Comparative Analysis of Ant Colony Optimization and Genetic Algorithm in Solving the Traveling Salesman Problem

6th June 2021

Hatem Mohi El Din
Faculty of Computing
Blekinge Institute of Technology
SE-371 70 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the bachelor's degree in software engineering. The thesis is equivalent to 10 weeks of full-time studies.

Contact Information:
Author:
Hatem Mohi El Din
E-mail: hatem.mohieldin@gmail.com

University Advisor:
Emil Folino
E-mail: efo@bth.se

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
ABSTRACT

Metaheuristics is a term for optimization procedures/algorithms that can be applied to a wide range of problems. These problems for which metaheuristics are used usually fall in the NP-hard category, meaning that they cannot be solved in polynomial time. This means that as the input dataset gets larger the time to solve increases exponentially. One such problem is the traveling salesman problem (TSP) which is and has been widely used as a benchmark problem to test optimization algorithms. This study focused on two such algorithms called ant colony optimization (ACO) and genetic algorithm (GA) respectively. Development of such optimization algorithms can have huge implications in several areas of business and industry. They can for example be used by delivery companies to optimize routing of delivery vehicles as well as in material science/industry where they can be used to calculate the most optimal mix of ingredients to produce materials with the desired characteristics. The approach taken in this study was to compare the performance of the two algorithms in three different programming languages (python, javascript and C#). Previous studies comparing the two algorithms have reported conflicting results where some studies found that ACO yielded better results but was slower than GA, while others found that GA yielded better results than ACO. Results of this study suggested that both ACO and GA could find the benchmark solution, but ACO did so much more consistently. Furthermore javascript was found to be the most efficient language with which to run the algorithms in the setup used in this study.

Keywords: Ant colony optimization, genetic algorithm, metaheuristics, javascript, python, C#
# Table of contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>3</td>
</tr>
<tr>
<td>Table of contents</td>
<td>4</td>
</tr>
<tr>
<td><strong>1 Background/Literature review</strong></td>
<td>5</td>
</tr>
<tr>
<td>1.1 Traveling salesman problem and optimization algorithms</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Global/Local optima</td>
<td>8</td>
</tr>
<tr>
<td>1.3 Previous studies</td>
<td>8</td>
</tr>
<tr>
<td><strong>2 Focus area</strong></td>
<td>9</td>
</tr>
<tr>
<td>2.1 RQ1.</td>
<td>9</td>
</tr>
<tr>
<td>2.2 RQ2.</td>
<td>10</td>
</tr>
<tr>
<td>2.3 RQ3.</td>
<td>10</td>
</tr>
<tr>
<td>2.4 RQ4.</td>
<td>10</td>
</tr>
<tr>
<td><strong>3 Materials and Methods</strong></td>
<td>11</td>
</tr>
<tr>
<td>3.1 Traveling salesman problem</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Experimental setup</td>
<td>11</td>
</tr>
<tr>
<td>3.3 Iteration constraint</td>
<td>12</td>
</tr>
<tr>
<td>3.4 Time constraint</td>
<td>12</td>
</tr>
<tr>
<td>3.5 Ant colony optimization</td>
<td>13</td>
</tr>
<tr>
<td>3.5.1 Tweaking pheromone weight</td>
<td>15</td>
</tr>
<tr>
<td>3.6 Genetic Algorithm</td>
<td>15</td>
</tr>
<tr>
<td>3.6.1 Crossover</td>
<td>16</td>
</tr>
<tr>
<td>3.6.2 Mutation</td>
<td>16</td>
</tr>
<tr>
<td>3.7 Brute force solution</td>
<td>16</td>
</tr>
<tr>
<td>3.8 Containerizing the experiments</td>
<td>17</td>
</tr>
<tr>
<td><strong>4 Results</strong></td>
<td>18</td>
</tr>
<tr>
<td>4.1 Ant colony optimization</td>
<td>18</td>
</tr>
<tr>
<td>4.1.1 ACO - Iteration</td>
<td>18</td>
</tr>
<tr>
<td>4.1.2 ACO - Time</td>
<td>21</td>
</tr>
<tr>
<td>4.2 Genetic Algorithm</td>
<td>22</td>
</tr>
<tr>
<td>4.2.1 GA - Iteration</td>
<td>22</td>
</tr>
<tr>
<td>4.2.2 GA - Time</td>
<td>23</td>
</tr>
<tr>
<td>4.3 Brute force vs optimization algorithms</td>
<td>24</td>
</tr>
<tr>
<td><strong>5 Discussion</strong></td>
<td>25</td>
</tr>
<tr>
<td><strong>6 Conclusion</strong></td>
<td>29</td>
</tr>
<tr>
<td><strong>7 Validity Threats</strong></td>
<td>30</td>
</tr>
</tbody>
</table>
1 Background/Literature review

1.1 Traveling salesman problem and optimization algorithms

In the world of combinatorial optimization, the traveling salesman problem has become the most famous prototype problem and can be considered a benchmark for testing the performance of various optimization algorithms[1],[2]. Briefly, the traveling salesman problem (TSP) is presented as a salesman who has to visit customers in a certain number of cities. The goal is to find the shortest possible route the salesman can take in order to visit each city exactly once and finally return to the starting city. At first glance this sounds easy enough, but as the number of locations to visit grows, an exact solution to the problem becomes next to impossible in a feasible amount of time[2]. Due to this, the TSP is said to be an NP-hard problem[3]. A detailed look into NP problems and all that NP entails is beyond the scope of this thesis, but grossly simplified one can say that, as it pertains to the TSP, complexity and time to solve will exponentially increase for each added location. To further illustrate how quickly the complexity increases for every city added, 11 cities gives us ~40 million possible orders in which to visit each city, 12 cities gives us ~ 500 million possible orders and 13 cities gives us over 6 billions possible orders. We do not have to add too many cities for the time it would take to check all possibilities to be in the billions of years. Thus scientists have come up with numerous optimization techniques and algorithms with which an optimal solution to these sorts of problems can be found within a set time limit.

There are many different kinds of optimization techniques that are based on or inspired by processes found in for example science (physics, chemistry, biology), social and sports based behaviour and even the arts where some algorithms are inspired by how music is composed. Depending on the problem being studied some techniques may be
more suited than others. In some cases it may even be beneficial to combine more than one algorithm to reach a better solution than any one individual algorithm could achieve on its own. One algorithm that is based on a prohibitive search is called tabu search. This algorithm stores already explored solutions of a problem in “memory” so that further iterations exclude these which promotes diversification of the search space. This concept is used in other algorithms such as the ant colony optimization (discussed further below) where ants need to keep track of regions they have already explored so as not to backtrack or get stuck in a loop.

Two optimization techniques that have been applied to solving the TSP (and are the subject of this thesis) are ant colony optimization (ACO) and genetic algorithm (GA). The choice of GA as one of the algorithms is based on the fact the author is a molecular geneticist and is familiar with the concepts which GA aims to mathematically mimic. The choice of ACO was based on a personal fascination of the author by swarm intelligence systems. Another swarm intelligence technique that was considered was particle swarm optimization [4] which is based on how animals such as flock of birds or a school of fish coordinate. After researching the implementation of ACO and PSO it was evident that ACO was more suited for use with the TSP.

