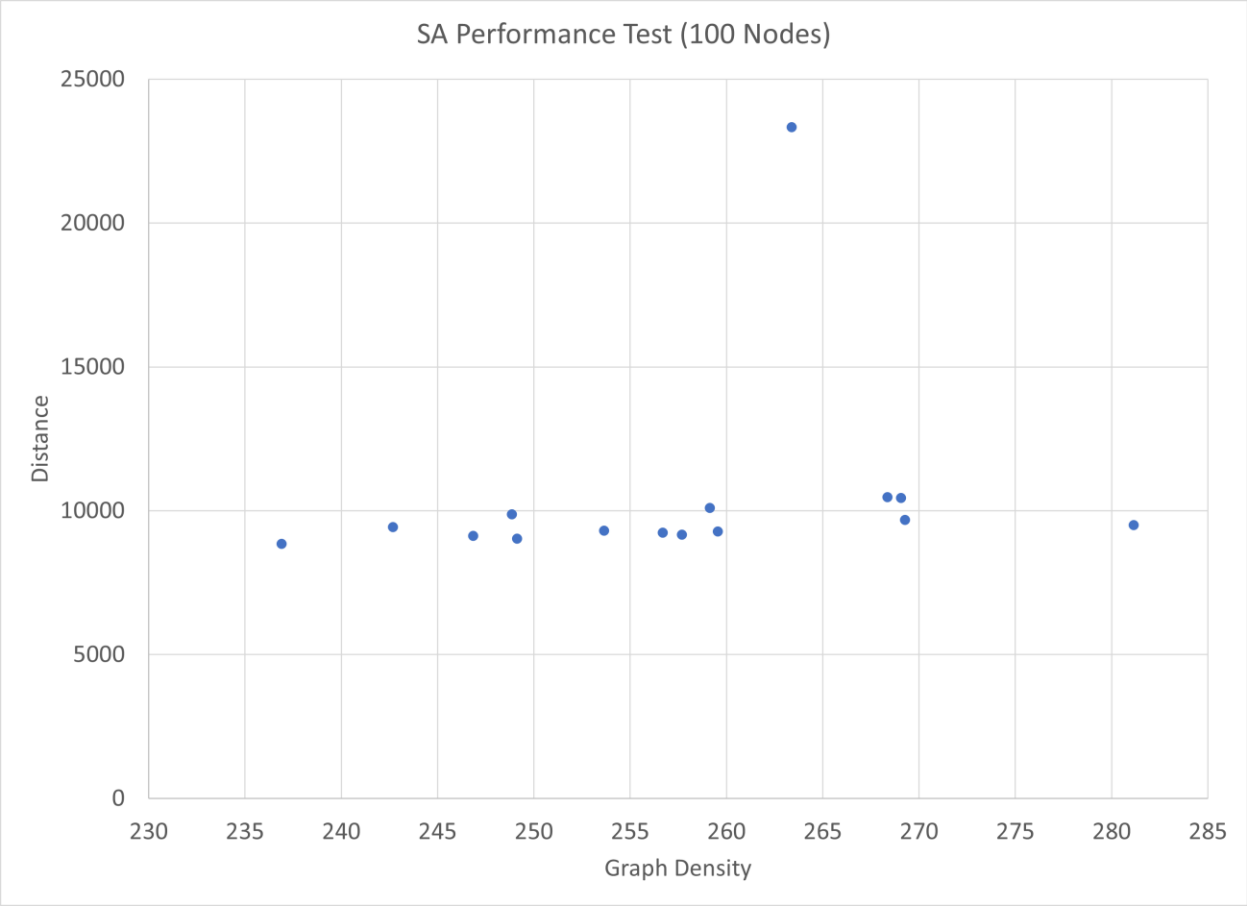

Overall Performance Comparison (50 Nodes)
7000
6000
5000
4000
3000
2000
1000
0
Distance
Ant Colony Optimization
Genetic Algorithms
Simulated Annealing
Heuristic
Best
Worst
Average

