Optimization Algorithms for Traveling Salesman Problem: A Comparison of Basic Ant System and Ant Colony Optimization-Genetic Algorithm Hybrid

Michael A. Webber, Andrew W. Stoneman, and Alexander L. Clark

Keywords: Ant Colony Optimization (ACO), Traveling Salesman Problem (TSP), NP-hard, Optimal Tour, Genetic Algorithm (GA)

Abstract: The Traveling Salesman Problem (TSP) is a standard optimization problem that aims to minimize the path of a salesman traveling between a collection of cities. A completed TSP occurs when the salesman finds the most optimal path or a near-optimal solution. In this experiment, we test the Basic Ant System Ant Colony Optimization (ACO) algorithm against two Ant Colony Optimization-Genetic Algorithm (ACO-GA) hybrid configurations to try and find a solution closest to the most optimal solution for TSP. Both algorithms operate using the same form of Ant Colony Optimization, where, using heuristics and a digitized version of a chemical called pheromone that ants use to find food sources, they search for the most optimal path. However, ACO-GA adds a genetic algorithm to the process, carrying tours through an evolutionary process. We compare these algorithms' performance based on various factors such as the most optimal tour they can achieve and how quickly they can reach an optimal range. We will also determine the success of an algorithm by measuring which performed best on different population sizes. Our results show that BAS barely outperforms one of the ACO-GA hybrids on smaller file sizes, but this same ACO-GA hybrid outperforms BAS on larger file sizes, and the other ACO-GA hybrid performs quite poorly.

1 INTRODUCTION

In this study, we are implementing a hybrid algorithm made up of a Genetic Algorithm (GA) and an Ant Colony Optimization (ACO) algorithm in order to compare it against the Basic Ant System (BAS) ACO developed in our paper "Ant Colony Optimization for the Traveling Salesman Problem." We will be testing these algorithms for solving the Traveling Salesman Problems (TSP). TSP is a common problem in computer programming where a salesman is trying to visit every "city" in a list of cities and return to the starting city in the shortest distance possible. For our combined algorithms, we used 2 different recombination methods, ordered crossover (OX) and partially mapped crossover (PMX), making a total of three algorithms we will be testing.

To perform this study we ran two different experiments across four different file sizes for the three algorithms. First, we specified the number of iterations to see how close the algorithm

gets to optimal within the specified iterations. Secondly, we measured the number of iterations it took to reach a specified percentage over the optimal tour length, allowing us to specify how close the algorithm should get before stopping.

The results of the tests were quite promising for the ACO-GA with OX. There were clear relationships between the accuracy of our ACO-GA with OX hybrid and the original BAS through the various test files, as their average optimal solutions were quite close to one another. Towards the later files, there is a slight difference between the ACO-GA with OX hybrid and the original BAS, where ACO-GA with OX had a better performance. The ACO-GA with PMX algorithm did not perform as well and the run time was also consistently slower.

In Section 2, we will look deeper into the Traveling Salesman Problem and dissect the entirety of the problem. Section 3 will explore the meaning of Ant Colony Optimization and the implementation of the Basic Ant System. This leads into Section 4, where we will discuss the Genetic Algorithm in greater detail. In Section 5 we will discuss how these two algorithms will be hybridized in great detail to solve the TSP. Section 6 will discuss our experimental methods for comparing the standard ACO algorithm developed in our paper "Ant Colony Optimization for the Traveling Salesman Problem," with our combined ACO and GA algorithm. Section 7 will analyze the results of the experiments that we have run on our hybrid algorithm and original ACO algorithm, comparing their results. Section 8 will use what we have found in our results to show what could be done to improve our algorithm if we were given more time to continue research. Lastly, Section 9 will conclude the paper by summarizing our findings and determining the most efficient among the three ACO algorithms for solving TSP.

2 TRAVELING SALESMAN PROBLEM

The Traveling Salesman Problem (TSP) is a common problem in computer programming that is NP-hard and was first contemplated mathematically in the 1930s. An NP-hard problem is one that is at least as hard to solve as a nondeterministic polynomial-time problem. More often than not, NP-hard problems are actually much harder than NP-problems. The idea of the TSP is relatively straightforward. There is an input of a list of cities and their positions, thus a distance can be associated with the space between each city. The salesman must travel to every city and return to the city that he or she started at, the goal being to travel the shortest distance required to visit all cities. Constructing this path becomes obviously complicated quickly, as the number of potential tour subsets drastically increases every time a city is added to the problem.

This problem has gotten this specific name of TSP because of the origins of the problem, but it can be applied more generally to optimization problems. The broader scope of this problem is to minimize the distance of a path through multiple different points. Other examples could be minimizing the path a bus must travel to pick up students for school, or minimizing the distance the mailman must travel in order to deliver all their mail before returning to the post office. As we can see, we will be specifically addressing the TSP, but if thought about more generally, it can apply to a variety of different optimization problems that take place every day.

3 ANT COLONY OPTIMIZATION

ACO is, as the name suggests, inspired by how ant colonies function when scavenging for food sources. When an ant within the colony finds a food source, he brings it back to the colony, and leaves a trace of chemicals called "pheromones," which other ants can discover, and follow the path the chemicals create to the food source. These pheromones evaporate over time, but if more ants are continually visiting a food source, more pheromones will be laid down, creating an ideal path to follow for all other ants in the colony. If the food source were to run out, then fewer ants would visit it, and the pheromones would evaporate so that other ants do not continue to visit a poor food source.

In the context of the TSP problem, a path between two cities is analogous to a path between a potential food source and the colony. This path between two cities is referred to as a "leg" in a tour, and every leg in the TSP contains a certain amount of pheromone, which is roughly based on the length of a tour that an ant had in which that leg was a part. After every iteration, the pheromone is updated, both being evaporated and deposited based on how desirable of a leg it was. This is based on if it is included in a good or bad tour or not utilized at all.

In this problem, an ant starts at a random city, repeatedly chooses its next city based on the level of pheromones and distance on the leg from its current city to a potential next, and once it has visited all cities, it completes its tour. This is performed on a set number of ants, (20 ants were used in this experiment), and once an iteration completes, the best value is determined amongst all of the ants, referring to the best tour for that iteration. This is then run for either a set number of iterations in search of an increasingly better optimal performance. After all, ants have constructed their tours for a given iteration, the pheromone levels for each leg are updated based on the performance (tour distance) of each ant that utilized that leg.