The ACO falls under the biological based metaheuristic algorithms, and as stated, is further classified as a swarm intelligence algorithm [5]. It is influenced, as the name suggests, by the behaviour of real ants. In the simplest of terms, ants, when looking for example for a source of food to bring home to the colony, are able to determine the best route to most efficiently bring the food from its source to the colony. They do this by leaving pheromone trails which their fellow ants can detect and follow. Fig 1 (taken from [6]) illustrates this concept of ants converging on the shortest route between two points. In the beginning ants will randomly walk to the food source and back to the nest, and as the pheromone accumulates, it will do so more strongly on the shortest path which in turn guides future ants making the same trip.
The ACO aims to mathematically mimic this behaviour by applying probabilistic bias to certain solutions according to a set of criteria. Considering the traveling salesman problem, the first criterion for the ACO is the length of the path travelled by the ant(s). The longer the path the less “attractive” it will be for the ants to travel. The length is at an inverse relationship with the second criterion which is the pheromone level. Ants will deposit more pheromone on shorter paths compared to longer ones. The ACO uses these two criteria so that each ant determines the best next destination for it from any given city it is currently in. As such the ACO can be considered a step by step algorithm, as each step of the journey is evaluated before the ant makes a decision. The weighting of pheromone and path length can be tweaked in search for better performance which is further discussed in the methods section. As mentioned above, tabu search or part of the tabu search concept is applied in ACO. Again, considering the ACO application for the TSP, each ant needs to keep track of which cities they have already visited so that they do not visit any city more than once.

The genetic algorithm (GA) is the second algorithm studied in this thesis and is inspired by natural selection or survival of the fittest as it is also referred to[7][8]. It is also a biologically based algorithm and if further classified into the evolutionary algorithms category[9]. In this algorithm populations are made up of random solutions to a problem. With the TSP in mind, every such solution will contain the complete order in which the cities should be visited. The GA will evaluate this order and give that particular solution
a fitness score. The next iteration will choose a predefined set of these solutions (usually the top nth percentile) and randomly swap the values of selected parameters within this set in a process called crossover. The order of cities within each solution can also be changed in a process called mutation. The population undergoes crossover and mutation every iteration and if any of the “children” solutions turn out to be a better fit, this becomes the new best fit solution. This continues until a stopping criterion has been met. One significant difference between GA and ACO is that GA evaluates a solution as a whole while ACO evaluates the solution while “traveling the road”.

1.2 Global/Local optima

One of the challenges of optimization algorithms is falling into so-called local optimum scenarios[10]. Optimization algorithms typically work by narrowing down the search space at every iteration to ultimately try to reach the best most optimal solution to the given problem. As such, there is an inherent risk that a search algorithm will “get stuck” in a search space that contains a locally optimal solution while a better global solution exists. This is illustrated by figure 2, where a solution has found several local optimum points that are not the global optimum. Therefore if an optimization algorithm is to be able to generate good results, it needs to plan for such scenarios and be able to “break loose” from local optima.

![Figure 2 - global vs local optimum.](image)

Figure 2 - global vs local optimum. The figure is a visual representation of local and global optima. While there can be several local optima for any given problem, there is always only one global optimum. By Christoph Roser at AllAboutLean.com under the free CC-BY-SA 4.0 license [11].

1.3 Previous studies

As mentioned, many studies have been done using the traveling salesman problem[12],[13],[14],[15]. It has been used in assessing the performance of both algorithms studied in this thesis. A study from 2015 by Mukhairez and Maghari compared performance and efficiency of 3 optimization techniques in solving the TSP [16]. Their TSP involved 30 cities giving a total possible number of combinations of ~2.7 x 10^{32} which is an
enormous number. The techniques they compared were Simulated Annealing (SA), a
technique that models the annealing process of materials from the field of metallurgy,
Genetic Algorithm and Ant Colony Optimization. They observed both the execution
times of each algorithm as well as the optimal result achieved by the respective
algorithms. They reported that in terms of best result (shortest path) ACO was the clear
winner but also had the slowest execution time. SA had the fastest execution time and
had comparable results to GA, while GA came in second place for both execution time
and result. The programming language in which the algorithms were written in this study
was JAVA and all tests were done using the same platform. Another study by Haroun et
al. which compared ACO and GA had similar results, where they found that GA was
indeed faster and more “light weight” than ACO, but ACO consistently found shorter
distances especially with larger input sizes (number of cities)[16], [17].

However not all studies reported have favoured ACO over GA in terms of best result. A
study by Alhanjouri and Alfarra, which used a dataset of 25 city locations, showed that
GA yielded better results than ACO [18]. This suggests that, at least under certain
circumstances, the GA can be a contender to ACO for finding the shortest distance for a
TSP.

2 Focus area

This thesis focused on implementing ant colony optimization and genetic algorithms to
solve the traveling salesman problem given a set of constraints. The algorithms were
written in javascript, python and C# programming languages. The thought behind the
choice of languages was to include a client side language (Javascript), an interpreted
language (Python) and a compiled language (C#), and those three languages fit that
criterion and were the ones that the author is familiar with. Both intra and inter language
comparisons were carried out. Intra language comparison focused on comparing both
algorithm within each language, while inter language comparison focused on comparing
each of the algorithms in the three different languages.

2.1 RQ1.
How does ACO compare to genetic algorithm within each respective language in
solving the TSP within a given iteration constraint? The study aims to answer if one
algorithm is superior to the other. With this approach, the aim is to find out which of the
algorithms performs better (find the solution with the shortest total distance) when both
can run for a set number of iterations with no time limit. This is an important factor when
setting up optimization experiments as researchers or businesses need to be able to trust the reliability of their results.

2.2 RQ2.

How does ACO compare to genetic algorithm within each respective language in solving the TSP within a given time constraint? The study aims to answer if one algorithm is superior to the other. This approach imposes a different constraint on the algorithm compared to RQ1. Rather than a set number of iterations, the algorithms are left to run as many iterations as possible within a given time frame. This approach is important as it not only tests the performance of the algorithms in achieving good results, it also evaluates their time efficiency. While in research one might afford to run simulations that run for hours or days, in most businesses time is an extremely crucial factor. Thus the faster an algorithm can achieve its end goal the better.

2.3 RQ3.

Does performance of the algorithms vary significantly depending on which of the three languages they are running in? The aim with RQ3 is to shed light on potential performance/efficiency differences between the three languages chosen for this study (python, C# and javascript). Answering this may give a basis for future studies where for example one language is favoured due to its performance/efficiency.

2.4 RQ4.

How do the optimization algorithms compare to a brute force solution given a number of cities that can feasibly be solved by brute force? A brute force method refers to a method that analyses every possible route of a TSP. This means that for a TSP with 11 cities, the brute force method will calculate the total distance of all 40 million (!11) possible solutions. The aim of this question is to evaluate the results of the optimization algorithms to the actual best possible solution found by a brute force algorithm. The results of this question will be important coupled with the results of the other 3 research questions posed in this study.
3 Materials and Methods

3.1 Traveling salesman problem

As the basis of the TSP for this study, google's city matrix, which they use to showcase their OR-tools, was used [19]. The matrix includes 13 cities with which the number of possible combinations are over 6 billion.

The basic concept of the TSP is simple. A salesman has to travel to a certain number of cities and wants to know the most efficient route that can be taken to visit each city exactly once and then return to the starting city. As mentioned in the introduction, the time to find the best solution to this problem exponentially increases with each city added until a point where it is no longer possible to solve within a feasible amount of time.

3.2 Experimental setup

The algorithms were written in three different programming languages, Javascript, Python and C#. There is an abundance of information on these two algorithms in the
literature, and the ones written for this study are an adaptation of them based on pseudocode found in the literature (see pseudocodes for the respective algorithms in section 3.5 and 3.6). All experiments were run on a computer with a Core i5-6600K CPU and 16 gb ram. Experiments were containerized using docker (see 3.8 for environments used).

### 3.3 Iteration constraint

For these experiments the algorithms were run for a set number of iterations. For ACO there is no optimal number of iterations that can be claimed by any one study, but previous studies tend to use between 50 and 300 iterations. A study by Nwamae et al. in 2018[20] found that ACO results (with TSP) significantly improve up to 50 after. The results did not improve as much between iterations 100 and 200. Based on this, a 100 iterations limit was chosen for this study for ACO. Similarly there is no best/optimal number of iterations for GA, and the choice was made based on similar previous studies. One such study by Alhanjouri et al. 2011[17], reported their optimal solution with just over 100 iterations for GA.