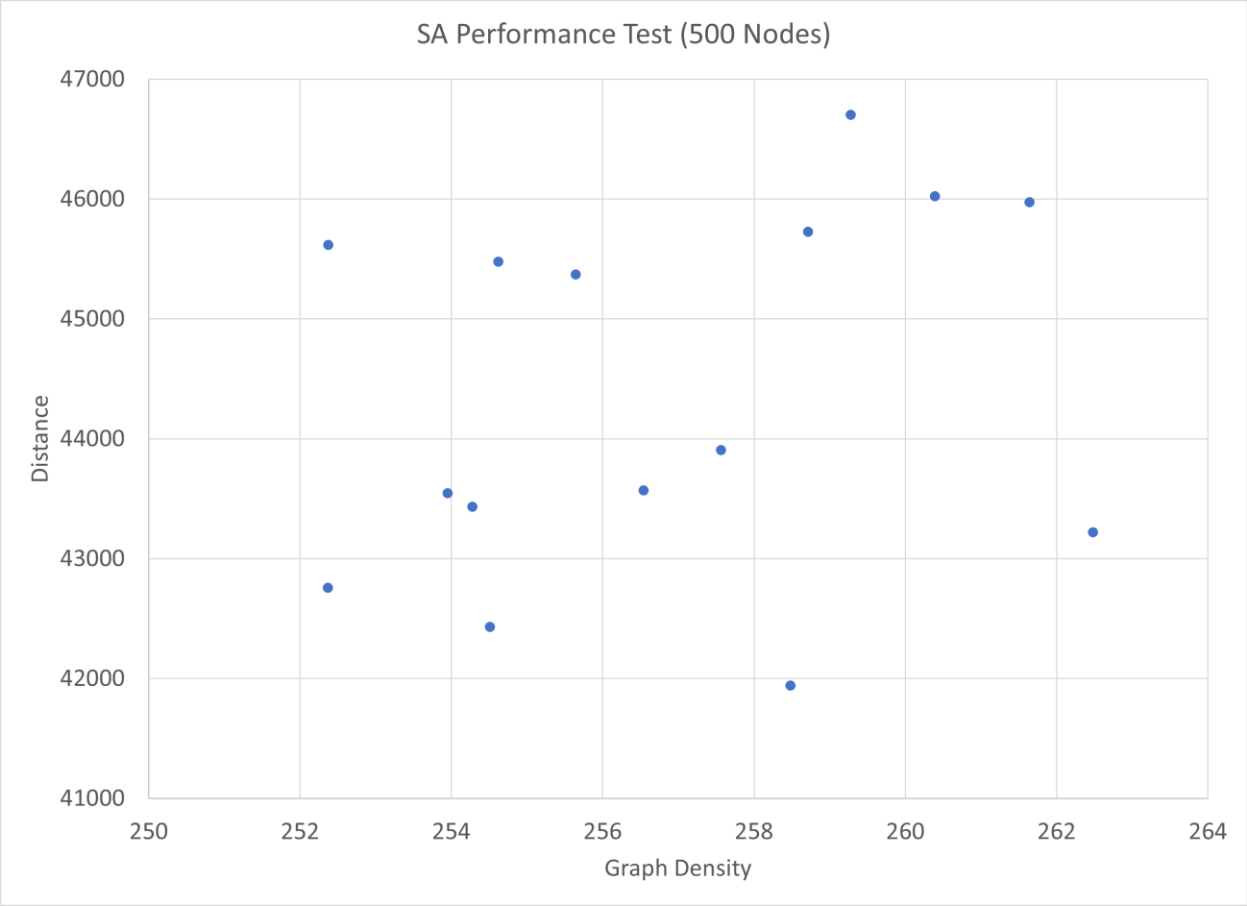

Overall Performance Comparison (100 Nodes)
25000
20000
15000
10000
5000
0
Distance
Ant Colony Optimization
Genetic Algorithm
Simulated Annealing
Heuristic
Best
Worst
Average