The formula seen in Figure 1 denotes the probability of ant k in city i going to city j based off of pheromone level, τ_{ij}, the inverse of the distance from i to j, $\eta_{ij} = 1/d_{ij}$, two positive constants, α and β (in this experiment, set to 1.5 and 3.5 respectively), which denote the importance of pheromones versus heuristics, and in the denominator, the sum of the cities that the ant has not yet visited, N_k, to normalize the probability.

$$p_{ij}^{k} = \frac{\tau_{ij}^{\alpha}\eta_{ij}^{\beta}}{\sum_{l \in N_k} \tau_{il}^{\alpha}\eta_{il}^{\beta}}$$

Figure 1: Probability that an ant k will move from city i to city j

After every tour, the pheromones on the legs are updated using the equation below in Figure 2. Pheromone is evaporated uniformly on all legs by a factor of 1 – ρ (where ρ equals 0.5 in this experiment). Pheromone is then deposited on each visited leg of the tour. This is exemplified by the second half of the equation in Figure 2, where we take the sum of all the times that the leg, τ_{ij}, is being used by all of the ants in the colony. Every time an individual ant includes a particular leg in its tour, that leg is deposited 1/Length of the Ant's Tour. This way the leg gets a proportional amount of pheromone to how well its tour performed. Overall, if the leg is used often by the ants, then there will be more pheromone deposited onto that leg. Conversely, if the leg is not often used, then less pheromone is going to be deposited on that leg. The changes in the pheromone levels alter the probabilities of selecting that leg in the next tour, ideally pointing an ant to a leg included in the most optimal solution. The pheromones are updated based on the number of ants, m, the pheromone levels on a leg from i to j, τ_{ij}, and an evaporation factor, ρ.

$$\tau_{ij} = (1 - \rho)\tau_{ij} + \Sigma_{k=1}^{m} \Delta\tau_{ij}^{k}$$

Figure 2: Updating the pheromone on a leg ij

4 GENETIC ALGORITHMS

Genetic Algorithms (GA) mimic natural evolution, where a certain number of individuals are represented by a particular type of data structure and are considered candidate solutions. In the case of the TSP, a candidate solution is represented as a set of cities representing a path through the cities. These candidates then go through a modified "evolutionary" process, where they are judged by a fitness model to determine the potential of their candidacy as a solution.

Next, the selection process begins, where tours are chosen based on their fitness to be put into a breeding pool. An ant's fitness is measured by the distance of its tour, where the smaller the distance, the better the fitness. The size of the breeding pool is typically equal to the number of individuals in the initial population, but this is not always the case. Oftentimes, due to the selection process, duplicates will appear in the breeding pool. There are many different selection types, including selection from tournament, ranking, and truncation, among others. Tournament selection randomly chooses two individuals from the initial population and selects the one with better fitness, adding it to the breeding pool. It repeats this process until the number of individuals in the breeding pool is equal to the size of the initial population. There are other selection types as well, including Boltzmann selection and fitness proportional selection, but we will not detail those here.

Once the selection process has been completed, candidates are put through a recombination procedure that can use multiple types of crossover operators to create two new

“children” from parent candidates. These children are then moved on to the next iteration. There are multiple different crossover types, each of which varies depending on the problem type. A common crossover operator is a one-point crossover, which works by splitting two individuals between one randomly chosen index position and swapping the second halves of each child with one another. This works well for solving MAXSAT problems but does not work for TSP, as we cannot have any duplicates in a TSP candidate, and one-point crossover does not ensure this. A few crossover types that are better suited for TSP are order crossover (OX) and partially mapped crossover (PMX). These crossovers, which are detailed in the following section, guarantee that no duplication occurs in child tours. It should also be noted that crossover only happens to some of the candidates, as the crossover rate is typically given by a specified probability, allowing some parent candidates to go untouched on to the next generation.

After crossover, the last portion of the GA is mutation. During this process, candidates in the breeding pool are susceptible to mutation, happening at a specified mutation probability that is generally kept very low. Generally speaking, mutation is supposed to make a random change to a candidate solution to keep some degree of randomness during the process. Like crossover, there are various forms of mutation depending on the problem type. For TSP, a simple mutation would be, at some infrequent probability, switching a city within a candidate solution with its neighbor in the set. While this happens very infrequently, it allows for a possibly important element of randomness to be maintained, while also introducing new genetic material to the population.

At this point, the GA has gone through one iteration, as we are left with a new set of candidate solutions. This process is repeated for a specified number of iterations, ideally making candidate solutions increasingly fit, and ultimately arriving at an optimized solution.

5 ANT COLONY OPTIMIZATION-GENETIC ALGORITHM HYBRID

5.1 Hybrid Overview

Both the Ant Colony Optimization algorithm and the Genetic Algorithm are tried and true solutions for solving optimization problems. In the past, we have used the ACO as a solution to the TSP but have only used GA when examining the MAXSAT problem. We hypothesize that by slightly modifying the GA and hybridizing it with ACO, we will arrive at a more optimal solution for the TSP than the ACO alone.

We hybridize these two algorithms by running both in every iteration. The way this works is by first running the basic ant system ACO (we have also referred to this as standard ant system or BAS), constructing tours for the chosen number of ants (twenty in our case), and then using each ant’s tour as a candidate solution to start the GA. It should be noted that the ACO itself has not been modified at all from its original configuration, where laying pheromones and tour construction work in the exact same manner. The only difference comes in with the GA, where after the candidate solutions from tour construction are put through a modified GA process detailed in the next section. These newly constructed tours are used to deposit pheromone, affecting how tours will be constructed in the following iterations.

5.1 The Modified Genetic Algorithm

The genetic algorithm in this hybrid version is largely similar to the standard GA but is modified in a way intended to both provide functionality for ACO and TSP, and also provide optimality.

The first primary difference between the standard GA and our hybrid GA is that it only takes the best eight ants (smallest tour distances), and, after recombination, replaces only the bottom eight ants from the original ants. In this way, the algorithm is fairly greedy, as it uses the best for recombination, but the children it creates replace the worst, thus the best is still kept in the original population, and the worst are replaced by recombinations of the best. Note that the number eight is specific to the number of ants being fixed at twenty, as it appeared to be the most optimal choice to recombine for this fixed rate. This number would certainly be changed if the number of ants was increased or decreased.