Both shortest distance and execution time were measured. Each experiment was run 100 times and the average was used in result analysis. A possible approach to collect the data of the 100 repeats was to implement a write function in the algorithms to automatically print the results of each run to a file. However this was decided against as it would potentially impact the performance of the algorithms. Instead, to keep performance impact as low as possible, the results were output to a console window from which they were manually copied into a spreadsheet.

### 3.4 Time constraint

For these experiments the algorithms were run for a certain number of seconds. The exact duration depended on the execution time of the algorithms in the different languages (see table 1 below). Execution time was measured and tracked in code using the respective time functions for each language. Each experiment was run 100 times and the average was used in result analysis. The same rationale in collecting results was used for this section as for section 3.3.

<table>
<thead>
<tr>
<th>Language</th>
<th>Algorithm</th>
<th>Duration (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>ACO</td>
<td>850, 1700, 2550</td>
</tr>
<tr>
<td>C#</td>
<td>ACO</td>
<td>400, 800, 1200, 3200, 4600, 6400</td>
</tr>
<tr>
<td>Language</td>
<td>Algorithm</td>
<td>Pheromone Levels</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>------------------</td>
</tr>
<tr>
<td>Javascript</td>
<td>ACO</td>
<td>200, 400, 600</td>
</tr>
<tr>
<td>Python</td>
<td>GA</td>
<td>1000, 2000</td>
</tr>
<tr>
<td>C#</td>
<td>GA</td>
<td>1000, 2000</td>
</tr>
<tr>
<td>Javascript</td>
<td>GA</td>
<td>1000, 2000</td>
</tr>
</tbody>
</table>

### 3.5 Ant colony optimization

The ACO algorithm is based around two core metrics, cost (in this case the length of each edge) and pheromone level/concentration on each edge. To model this, weighted graphs were used [21], [22]. Each metric was represented by its own weighted graph. For the purposes of using this in code, the adjacency matrix for each of the graphs were used. Due to this, the google matrix on which the TSP was based was ideal as it is already in the format of an adjacency matrix representing cost. An initial adjacency matrix was generated for pheromone levels where each edge was given a starting pheromone level of 1 and was updated on every iteration. The algorithm flow was based on the below pseudocode.

Pseudocode was adapted from several sources to fit this study (example pseudocode in [23], [24], [25]):

**Start**

Set starting pheromone on every edge

**While** \( i < \) maxIterations **do**

**For** each artificial ant **do**

**Until** ant has visited all cities **do**

Select next node (city) based on given probabilities

Update pheromone trail for solution (total path travelled)

Record length of solution

**End for**

**End while**

**Return** best solution
The specifics of the implementation written for this study was largely based on the lecture by Seyedali Mirjalili [21] who is the director of the Centre for Artificial Intelligence Research and Optimization at Torrens University Australia.

The cost matrix represented the distances between each city in miles and the pheromone levels, in essence, represented the probabilistic bias of each edge. Experiments were run with 200 ants under two different constraints; time and iterations. For time constraints, the loop was run (see psuedocode) until the program reached a given time limit, while for the iteration version, the program was run for a set number of iterations. The time limits used for each language can be seen in table 1.

At the beginning of a loop a virtual ant (henceforth just referred to as an ant) started at a given starting city and considered every other city which it had not previously visited (the ant kept track of visited cities using a tabu list discussed in the introduction). The decision of which city would be the next destination was probabilistic and made based on two factors, the quality of the path to that city, and the amount of pheromone deposited on that edge. The quality of any given path is represented by \(1/L_{i,j}\) where \(L_{i,j}\) is the length of the path being considered (the shorter the path the higher the quality)[21]. Pheromone was deposited on every edge on which an ant traveled. The amount of pheromone \(\Delta \tau_{i,j}\) deposited is based on the length of the total path traveled from the starting city to the last city and is described by the following formula:

\[
\Delta \tau_{i,j} = \frac{1}{\text{length of total path}} \quad \text{(where delta tau is the change in pheromone for the edge \(i,j\))}[21].
\]

If multiple ants travel on the same edge then the total pheromone level for that edge would be the sum of all \(\Delta \tau_{i,j}\) and can be described with the following formula:

\[
\tau_{i,j} = \sum_{k=1}^{m} \Delta \tau_{i,j} \quad \text{(where k is the current ant, and m is the total number of ants that traveled edge \(i,j\)).}
\]

Initially all pheromone levels were equal on all edges thus the first ant made the decision purely based on the length of the path. The formula for choosing next destination is described by the following formula:

\[
P_{i,j} = \frac{\tau_{i,j}^{a} \eta_{i,j}^{b}}{\sum(\tau_{i,j}^{a} \eta_{i,j}^{b})}
\]
where $\eta_{ij} = 1/L_{ij}$, $\alpha$ and $\beta$ are parameters (with a value between 0 and 1) that can be used to tweak the impact of either pheromone level or length of the path in the decision making of each ant. In this study $\alpha$ and $\beta$ were both 1. Described in words, this formula calculates the product of the pheromone level and the inverse path length of a given edge, and divides that by the sum of all those products for every possible edge the ant can travel from its current location. With this formula a probability table was generated with the probability for each possible destination. The choice of destination was then made with a randomly generated floating point number.

3.5.1 Tweaking pheromone weight
To assess the importance of pheromone weight on performance of the ACO, the experiment was carried out with normal pheromone level, 10X the applied pheromone levels and 100X the applied pheromone levels. This meant that for the 10X and 100X the applied pheromone $\tau_{ij}$ was multiplied 10-fold and 100-fold respectively.

3.6 Genetic Algorithm
In contrast to the ACO, the genetic algorithm does not consider cost or any sort of pheromone levels for each city it can visit. Rather, random orders of cities (called solutions) are evaluated for their fitness (fitness in this case being the length of the total path through all cities in the chosen order) and based on the fitness some of the solutions may be kept and “inherited” by the next generation of solutions that is generated. Due to this the GA may be considered more flexible in the way the input data is structured compared to the ACO. For the purposes of comparability however the same adjacency matrix for cost used with the ACO was also used with the GA.

A starting population of 200 random solutions was generated. Fitness (total distance) of each solution was calculated and an appropriate container was created where each solution was associated with its fitness score. The container was sorted based on fitness score with the solution with the best fitness (lowest total distance) being the top entry. The next generation was generated from the starting population using the following criteria.

1. The top 50 solutions were added directly to the next generation.
2. Crossover was applied to the top 150 solutions after which they were added to the next generation.
3. All 200 populations added to the next generation underwent mutation.
3.6.1 Crossover
Crossover is the GA’s way to search the local solution area for a more optimal solution than the current best. This is implemented by mimicking crossover in actual chromosomes. In this study a crossover rate of 0.3 was used. This meant that for a crossover between 2 solutions, each with 13 cities, 9 cities were kept in the original order from solution 1 and supplemented with the 4 remaining cities from the other solution.

3.6.2 Mutation
To try and overcome getting stuck in a local optima, GA implements mutation to widen the search space beyond the immediate search area. Therefore, after crossover, the population underwent mutation at a rate of 0.9. If a solution was chosen for mutation, 2 cities at random would swap positions within that solution.

Having undergone both crossover and mutation, this newly generated generation was then used as the starting population of the next iteration. The overall algorithm flow was based on the below pseudocode.

Pseudocode adapted from [26]:

Start
Generate starting population (city orders)
While i < maxIteration do
    Calculate fitness of each solution
    Store solution with best fitness (shortest distance)
    Generate new generation by crossover
    Introduce mutations at a predefined rate
End while
Return best fitness

3.7 Brute force solution
A brute force algorithm was implemented for each of the three languages. This algorithm calculated the total distance of each possible combination of city order with
the given input. For python the itertools library was used to generate all possible permutations.