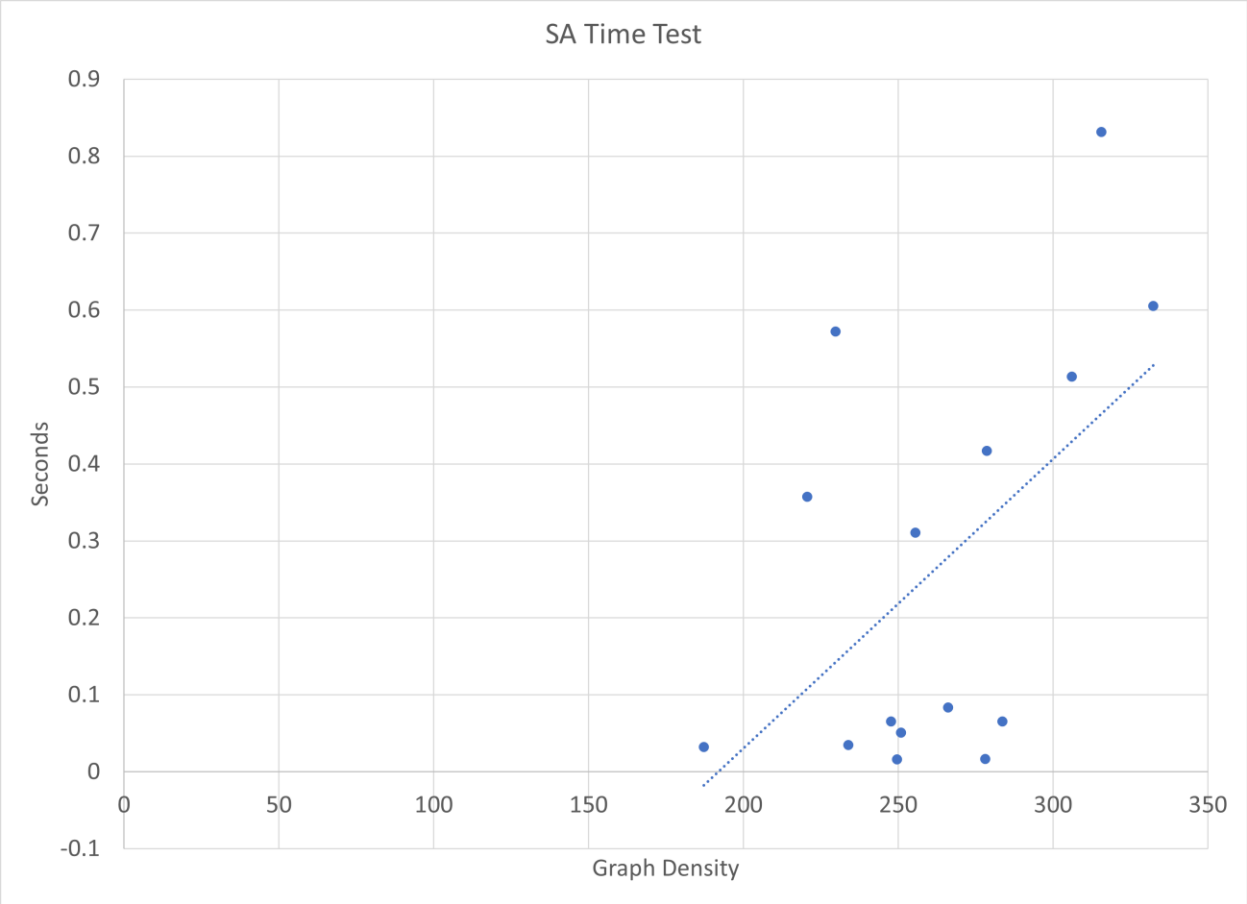

Overall Performance Comparison (500 Nodes)
120000
100000
80000
60000
40000
20000
0
Distance
Ant Colony Optimization
Genetic Algorithm
Simulated Annealing
Heuristic
Best
Worst
Average


Overall Runtime Comparison (6 Nodes)
Seconds
1
0.5
0.25
0.125
0.0625
0.03125
0.015625
0.0078125
0.0039063
0.0019531
0.0009766
0.0004883
0.0006518
0.0677829
0.006809893
0.0676105
0.4828257
0.179761947
0.016222
0.8314131
0.2647972
Ant Colony Optimization
Genetic Algorithm
Simulated Annealing
Heuristic
Min
Max
Average


SA Performance Test (50 Nodes)
Distance
7000
6000
5000
4000
3000
2000
1000
0
240
245
250
255
260
265
270
275
280
285
Graph Density


SA Performance Test (100 Nodes)
25000
20000
15000
10000
5000
0
Distance
230
235
240
245
250
255
260
265
270
275
280
285
Graph Density


SA Performance Test (500 Nodes)
47000
46000
45000
44000
43000
42000
41000
Distance
250
252
254
256
258
260
262
264
Graph Density


SA Time Test
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-0.1
Seconds
0
50
100
150
200
250
300
350
Graph Density