Another major part of the GA portion of this hybrid algorithm is the way in which crossover is performed in the context of TSP. As previously mentioned, the one-point crossover does not prevent duplicates from appearing in a tour, which is necessary for the TSP, so we must use different crossover operators to prevent duplicates from appearing, and for this study, we implement two.

The first crossover operator we implemented is called the Order Crossover operator (OX), which was created by Davis.[1] This operator works to maintain a parent's relative order while combining it with another parent. This works by taking two random cut points that represent a start and an end to a portion of each parent and then placing each portion into each one of the children. For simplicity, the portion in parent one is placed into child one in the same index positions, and the portion in parent two is placed into child two in the same index positions. At this point, each of our children has the same portion filled by the corresponding parent. Next, starting at the end of the portion, we iterate through a parent and place values into the opposite child in the corresponding index positions. However, while doing this, we make sure to not place any cities that are already in the child to prevent duplicates, so when we come across a duplicate city, we continue traversing through the parent but remain in the same index position in the child. In accordance with the name, we have maintained some degree of order from each parent within the children and successfully recombined them without duplicating any cities. Below are the steps of OX in figures 3-5.

Start with two parents:

$$P_1 = (8\ 3\ 4 \mid 1\ 2\ 7 \mid 5\ 6)$$

$$P_2 = (5\ 4\ 2 \mid 8\ 1\ 6 \mid 3\ 7)$$

Figure 3: Step 1 of Order Crossover

This leads to the creation of the children who maintain the middle portion of the corresponding parent:

[1] L. Davis, "Applying adaptive algorithms to epistatic domains," *IJCAI*, vol. 85, pp. 162–164, 1985.

$$C_1 = (x\ x\ x \mid 1\ 2\ 7 \mid x\ x)$$

$$C_2 = (x\ x\ x \mid 8\ 1\ 6 \mid x\ x)$$

Figure 4: Step 2 of Order Crossover

After the ordering of the opposite parent's values after the cutoff and removing the values that are already in the child, the result is:

$$C_1 = (4\ 8\ 6 \mid 1\ 2\ 7 \mid 3\ 5)$$

$$C_2 = (4\ 2\ 7 \mid 8\ 1\ 6 \mid 5\ 3)$$

Figure 5: Step 3 of Order Crossover

The second crossover operator we implemented is called the Partially Mapped Crossover operator (PMX), which was proposed by Goldberg and Lingle.[2] Like OX, this operator takes two random cut points in both parents creating a portion of them that is passed on. In this version, the portion in parent one is placed into child two and the portion in parent two is placed into child one. Next, two "partial maps" are created, one for each parent, where the keys represent the cities within the cut portion in the parent, and the values represent the corresponding cities (in the same index positions) in the other cities' cut portion. Now, two maps have been created that point to cities opposite each parent. Next, we iterate through the children and place all of the non-duplicate cities from the corresponding parent into the corresponding index positions, so parent one for child one, and parent two for child two. Now we are left with children that are mostly filled with cities, except for a few places. To fill these last positions, we check the number that should go there from the corresponding parent and look up the number that corresponds to it in the map for that parent. If the value is not a duplicate, it is placed in this position, and if it is another duplicate, this same value is checked in the map a second time, until we find a number that is not a duplicate. This same process is repeated in the other child, using the opposite parent and map. Once this process is complete, we have two newly recombined children that consist of no duplicates. Below are the steps of PMX in figures 6-9.

Start with two parents:

$$P_1 = (8\ 3\ 4 \mid 1\ 2\ 7 \mid 5\ 6)$$

$$P_2 = (5\ 4\ 2 \mid 8\ 1\ 6 \mid 3\ 7)$$

Figure 6: Step 1 of Partially Mapped Crossover

This leads to the creation of the children who maintain the middle portion of the opposite parent:

---

[2] D. Goldberg and R. Lingle, "Alleles, Loci and the Traveling Salesman Problem," in *Proceedings of the 1st International Conference on Genetic Algorithms and Their Applications*, vol. 1985, pp. 154–159, Los Angeles, USA.

$$C_1 = (x\ x\ x \mid 8\ 1\ 6 \mid x\ x)$$

$$C_2 = (x\ x\ x \mid 1\ 2\ 7 \mid x\ x)$$

Figure 7: Step 2 of Partially Mapped Crossover

The pieces where there is no conflict are then filled in:

$$C_1 = (x\ 3\ 4 \mid 8\ 1\ 6 \mid 5\ x)$$

$$C_2 = (5\ 4\ x \mid 1\ 2\ 7 \mid x\ 3)$$

Figure 8: Step 3 of Partially Mapped Crossover

Finally using mapping the result is:

$$C_1 = (2\ 3\ 4 \mid 8\ 1\ 6 \mid 5\ 7)$$

$$C_2 = (5\ 4\ 8 \mid 1\ 2\ 7 \mid 6\ 3)$$

Figure 9: Step 4 of Partially Mapped Crossover

Note that in both of these crossover operators, the portion that is cut is guaranteed to be greater than one, and also guaranteed to not be the entire length of the candidate solution. Also note that there is no crossover rate, as the eight best candidates from the initially constructed tours are guaranteed to be recombined, creating children that replace the eight worst candidates from the initially constructed tours.

Finally, while the typical GA uses mutation, we have decided to leave out this portion of the GA entirely. We implemented a form of mutation that seemed to consistently return poor results, and whenever it was removed, the optimality of the algorithm improved. Thus we made a decision to remove it from the process entirely.

Once this entire process has been completed, and the eight child ants replace the eight worst ants in the initial population, we then use this new population of ants to update pheromone in the same way the BAS does, where legs are increased in proportion to tour distance.

6 EXPERIMENTAL METHODOLOGY

After implementing the combined ACO-GA Hybrid algorithm (ACO-GA), we tested both the regular BAS, ACO-GA with OX, and ACO-GA with PMX on four different files, ranging in city size from 2,103 to 5,915. We tested each configuration fifteen times per file, calculating averages across the fifteen experiments. This allowed us to see how the ACO-GA algorithm with varying crossover operators performed across file sizes in comparison to the BAS.