3.8 Containerizing the experiments
Experiments were run using docker containers. The respective docker images used for each of the language environments are listed in table 2.

<table>
<thead>
<tr>
<th>Language</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>python:3.9.4</td>
</tr>
<tr>
<td>C#</td>
<td>mcr.microsoft.com/dotnet/sdk:5.0</td>
</tr>
<tr>
<td>Javascript</td>
<td>node:14</td>
</tr>
</tbody>
</table>
4 Results

The traveling salesman problem based on google’s city matrix was solved using ACO and GA. Data was collected by experiments run with code written by the author. The two algorithms were studied and compared in their efficiency to find an optimal solution to the TSP. Google advertises the result of solving this problem with their OR-tools to be 7293 miles with the city order (order of their indices) [0, 7, 2, 3, 4, 12, 6, 8, 1, 11, 10, 5, 9, 0]. This solution was used as the benchmark solution to which the ACO and GA solutions were compared.

4.1 Ant colony optimization

4.1.1 ACO - Iteration

The first experimental setup evaluated ACO performance during a set number of iterations in all three languages. The best performance to time ratio was found to be 100 iterations each with 200 ants. To be able to distinguish efficiency differences between the languages, execution times of the program were measured using the respective language’s inbuilt time functions. Results showed that python had an average execution time of 3.4 seconds while C# had an average execution time of 1.6 seconds, and javascript had an execution time of 0.8 seconds making it the fastest of the three languages to run the ACO program.

<table>
<thead>
<tr>
<th>Language</th>
<th>Average execution time (s)</th>
<th>Best solution found</th>
<th>Rate with normal pheromone</th>
<th>Rate with 10x pheromone</th>
<th>Rate with 100x pheromone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>3.4</td>
<td>Yes</td>
<td>87%</td>
<td>98%</td>
<td>100%</td>
</tr>
<tr>
<td>C#</td>
<td>1.6</td>
<td>Yes</td>
<td>5%</td>
<td>38%</td>
<td>98%</td>
</tr>
<tr>
<td>Javascript</td>
<td>0.8</td>
<td>Yes</td>
<td>90%</td>
<td>99%</td>
<td>100%</td>
</tr>
</tbody>
</table>

With C#, even though results for the ACO could be considered satisfactory and, as mentioned, could reach the google benchmark minimum distance of 7293, it could not do so consistently, and indeed only rarely reached this minimum distance (average 5%). To try to reduce the variance of the results, the pheromone bias was tweaked to apply 10X and 100X the pheromone respectively as outlined in section 3.5.1. An observable difference could be seen when the pheromone applied was increased 10-fold where the
7293 benchmark was reached at a much more consistent rate (38%). Increasing the applied pheromone 100-fold increased the rate of finding the best solution to 98%. With javascript the best solution was found at a rate of 90% even with normal levels of applied pheromone. This increased to 99% at 10x applied pheromone and 100% at 100x applied pheromone. The same experiments with python achieved the best solution at a rate of 87%, 98% and 100% for normal pheromone, 10x, and 100x applied pheromone respectively (see table 1).

Another aspect that was investigated was how fast the algorithm converged on the minimum solution (i.e how many iterations did it take) be it the local or global minimum. There was no clear cutoff point for any of the pheromone levels where or around where convergence on the minimum always happened. There was a large variance from repeat to repeat as to at which iteration this occurred. However a correlation between pheromone level and average speed of conversion could be seen, where higher pheromone levels resulted in a quicker conversion than lower pheromone levels. This was true for all three languages. An example graph for the results of one repeat in each language are illustrated below (figures 3-5).

Figure 3 - Convergence of ACO on optimal solution in C#. The x-axis shows the number of iterations. The y-axis shows the value of the current best solution in miles. Blue represents normal pheromone, red represents 10x pheromone and yellow represents 100x pheromone.
Figure 4 - Convergence of ACO on optimal solution in Javascript. The x-axis shows the number of iterations. The y-axis shows the value of the current best solution in miles. Blue represents normal pheromone, red represents 10x pheromone and yellow represents 100x pheromone.

Figure 5 - Convergence of ACO on optimal solution in Python. The x-axis shows the number of iterations. The y-axis shows the value of the current best solution in miles. Blue represents normal pheromone, red represents 10x pheromone and yellow represents 100x pheromone.
4.1.2 ACO - Time

The second experimental setup evaluated ACO performance during a set time limit. This was carried out in python, javascript and C#. Initially, before any experimental data from the iteration constraint experiment had been obtained, the starting duration of each experiment was 5s, 10s and 20s respectively. However, since the iteration experiments showed that the execution time of the slowest language did not even reach 5 seconds, an adjustment was made. Instead of having set time limits for all three languages, each language got its specific time limits based on the execution time of 1 iteration. The time limits used were ¼ and ½ of the execution times for the respective languages and were carried out with 100x applied pheromone. Experiments were repeated 100 times to evaluate reproducibility and variance of results. Results showed that python performed poorly at both ½ and ¼ execution time, only succeeding to find the best solution in 31% and 13% of the repeats respectively. However, for javascript and C#, ACO found the best solution already at ¼ of the execution time with a high rate of success (over 70%) which is consistent with the results of the convergence experiment. This prompted another timed test where the performance of the algorithm in the respective language was tested with under a 100ms duration. The results showed (see table 3) that javascript, the language with the lowest execution time by a large margin, had the best performance with a 100ms time limit followed by C# and lastly by python, which is consistent with both the results from the execution time experiment as well as the convergence experiment.

<table>
<thead>
<tr>
<th>Language</th>
<th>100ms</th>
<th>200ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>8%</td>
<td>18%</td>
</tr>
<tr>
<td>C#</td>
<td>33%</td>
<td>46%</td>
</tr>
<tr>
<td>Javascript</td>
<td>40%</td>
<td>71%</td>
</tr>
</tbody>
</table>

Furthermore, for C# time limits corresponding to 2x, 3x and 4x the execution time with normal pheromone was tested to see if this gave results comparable to those of python and javascript at normal pheromone levels. While the results significantly improved (14%, 24% and 47% for the respective duration, compared to 5% with 100 iterations), they did not come close to those of python and javascript which had 99% and 90% respectively (see table 1).
4.2 Genetic Algorithm

4.2.1 GA - Iteration

As for the ACO algorithm, experiments on the same dataset were carried out in all 3 languages using GA under both time and iterations constraints. The iteration constraint approach used an iteration limit of 100 iterations with a population of 200 solutions. Several different combinations of crossover rate and mutation rate were evaluated (based on a similar study by Hassanat et al. 2019 [27]). For all three languages the best result was achieved with a crossover rate of 0.6 (keeping the first 5 cities from first solution) and a mutation rate of 0.9 (see figure 6, this particular combination is denoted as c5m90). For all combinations c stands for how many cities were used in cross over and m stands for mutation rate in percent). The overall patterns of results were very similar in all three languages, but the actual results differed between the three languages. For example the top performing combination of c5m90 in C# found the best solution at an average rate of 44%, where python found the best solution at a rate of 51% and javascript at a rate of 57%. At low crossover and low mutation rate (c10m10) neither python nor javascript could find the best solution. C# did find the best solution with this crossover/mutation combination albeit at a very low rate (4%). With the combination c5_nomut (crossover rate 0.6 and no mutations) none of the languages could find the best solution. Furthermore, this combination was also the one with which the worst results were obtained for any given repeat (result not shown). With the combination of no crossover but a 0.5 mutation rate (nocross_m50) all languages could find the best solution but at very low rates.
4.2.2 GA - Time

Similar to the ACO setup, the initial thought was to run the algorithm for 5, 10 and 20 seconds and evaluate results for each time. However, as for the ACO, execution times for GA were much lower than this. In fact the execution times for GA were below 0.2 seconds for all three languages. Therefore two of the combinations (c5m90 and c5m50) were chosen and run for 1 second in each language. This translated into just over 500 iterations of the algorithm (rather than 100 as was the limit for the iteration constraint experiment). The idea was to evaluate if running these combinations for longer would improve the rate at which GA could find the best solution in the respective languages. Python showed a marked improvement in results with this approach where both c5m90 and c5m50 jumped to 60%. The same pattern was seen for javascript where c5m90 and c5m50 jumped to 68% and 65% respectively. C# on the other hand did not show any improvement with this approach and the obtained results were largely similar to those obtained with a limit of 100 iterations (see table 4).
Table 4 - Result for running GA algorithm for 1 second. The results represent the percentage of runs where the algorithm found the best solution.