The first test that we ran to compare the ACO algorithm to the two ACO-GA configurations was by giving a specific number of iterations to run and measuring the distance of the best tour found divided by the optimal tour. This test was important to our experiment as it shows the best solution each algorithm computed in relation to the optimal while under iteration constraints, rather than letting it run an infinite number of times.

The second test we ran to compare the ACO algorithm to our two ACO-GA configurations was measuring the number of iterations needed to find a specified optimal range. By optimal range, we refer to a user-specified percentage over the optimal solution (the best tour provided by the algorithm, divided by the optimal solution). This test was important to include as it gives insight into which algorithm is most efficient by demonstrating how long it takes to reach a particular point of accuracy.

Last, it should be noted that the runtime of all these algorithms was roughly the same across each, so we did not end up comparing them in these results. If there were major differences, this would be a parameter that could be taken into consideration.

7 RESULTS

Figures 10-13 depict the most optimal solution that each algorithm could come up with across all file sizes on average. This is calculated by dividing the best tour in 50 iterations by the optimal tour. Thus the smaller the number, the closer to the optimal, indicating better performance.

Figures 14-17 depict how quickly each of the algorithms can reach a specified range from the optimal distance across all files on average. The cap was set at 100 iterations to prevent the algorithm from running for an arbitrarily large amount of time. This demonstrates another way to measure the efficiency of each algorithm.

After performing tests to find the most optimal over 50 iterations, our findings suggest that the BAS algorithm performs just slightly better than the ACO-GA with OX algorithm on smaller files, but the ACO-GA with OX algorithm seems to slightly outperform the BAS algorithm on larger files. Depicted in figures 10-13 below, which show the most optimal solution over 50 iterations for each file, it is evident that as file size increases, the performance of the ACO-GA with OX becomes increasingly closer to BAS, until it finally overtakes it in file fnl4461.tsp. It increases how much it overtakes it on the next larger file rl5915.tsp.

Another finding is the poor performance of the ACO-GA with PMX algorithm in comparison to BAS and ACO-GA with OX. Though the most optimal performance fluctuated between BAS and ACO-GA with OX, these differences were typically within a range no larger than 0.017 and were even smaller on most files. As seen in figures 10-13 below, which depict the best ratio over the optimal on average for each file size, the ACO-GA with PMX returned results that were at best 0.03 from the optimal and fell as far behind as 0.14 from the optimal. This demonstrates its significantly worse performance on the TSP.

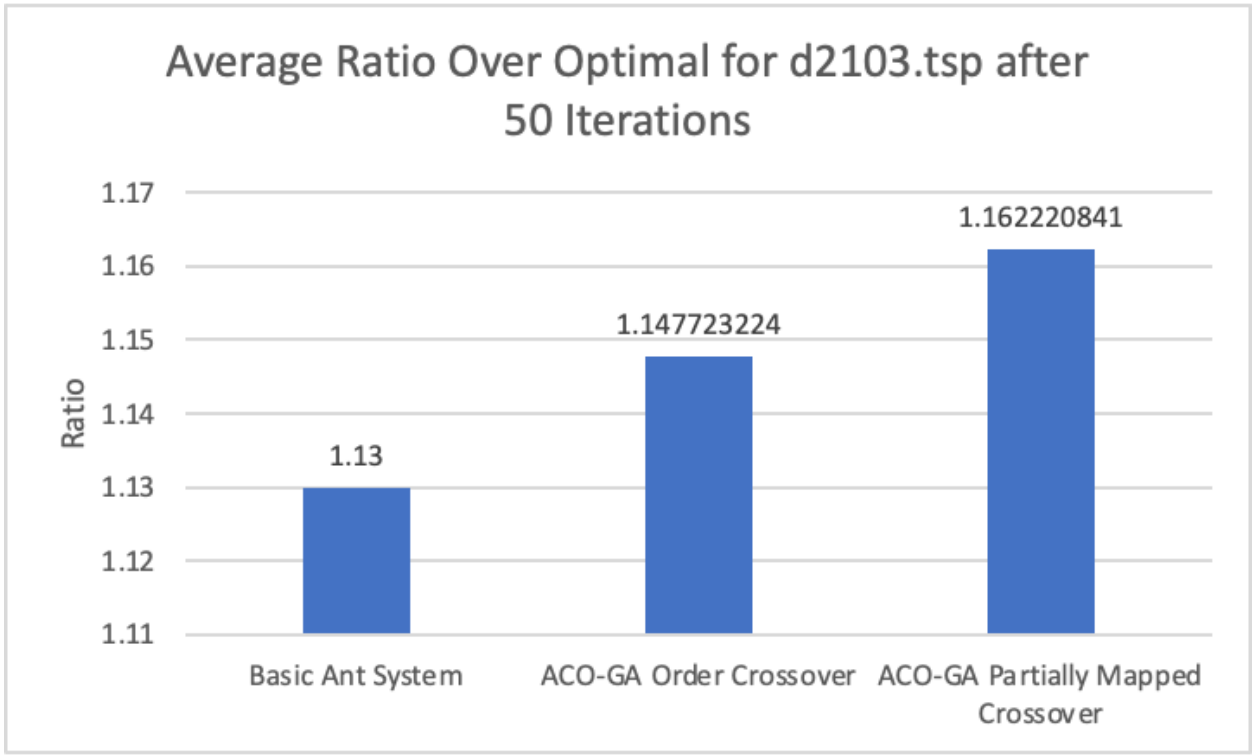

Figure 10: Best tour found divided by optimal solution for file d2103.tsp over 50 iterations

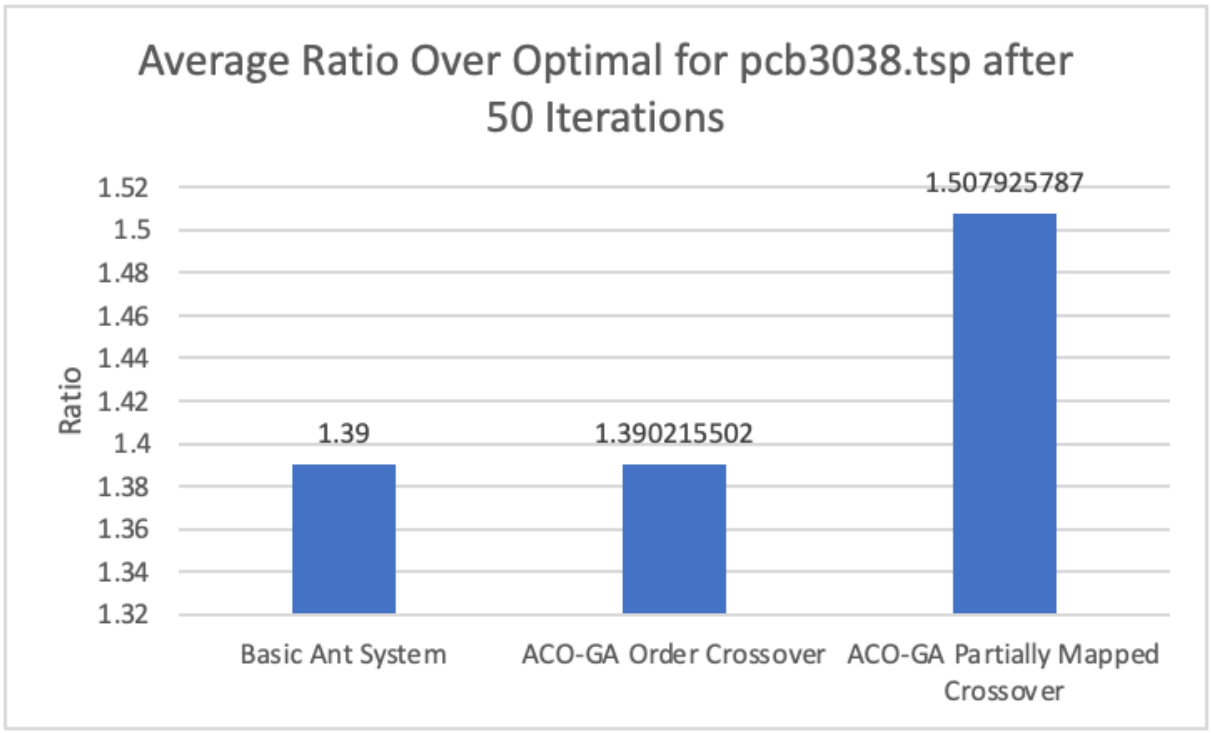

Figure 11: Best tour found divided by optimal solution for file pcb3038.tsp over 50 iterations

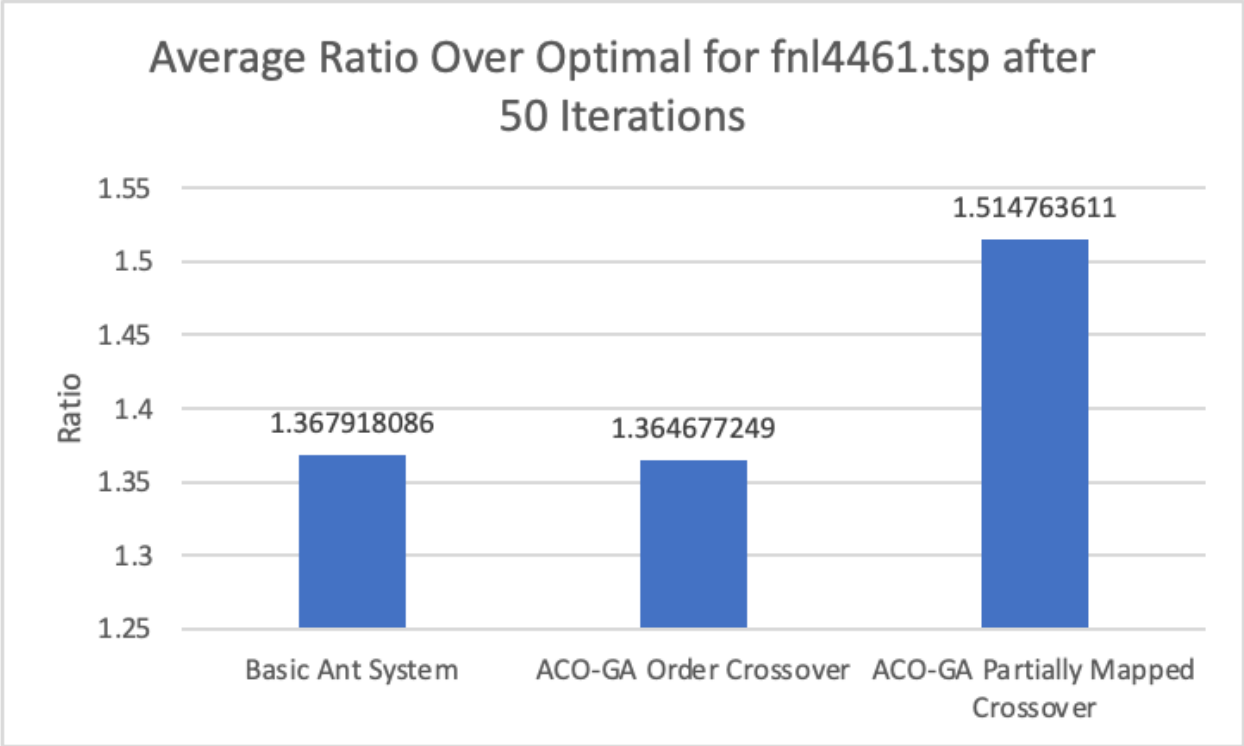

Figure 12: Best tour found divided by optimal solution for file fnl4461.tsp over 50 iterations