<table>
<thead>
<tr>
<th></th>
<th>c5m90 iteration</th>
<th>c5m90 time</th>
<th>c5m50 iteration</th>
<th>c5m50 time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>51%</td>
<td>60%</td>
<td>49%</td>
<td>60%</td>
</tr>
<tr>
<td>C#</td>
<td>44%</td>
<td>40%</td>
<td>22%</td>
<td>19%</td>
</tr>
<tr>
<td>Javascript</td>
<td>57%</td>
<td>68%</td>
<td>49%</td>
<td>65%</td>
</tr>
</tbody>
</table>

4.3 Brute force vs optimization algorithms

RQ4 proposed the question of whether the ACO and GA algorithms implemented in this study could achieve the actual best solution found by a brute force approach. For this purpose a search algorithm exploring every possible solution was implemented for python, javascript and C#. With this algorithm, the largest number of cities that could be processed was 11 for C# and python. At 12 cities the program ran out of memory and could not fully execute. For javascript, memory ran out already with 11 cities even with specifically allocating more memory to the program. Therefore it was decided to exclude javascript from this experiment. Thus, the first 11 cities of the google city matrix were used in this experiment in python and C#. In python, the brute force technique took on average 122 seconds to run and found the best solution to be 7168 miles. Using ACO, the solution found by the brute force algorithm was achieved consistently in under 2 seconds, while it took on average just over 2 seconds for GA. For C# the brute force algorithm took 242 seconds to complete with 11 cities while ACO and GA could find the best solution consistently in 1.3 and 1.1 seconds respectively.
5 Discussion

Experiments were carried out with the purpose of answering the questions posed under the section “Focus Area”. Google’s city matrix([28]) was used as the basis for the traveling salesman problem and the algorithms used in this study was written by the author. The benchmark solution which the algorithms aimed to match had a total path length of 7293 miles. Results of the experiments aimed at answering RQ1, which aimed to compare ACO and GA using iteration limits, showed that both ACO and GA were able to find this solution but with varying success rates. The top performance of ACO using an iteration limit of 100 iterations was achieved by both python and javascript where the best solution could be found at a rate of 100% by tweaking the amount of applied pheromone. The fastest of these was the algorithm run in javascript which had an execution time of 0.8 seconds. In contrast, the best performance for GA using an iteration limit of 100 iterations, found the solution at a rate of 57% with an execution time of under 0.2 seconds. While all three languages had very similar execution times with the GA algorithm, javascript was the one which gave the best result (the aforementioned 57%). This shows that, at least with the setup used in this study, ACO had a significantly better rate of finding the most optimal solution compared to GA. However this came at a cost of a higher execution time which at its lowest (javascript) was 4x higher than that of GA and at its highest was 17x higher that its GA counterpart (python). While this did not have a huge impact in this study (a couple of seconds), it could be speculated that as the size of the TSP increases, this difference in execution time becomes more and more pronounced. Thus, in answer to RQ1, ACO was found to be superior to GA in terms of consistency, but GA was superior to ACO in terms of speed.

It is worth mentioning that while GA could not achieve the optimal solution at the same consistency of ACO, there were a lot of unexplored optimizations of the GA algorithm. Different combinations of crossover and mutations were tried, but the actual method of crossover and mutation was static. Some studies also use dynamic crossover and mutation rates based on which generation the algorithm is currently in. It is possible that tweaking the way in which crossover or mutation are implemented could yield better results. Other studies using GA for solving TSP have usually reported high crossover rates with low mutation rates to be optimal which is in disagreement to the data of this study, especially with regards to mutation rate. One aspect that could play a role in this is the size of the population, as a study by Hassanat et al. in 2019[27] showed that the larger the population the more reliant the GA is on crossover and mutation to reach peak performance. To note is that this study was done on a different problem (the onemax problem), but gives weight to the notion that different experimental setups could have significantly different requirements for crossover and mutation.
To answer RQ2, timed experiments were carried out with both ACO and GA. GAs duration were increased to 1 second (which was slightly higher than the lowest execution time for ACO held by javascript) for all languages to see if GA could compete with ACO with a running duration just over 5x the execution time. ACO performance was evaluated for all languages with 100ms and 200ms (duration comparable to GA execution times) durations and compared to GA results. Increasing the running duration of GA did indeed improve results, but they were still not as high as for ACO. GA was further tested by increasing running duration to 2 seconds (result not shown) but this did not yield any improvement to the results, suggesting that GA reaches a plateau at around 60-65% success rate for this setup. ACO results on the other hand were quite interesting. At 100ms (which is a duration lower than that of the GA execution time) results were lower than those of GA but not by a huge margin, at least for javascript and C#. Increasing the duration to 200ms (which is roughly as long as the GA execution time) the ACO actually outperformed GA by a significant margin in javascript, and achieved comparable results to GA in C#. For python however, the rate of success was significantly lower with 8% for 100ms and 18% for 200ms. These results are consistent with the respective execution times for ACO in the different languages as well as the speed of convergence in the different languages. These results further support the idea of ACO having an edge over GA in this study but, depending on the language used, is more time consuming. This is in agreement with the study by Mukhairez and Maghari in 2015 [16] where they reported that in their TSP setup, ACO gave much better results than GA, but GA was significantly faster than ACO. If only the first part of the timed experiment was considered, then the answer to RQ2 would be very clear, GA is far superior to ACO at very low time limits. However, results of the experiments that forced a very low time limit on ACO showed that ACO could indeed match GA or even perform better even at these limits, depending on the language in which the algorithms were run. This made answering RQ2 more complicated, and suggested that the ACO could potentially be optimized further to run for fewer iterations or with fewer ants. Interestingly this is an agreement with a study by Haroun et al. 2015 [17]where they found that ACO does indeed converge on the optimal solution quicker than GA while GA as an algorithm was much faster to run compared to the ACO.

Another question that this study aimed to answer (RQ3), was if there are any significant differences in performance depending on which language the respective algorithm is written in. For both algorithms the results suggested that C# had a worse performance than both javascript and python pertaining to the actual end results. For ACO this was most pronounced with normal and 10x the applied pheromone where results differed significantly with those of javascript and python. It was only at 100x the pheromone where C# achieved similar results. This significant difference between C# and the other
two languages could potentially be explained by the the author being less familiar with C# compared to Javascript and Python. Comparing execution times however, C# came in second place with an execution time of 1.6 seconds. Python had the longest execution time of 3.4 seconds on average, but was also tied at first place with javascript in terms of results at all 3 pheromone levels (normal, 10x and 100x). Javascript had the fastest execution time by a large margin at 0.8 seconds with results rivaling those of python, making javascript the most cost efficient language with which to run ACO in this study. Hence, for ACO the answer to RQ3 was quite clear; there were indeed significant differences in ACO performance based on the language.