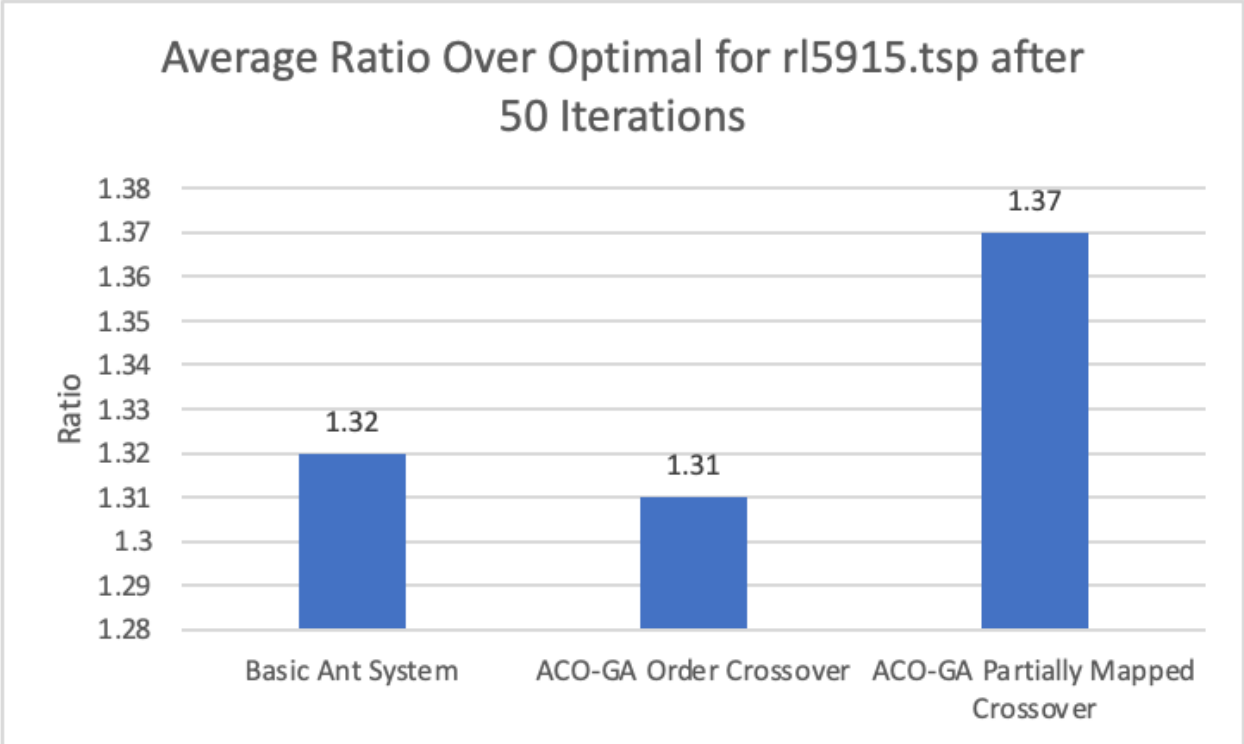

Figure 13: Best tour found divided by optimal solution for file rl5915 over 50 iterations.

It is difficult to determine exactly why we have found these results, but it can most likely be attributed to file size having an impact on performance. In the case of TSP, the larger the file size means the increase in the number of cities that a salesman must visit. When we apply this to ACO, this means that every ant's tour increases substantially with file size, since each ant must visit every city. A possible reason that ACO-GA would improve with this increase, may be that the candidate solutions, or constructed tours, are much larger, causing recombination to help more than it would with a smaller number of cities. However, this would not explain why this is only the case with OX and not PMX, but perhaps OX is just a superior form of crossover in the TSP context.

It should be noted that, while the difference in performance between BAS and ACO-GA with OX exists, these performances are so close to one another that it is difficult to fully endorse one over the other. While it appears each of their performance is correlated with file size, neither performs obviously worse than the other for any file size. However, ACO-GA with PMX consistently performs by a notably worse margin, thus we can conclude it returns less optimal solutions across all file sizes.

We also tested all of these algorithms on the iteration they arrived at a specified optimal range. This was used to determine the efficiency and speed of the algorithm. Figures 14-17 depict the iteration that an algorithm reached a specified optimal range. Almost exactly mirroring our results from the optimal performance over fifty iterations, BAS and ACO-GA with OX are consistently within a range of one iteration from the other, and ACO-GA with PMX consistently takes significantly longer in comparison. As file size increases, BAS and ACO-GA with OX stay extremely close, while ACO-GA with PMX actually increases the number of iterations it takes, as it moves from a difference of roughly 10 in file d2103.tsp to a difference of 50 in the largest file rl5915.tsp.

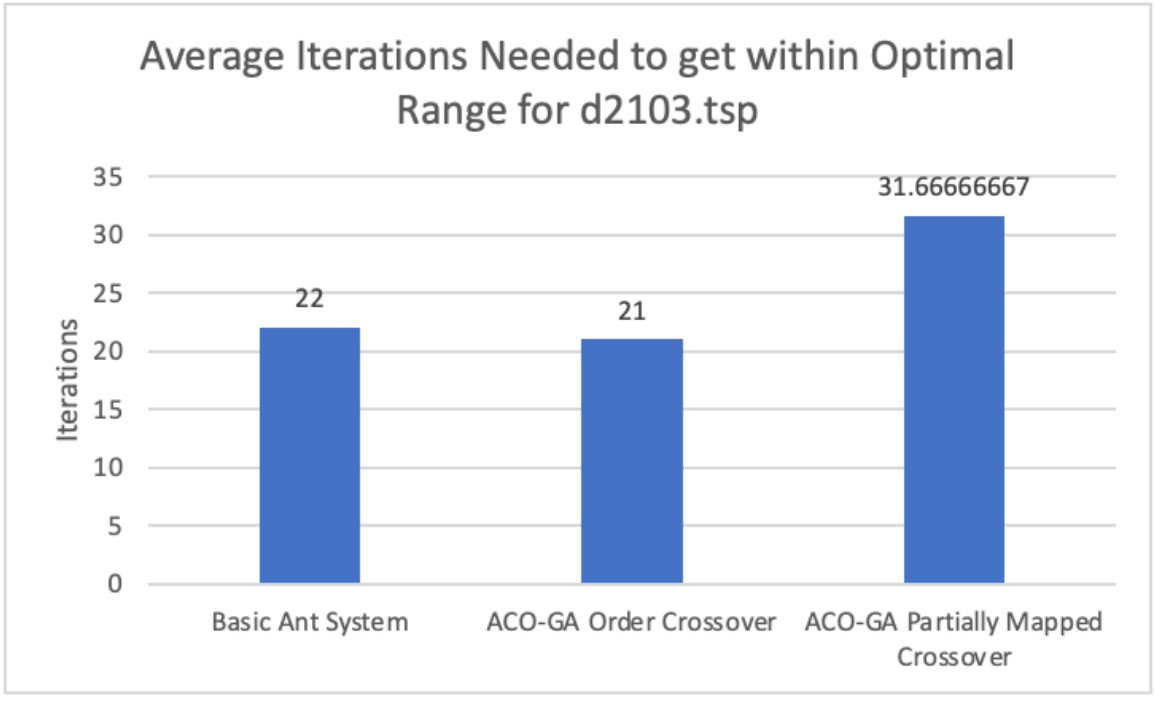

Figure 14: The number of iterations to find the specified optimal range (1.17) for file d2103.tsp. Capped at 100 iterations.

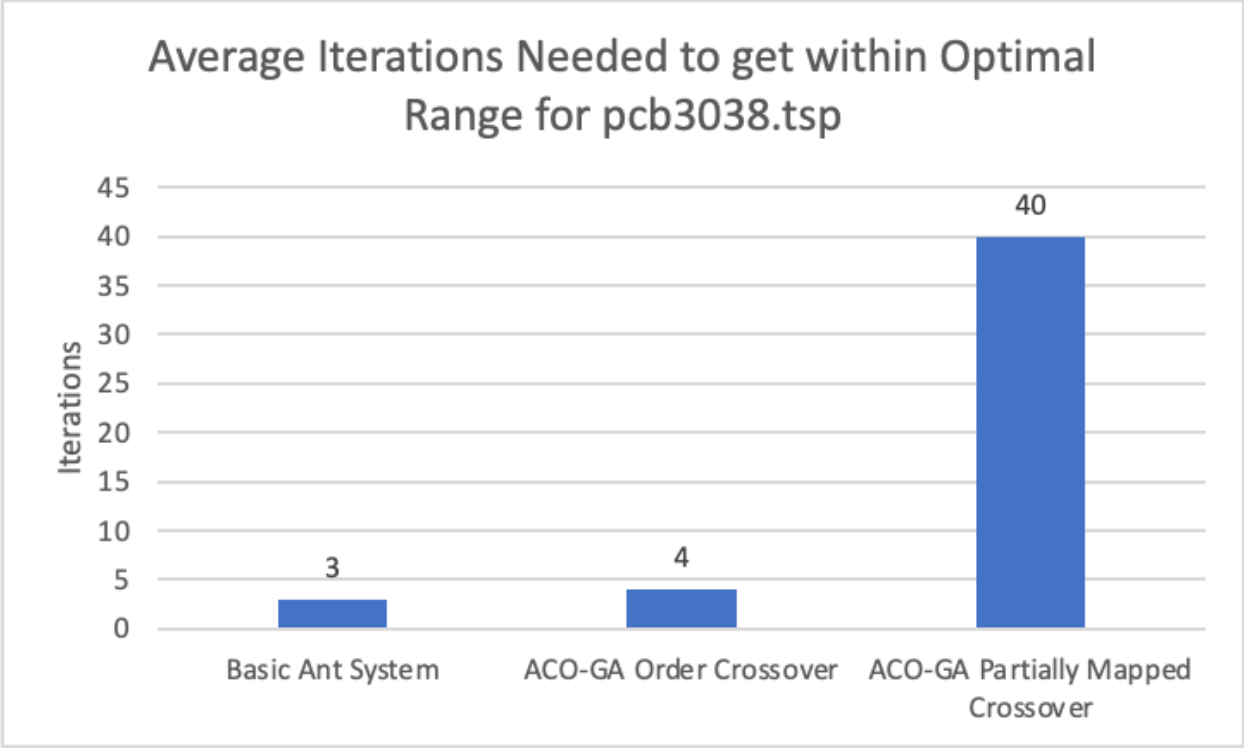

Figure 15: The number of iterations to find the specified optimal range (1.51) for file pcb3038.tsp. Capped at 100 iterations.

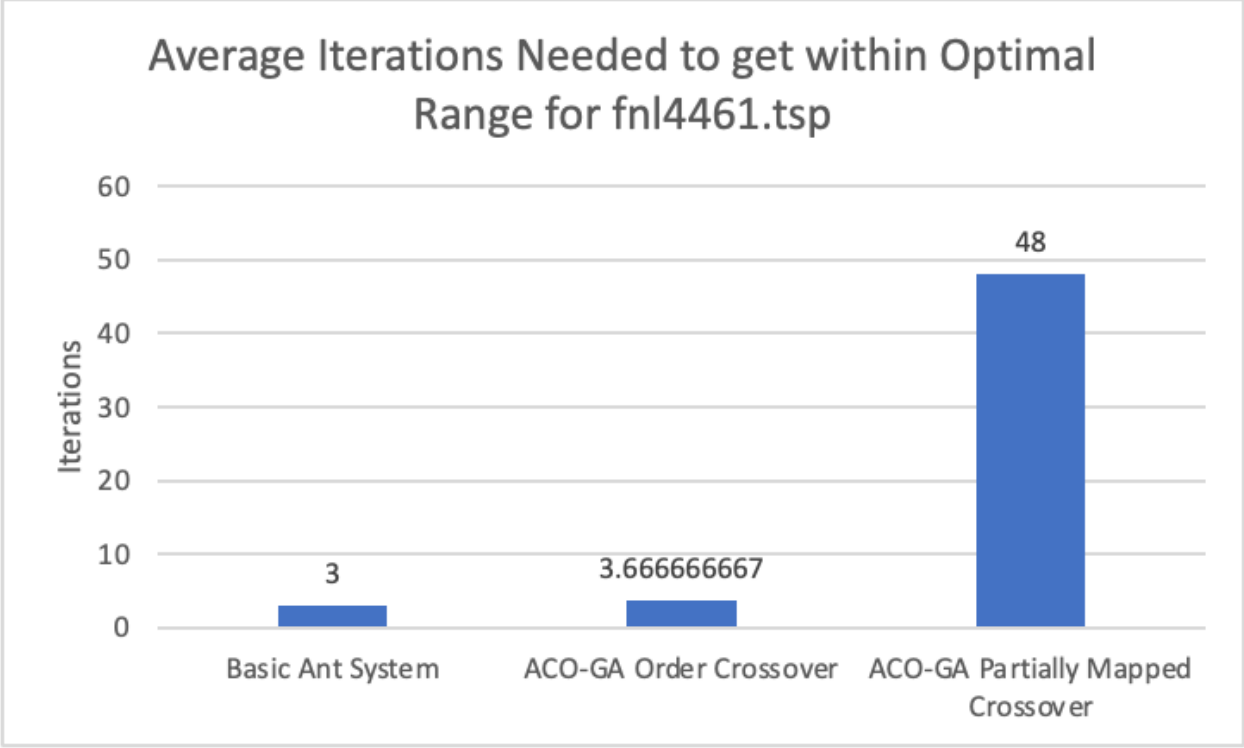

Figure 16: The number of iterations to find the specified optimal range (1.52) for file fnl4461. Capped at 100 iterations.