The answer to RQ3 for GA on the other hand was not as clear as for ACO. The execution times did not differ significantly between the 3 languages being approximately 150-200 ms in all languages. Several different combinations of crossover and mutation rates were evaluated. Python and javascript had very similar results across the board of combinations with javascript having a slight edge over python. C# struggled to achieve as high a success rate as python or javascript, which was also the case for ACO. Within the respective languages, comparison of the two algorithms was consistent, where ACO had a better performance but worse execution time in all three languages. Therefore, with RQ3 in mind, javascript and python had very similar performance, with C# coming in last place. Unfortunately, very little could be found in the literature with which to compare the results for this particular section. Most published studies have used JAVA, C++ or MATLAB in similar experiments and where javascript has been used, it was used to create gui based programs which is not comparable to the solution in this study. Overall however, the rankings of execution time results are reasonable from a technical point of view, where python, which is an interpreted language, is considered a relatively slow compare to javascript and C#.

With regards to RQ4 the unfortunate scenario occurred where Javascript ran out of memory when trying to solve the TSP with 11 cities. Javascript was able to solve the TSP with brute force when using only 10 cities, but at this point, the brute force method in all languages solved the TSP in just a few seconds making comparison to the optimization algorithms moot. For Python and C# however, the brute force method managed to solve the TSP with 11 cities in 2 minutes and 4 minutes respectively. A previous study used JAVA to solve the TSP with a brute force method which also had a relatively long solution time where it solved for 11 cities in around the minute mark. Interestingly this previous study managed to run the brute force with 12 cities, but were force to increase the Java heap size to a very large size of 22 GB (the amount of memory allocated to stories object created by an application [29]).
In contrast to the brute force method, both optimization algorithms in this study could consistently find the best solution (found by the brute force method) in 2 seconds or less in both languages, with C# having a slight edge over Python. These results suggest that the optimization algorithms are indeed capable of finding the best solution to problems such as the TSP. However one should be careful not to take these results at face value as the brute force experiment is not infinitely scalable (due to the n! solution to a TSP with n cities) and as such cannot be tested with larger implementations of the TSP.
6 Conclusion

In this study questions were posed regarding the performance of two metaheuristic optimization algorithms, ant colony optimization (ACO) and genetic algorithm (GA) in solving the traveling salesman problem. Furthermore, assessment of the performance and efficiency of these algorithms were carried out and compared in three different languages, javascript, python and C#. This study showed that ACO, while much slower than GA, yields far better results when applied to the TSP when execution times were not considered. On the other hand, if execution times were considered, GA was, at first glance, superior to ACO due to its extremely fast execution time. However, while the GA was several times faster to execute in all 3 languages tested, if a time limit is applied to ACO (in essence stopping it before it has run through all iterations), it was able to achieve better results than GA within the same time frame in javascript and match GA in C#. Comparing performance of the 3 different languages showed a similar pattern in both GA and ACO where python and javascript were more consistent in achieving the benchmark results under most parameter combinations while C# struggled in all but the most optimized parameter conditions. Execution speed however was quite varying depending on the language and algorithm for ACO with javascript being fastest by far, with C# in second place and python last. For GA all three algorithms had very similar execution times. The last experiment, comparing both optimization techniques to a brute force method could not be answered satisfactorily as javascript was unable to run the brute force method with the given input of 11 cities. Therefore this particular aspect was only fully explored for python and C#. Results showed that both languages could consistently reach the best solution found by the brute force method in a fraction of the time it took the brute force method.

The results of this study paints an interesting picture where strengths and weaknesses were observed both for the algorithms and also for the three languages. It was quite clear that with the experimental setup used in this thesis javascript was the most efficient language. However the inability of javascript to execute the brute force method suggests that while javascript may be superior in this setup, it may struggle with larger datasets.
7 Validity Threats

Due to the author’s relatively limited experience, it may be that the fundamentals of the code for the algorithms could be significantly more optimized.

One of the more curious finding was that C# showed very poor performance in ACO at lower pheromone levels compared to Javascript and Python. This could potentially be due to the author’s relative inexperience in C# compared to the other two languages. Due to this, nuances of the C# languages that might affect the algorithms could have been missed.

The crossover and mutation combinations for GA analyses were not exhaustive and there could potentially be a better combination than the one found in this study.

In contrast to ACO, GA does not need to use weighted graphs, and therefore, in hindsight, it could have been a better approach to use a different format for the input data for GA. An example of another format would be an array of city coordinates from which the distances can be calculated when needed. This is true also for the brute force method, where such an input could potentially have meant that Javascript would be able to run the brute force with 11 cities as there would be much less data stored during execution.

The input data used throughout this thesis was the google city matrix which included 13 cities. Therefore the scalability of the algorithms implemented in this study cannot be guaranteed.
8 References


const performance = require('perf_hooks').performance;
const costMatrix = [
    [ 0, 2451, 713, 1018, 1631, 2408, 213, 2571, 875, 1420, 2145, 1972 ],
    [ 2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357 ],
    [ 713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453, 1260 ],
    [ 1018, 1524, 355, 0, 700, 862, 1395, 1123, 1584, 466, 1056, 1280, 987 ],
    [ 1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371 ],
    [ 1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 225, 887, 999 ],
    [ 2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1114, 701 ],
    [ 213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300, 2099 ],
    [ 2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653, 600 ],
    [ 875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272, 1162 ],
    [ 1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017, 1200 ],
    [ 2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0, 504 ],
    [ 1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504, 0 ]
];

let pheromoneMatrix = [
    [ 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
    [ 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1 ],
    [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0 ]
];
const nextDestination = (city, pheromone, visited) => {
```
let denominator = 0;
let probabilities = {};
let availableDestinations = city.filter(d => d != 0 && !visited.includes(city.indexOf(d)));

for (let i = 0; i < availableDestinations.length; i++) {
    let pheromoneLevel = pheromone[city.indexOf(availableDestinations[i])];
    let distance = 1.0 / availableDestinations[i];
    let product = pheromoneLevel * distance;
    denominator += product;
}

for (let i = 0; i < availableDestinations.length; i++) {
    let pheromoneLevel = pheromone[city.indexOf(availableDestinations[i])];
    let distance = 1.0 / availableDestinations[i];
    let probability = (pheromoneLevel * distance) / denominator;
    probabilities[city.indexOf(availableDestinations[i])] = probability;
}

const sortable = Object.entries(probabilities).sort(((a,b) => b-a))
let orderedKeys = sortable.map(keyValuePair => keyValuePair[0])
let orderedValues = sortable.map(keyValuePair => keyValuePair[1])

let updatedProbabilityValues = [];
sortable.forEach(value => {
    let total = orderedValues.reduce((a, c) => a + c);
    updatedProbabilityValues.push(total);
    orderedValues.shift();
})

let rouletteProbability = Math.random();

for (let i = 0; i < orderedKeys.length - 1; i++) {
    if (rouletteProbability <= updatedProbabilityValues[i] &&
        rouletteProbability >= updatedProbabilityValues[i + 1])
```
return orderedKeys[i];
}

return orderedKeys[updatedProbabilityValues.length - 1];

const updatePheromone = (routeLength, route) => {
  let pheromoneToApply = 1.0 / routeLength * 100;
  for (let i = 0; i < route.length - 1; i++)
  {
    let startingPheromonePosition = pheromoneMatrix[route[i]];
    startingPheromonePosition[route[i + 1]] += pheromoneToApply;
  }
}

const CostCalculator = (route) => {
  let routeLength = 0;
  for (let i = 0; i < route.length - 1; i++)
  {
    let startingCity = costMatrix[route[i]];
    routeLength += startingCity[route[i + 1]];
  }
  return routeLength;
}

const shortestDistance = () => {

  let order = [];
  let shortestRoute = 999999;

  //For times experiments uncomment below and tweak
  // var t0 = performance.now();
  // while (performance.now() - t0 < 400)

  for (let i = 0; i < 100; i++)
  {
    for (let j = 0; j < 200; j++) {
order = [];
let count = 0;
let currentCity = costMatrix[Math.floor(Math.random() * Math.floor(13))];
order.push(costMatrix.indexOf(currentCity));

while (count < costMatrix.length - 1) {
    let nextIndex = nextDestination(currentCity, pheromoneMatrix[constMatrix.indexOf(currentCity)], order)
    currentCity = costMatrix[nextIndex]
    order.push(costMatrix.indexOf(currentCity));
    count++;
}

order.push(costMatrix.indexOf(costMatrix[order[0]]));
let routeLength = CostCalculator(order);

if (routeLength < shortestRoute) {
    shortestRoute = routeLength;
}

updatePheromone(routeLength, order);

console.log(shortestRoute)
return shortestRoute

for (let i = 0; i<100; i++) {
    shortestDistance();
}
const performance = require('perf_hooks').performance;

const costMatrix = [
    [ 0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145, 1972 ],
    [ 2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357, 579 ],
    [ 713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453, 1260 ],
    [ 1018, 1524, 355, 0, 700, 862, 1395, 1123, 1584, 466, 1056, 1280, 987 ],
    [ 1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371 ],
    [ 1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 225, 887, 999 ],
    [ 2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1114, 701 ],
    [ 213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300, 2099 ],
    [ 2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653, 600 ],
    [ 875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272, 1162 ],
    [ 1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017, 1200 ],
    [ 2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0, 504 ],
    [ 1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504, 0 ],
];

const RandomOrder = () => {
    const order = [];

    while (order.length < costMatrix.length)
    {
        var rng = Math.floor(Math.random() * 13);
        if (!order.includes(costMatrix.indexOf(costMatrix[rng])))
        {
            order.push(rng);
        }
    }

    return order;
```javascript
const GenerateSolution = () => {
  var solution = RandomOrder();
  solution.push(solution[0]);
  return solution;
}

const GeneratePopulation = (size) => {
  const population = [];
  for (let i = 0; i < size; i++) {
    population.push(GenerateSolution());
  }
  return population;
}

const CrossOver = (first, second) => {
  const firstInterim = first.slice(0, 5);
  second.forEach(city => {
    if (!firstInterim.includes(city)) {
      firstInterim.push(city);
    }
  });
  firstInterim.push(firstInterim[0]);
  return firstInterim;
}

const Mutate = (popToMutate) => {
  popToMutate.forEach(city => {
    var rng = Math.floor(Math.random() * 1000);
    if (rng < 500) {
      let oldIndex = Math.floor(Math.random() * 12) + 1;
      let newIndex = Math.floor(Math.random() * 12) + 1;
      // Mutate code
    }
  });
```

const NextGeneration = (currentGeneration) => {
    let nextGen = [];
    for (let i = 0; i < 50; i++)
    {
        nextGen.push(currentGeneration[i]);
    }

    let keepForCrossOver = currentGeneration.slice(0,150);
    for (let i = 0; i < keepForCrossOver.length - 1; i++)
    {
        var x = CrossOver(keepForCrossOver[i], keepForCrossOver[i + 1]);
        nextGen.push(x);
    }
    nextGen = Mutate(nextGen);
    return nextGen;
}

const CalculateFitness = () => {
    let fitness = 9999999;
    let bestSolution = [];

    var population = GeneratePopulation(200);

    //For times experiment uncomment below and comment out forloop on line 112
    // var t0 = performance.now()
    // while (performance.now() - t0 < 1000)
    //     
    
}
for (let j = 0; j < 100; j++)
{
    var orderedSolutions = {};
    population.forEach(item => {
        let currentSolution = 0;
        for (let i = 0; i < item.length - 1; i++)
        {
            currentSolution += costMatrix[item[i]][item[i + 1]];
        }

        if (currentSolution < fitness)
        {
            fitness = currentSolution;
            bestSolution = item;
        }
        if (!Object.keys(orderedSolutions).includes(item))
        {
            orderedSolutions[currentSolution] = item;
        }
    })

    const sortable = Object.fromEntries(
        Object.entries(orderedSolutions).sort(([a], [b]) => a-b)
    );
    let forNextGen = Object.values(sortable);
    population = NextGeneration(forNextGen);
}
console.log(fitness);
return fitness;
}