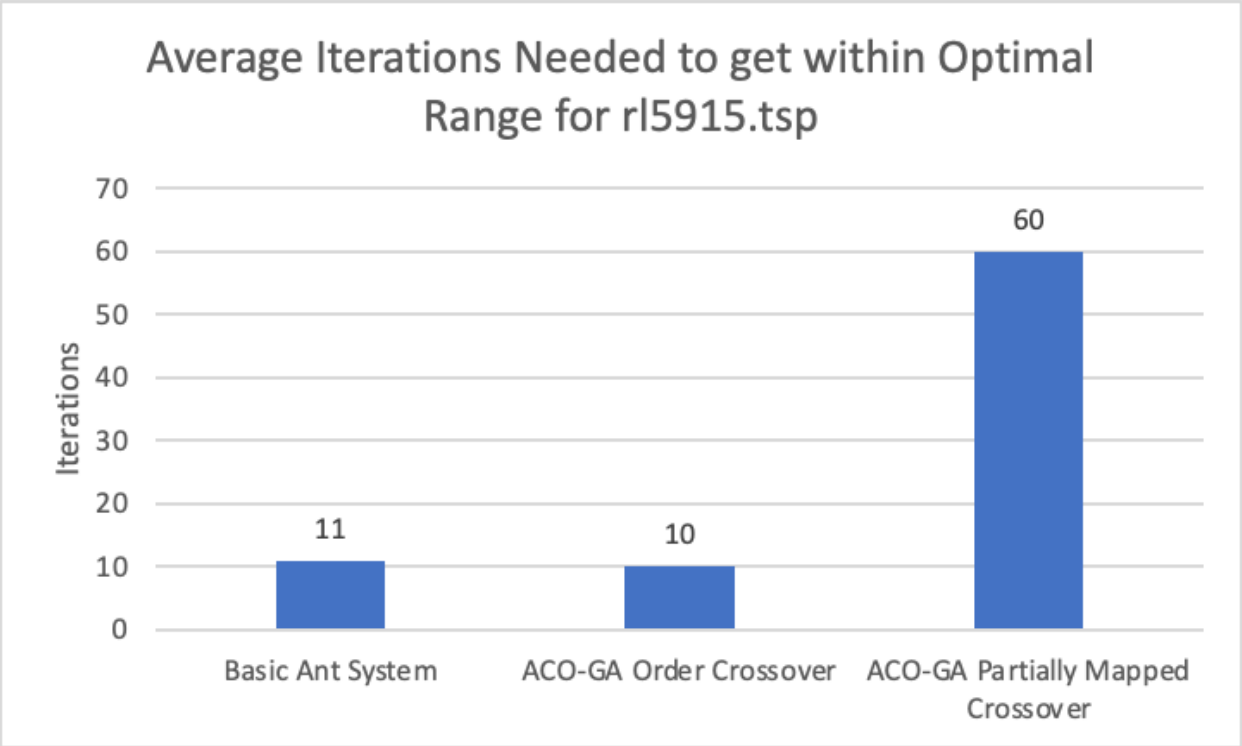

Figure 17: The number of iterations to find the specified optimal range (1.4) for file rl5915. Capped at 100 iterations.

Based on the optimal performance of each algorithm over 50 iterations, these results seem to make logical sense in relation. BAS and ACO-GA with OX reached an optimal range faster, and ACO-GA with PMX took significantly longer. The primary difference between the two results was that file size only seemed to play a significant role on ACO-GA with PMX, as it became markedly worse as file size increased. However, the optimal range changes across file sizes, so it is difficult to compare performance between file sizes, and better to compare performance between each algorithm for each file size. Regardless, we can conclude that BAS and ACO-GA with OX have a negligible difference between one another, and ACO-GA with PMX is still much worse. This seems to endorse the idea that OX is a superior crossover operator in comparison to PMX, and that speed differences should not be a determining factor when picking between BAS and ACO-GA with OX.

8 FURTHER WORK

If we were to continue this study further, one thing we would implement would be different selection methods. In this experiment, we used strictly a modified tournament selection.

In the future, we could implement ranked and truncated selection as well as other variations to see how the selection method impacts the performance of the algorithms. Note that we chose tournament selection in this case because it seemed like the best option when only using twenty ants, or candidate solutions, and only taking eight of the twenty ants.

Another aspect to add would be to bring in different ways to do the mutation process of the GA algorithm. For this study, we chose not to mutate the tours, but in the future, we could implement various ways to mutate the recombined tour other than just swapping neighboring cities. While we implemented and added this form of mutation, it consistently produced worse results, so if we could find a different way to implement this portion that bettered the results, that would be useful.

Finally, we would also take the time to run the algorithms on more TSP files that vary in size. This could be useful to extract more data and subtle differences as we saw a different algorithm perform better as the file size increased. Trying these algorithms on extremely large file sizes, in particular, would be a good indication of whether ACO-GA with OX would continue to increasingly outperform BAS, or if they maintain a tight range from one another.

9 CONCLUSION

We have presented three ACO algorithms for solving TSP: the Basic Ant System, the first hybrid ACO-GA algorithm that uses OX, and the second hybrid ACO-GA algorithm that uses PMX. With smaller file sizes, neither combination algorithm outperformed the original BAS, but ACO-GA with OX was quite close, and ACO-GA with PMX was significantly worse. However, as the file size increased, we saw the ACO-GA with OX begin to barely outperform the BAS. We believe this is perhaps due to the GA having a better effect on larger files given they contain more cities and more opportunities for variance. We also notice that the amount of time it takes to reach an optimal range was roughly the same between BAS and ACO-GA with OX, but was much worse for ACO-GA with PMX.

Overall, we can conclude that for larger files, we suggest using ACO-GA with OX, and for smaller files, we suggest using BAS. However, these differences are quite small, and both algorithms should perform well on all file sizes, both in optimality and efficiency. In addition, we suggest using OX rather than PMX when using an ACO-GA hybrid algorithm for TSP, as OX consistently outperforms PMX by a noticeably large margin.