for (let i = 0; i < 100; i++) {
    CalculateFitness();
}
Javascript Brute Force code

```javascript
const performance = require('perf_hooks').performance;

const permutator = (inputArr) => {
    let result = [];

    const permute = (arr, m = []) => {
        if (arr.length === 0) {
            result.push(m)
        } else {
            for (let i = 0; i < arr.length; i++) {
                let curr = arr.slice();
                let next = curr.splice(i, 1);
                permute(curr.slice(), m.concat(next))
            }
        }
    }

    permute(inputArr)

    return result;
}

const costMatrix = [
    [0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145, 1972 ],
    [2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357, 579 ],
    [713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453, 1260 ],
    [1018, 1524, 355, 0, 700, 862, 1395, 1123, 1584, 466, 1056, 1280, 987 ],
    [1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371 ],
    [1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 225, 887, 999 ],
    [2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1114, 701 ],
    [213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300, 2099 ],
    [2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653, 600 ],
    [875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272, 1162 ],
    [1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017, 1200 ],
    [2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0, 504 ],
    [1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504, 0 ],
]```
const b = [
    [0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420],
    [2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374],
    [713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940],
    [1018, 1524, 355, 0, 700, 663, 1021, 1769, 949, 796, 879],
    [1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879],
    [1374, 1240, 803, 663, 0, 1681, 1551, 1765, 547, 225],
    [2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891],
    [213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605],
    [2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645],
    [875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679],
    [1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0]
]

var t0 = performance.now()
let shortest = 99999;
permutator(b).forEach(x => {
x.push(x[0]);
let current = 0;
for (let i = 0; i < x.length - 2; i++) {
    let top = b.indexOf(x[i])
    let nested = b.indexOf(x[i+1])
    current += costMatrix[top][nested]
}
if (current < shortest) {
    shortest = current
    best = x
}
})

var t1 = performance.now()
console.log(shortest);

console.log("Brute force duration: " + (t1 - t0) + " milliseconds.")
import random
import time

costMatrix = [
    [0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145, 1972],
    [2451, 0, 1745, 1524, 831, 1240, 959, 1858, 262, 940, 1453, 1260],
    [713, 1745, 0, 355, 920, 803, 1737, 851, 2596, 403, 1584, 2145, 1972],
    [1018, 1524, 355, 0, 700, 663, 1021, 1769, 949, 796, 879, 586, 371],
    [1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371],
    [1374, 1240, 803, 663, 0, 1681, 1551, 1765, 547, 225, 887, 999],
    [2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1114, 701],
    [213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300, 2099],
    [2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653, 600],
    [875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272, 1162],
    [1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017, 1200],
    [2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0, 504],
    [1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504, 0],
]
pheromoneMatrix = [
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1]
]
def nextDestination(city, pheromone, visited):
denominator = 0
probabilities = {}
availableDestinations = list(filter((lambda d: d != 0 and city.index(d) not in visited), city))

if len(availableDestinations) == 0:
    return

for i in range(0, len(availableDestinations)):
    pheromoneLevel = pheromone[city.index(availableDestinations[i])]
    distance = 1.0 / availableDestinations[i]
    product = pheromoneLevel * distance
    denominator += product

for i in range(0, len(availableDestinations)):
    pheromoneLevel = pheromone[city.index(availableDestinations[i])]
    distance = 1.0 / availableDestinations[i]
    probability = (pheromoneLevel * distance) / denominator
    probabilities[city.index(availableDestinations[i])] = probability

sortable = dict(sorted(probabilities.items(), key=lambda item: item[1], reverse=True))
updatedProbabilityValues = dict()

for k, v in sortable.items():
    accumulated_value = sum(sortable.values())
    updatedProbabilityValues[k] = accumulated_value
    sortable[k] = 0

rouleProbability = random.uniform(0, 1)
temp = list(updatedProbabilityValues)

for i in range(0, len(updatedProbabilityValues.items()) - 1):
    if rouleProbability <= updatedProbabilityValues[temp[i]] and rouleProbability >= updatedProbabilityValues[temp[i + 1]]:
        return temp[i]
return temp[-1]

def updatePheromone(routeLength, route):

    pheromoneToApply = 1.0 / routeLength
    for i in range(0, len(route) - 1):
        startingPheromonePosition = pheromoneMatrix[route[i]]
        startingPheromonePosition[route[i + 1]] += pheromoneToApply


def CostCalculator(route):

    routeLength = 0
    for i in range(0, len(route) - 1):
        startingCity = costMatrix[route[i]]
        routeLength += startingCity[route[i + 1]]

    return routeLength

def shortestDistance():

    order = []
    bestOrder = []
    shortestRoute = 999999
    #For times experiment uncomment below and comment for loop and comment out for loop on line 101
    #start_time = time.time()
    # while time.time() - start_time < 1.6:
    for i in range(0, 100):

        for j in range(0, 200):
            order = []
            count = 0
            rng_index = random.randint(0, 12)
            currentCity = costMatrix[rng_index]
            order.append(costMatrix.index(currentCity))
            while count < len(costMatrix) - 1:
nextIndex = nextDestination(currentCity, pheromoneMatrix[costMatrix.index(currentCity)], order)
    if nextIndex or nextIndex == 0:
        currentCity = costMatrix[nextIndex]
        order.append(costMatrix.index(currentCity))
        count += 1

    order.append(order[0])
    routeLength = CostCalculator(order)

    if routeLength < shortestRoute:
        shortestRoute = routeLength

    updatePheromone(routeLength, order)

print(shortestRoute)
return shortestRoute

for i in range(0, 100):
    shortestDistance()
import random
import time

costMatrix = [
    [ 0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145, 1972 ],
    [ 2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357, 579 ],
    [ 713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453, 1260 ],
    [ 1018, 1524, 355, 0, 700, 862, 1395, 1123, 1584, 466, 1056, 1280, 987 ],
    [ 1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371 ],
    [ 1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 225, 887, 999 ],
    [ 2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1114, 701 ],
    [ 213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300, 2099 ],
    [ 2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653, 600 ],
    [ 875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272, 1162 ],
    [ 1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017, 1200 ],
    [ 2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0, 504 ],
    [ 1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504, 0 ],
]

b = [
    [ 0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145, 1972 ],
    [ 2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357, 579 ],
    [ 713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453, 1260 ],
    [ 1018, 1524, 355, 0, 700, 862, 1395, 1123, 1584, 466, 1056, 1280, 987 ],
    [ 1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371 ],
    [ 1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 225, 887, 999 ],
    [ 2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1114, 701 ],
    [ 213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300, 2099 ],
    [ 2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653, 600 ],
    [ 875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272, 1162 ],
    [ 1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017, 1200 ],
    [ 2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0, 504 ],
    [ 1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504, 0 ],
]

def RandomOrder():

order = []
while (len(order) < len(costMatrix)):
    rng = random.randint(0, 12)
    if costMatrix.index(costMatrix[rng]) not in order:
        order.append(rng)

return order

def GenerateSolution():
    solution = RandomOrder()
    solution.append(solution[0])

    return solution

def GeneratePopulation(size):
    population = []
    for i in range(0, size):
        population.append(GenerateSolution())

    return population

def CrossOver(first, second):
    firstInterim = first[:5]
    for city in second:
        if city not in firstInterim:
            firstInterim.append(city)

    firstInterim.append(firstInterim[0])
    return firstInterim

def Mutate(popToMutate):
for city in popToMutate:
    rng = random.randint(0, 1000)
    if rng < 500:
        oldIndex = random.randint(1,10)
        newIndex = random.randint(1,10)
        city.insert(newIndex, city[oldIndex])
        if newIndex <= oldIndex:
            oldIndex += 1
        city.pop(oldIndex)
        city.insert(oldIndex, city[newIndex])
        if oldIndex <= newIndex:
            newIndex += 1
        city.pop(newIndex)

return popToMutate

def NextGeneration(currentGeneration):

currentGeneration = list(currentGeneration)
nextGen = []
for i in range(0, 50):
    nextGen.append(currentGeneration[i])

keepForCrossOver = currentGeneration[:150]

for i in range(0, len(keepForCrossOver) - 1):
    x = CrossOver(keepForCrossOver[i], keepForCrossOver[i + 1])
    nextGen.append(x)

nextGen = Mutate(nextGen)
return nextGen

def CalculateFitness():

    fitness = 9999999
    bestSolution = []
population = GeneratePopulation(200)

#For times experiment uncomment below and comment for loop and comment out for loop on line 121
start_time = time.time()
# while time.time() - start_time < 2:
for j in range(0, 100):
    
    orderedSolutions = {}

    for item in population:
        currentSolution = 0
        for i in range(0, len(item) - 1):
            currentSolution += costMatrix[item[i]][item[i + 1]]

        if currentSolution < fitness:
            fitness = currentSolution
            bestSolution = item

        if item not in orderedSolutions.values():
            orderedSolutions[currentSolution] = item

    sortable = dict(sorted(orderedSolutions.items(), key=lambda item: item[0]))

    forNextGen = sortable.values()

    population = NextGeneration(forNextGen)

    print(fitness)
    return fitness

for i in range(0, 100):
    CalculateFitness()
```python
import itertools
import time

costMatrix = [
    [0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145, 1972 ],
    [2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357, 579 ],
    [713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453, 1260 ],
    [1018, 1524, 355, 0, 700, 862, 1398, 262, 1858, 262, 940, 1453, 1260 ],
    [1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371 ],
    [1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 225, 887, 999 ],
    [2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1114, 701 ],
    [213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300, 2099 ],
    [2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653, 600 ],
    [875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272, 1162 ],
    [1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017, 1200 ],
    [2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0, 504 ],
    [1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504, 0 ],
]

c = [0,1,2,3,4,5,6,7,8,9,10]
count = 0
shortest = 99999
best = []
start_time = time.time()

for x in list(itertools.permutations(c)):
    x = list(x)
    x.append(x[0])
    current = 0
    for i in range(0, len(x) - 1):
        current += (costMatrix[x[i]][x[i+1]])
    if current < shortest:
        shortest = current
        best = x

print(shortest)
```
C# ACO code

```csharp
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TSP
{
    public class AntColonyAlgorithm
    {
        private readonly List<List<int>> distanceMatrix = new Matrices().GenerateDistanceMatrix();
        private readonly List<List<float>> pheromoneMatrix = new Matrices().GeneratePheromoneMatrix();
        private float CostCalculator(List<int> route)
        {
            float routeLength = 0;
            for (int i = 0; i < route.Count - 1; i++)
            {
                List<int> startingCity = distanceMatrix[route[i]];
                routeLength += startingCity[route[i + 1]];
            }
            return routeLength;
        }

        private void updatePheromone(float routeLength, List<int> route)
        {
            float pheromoneToApply = 1.0f / routeLength * 100;
            for (int i = 0; i < route.Count - 1; i++)
            {
                List<float> startingPheromonePosition = pheromoneMatrix[route[i]];
                startingPheromonePosition[route[i + 1]] += pheromoneToApply;
            }
        }
    }
}
```
private int nextDestination(List<int> city, List<float> pheromone, List<int> visited)
{
    float denominator = 0;
    Dictionary<int, float> probabilities = new Dictionary<int, float>();
    List<int> availableDestinations = city.Where(d => d != 0 &&
    !visited.Contains(city.IndexOf(d))).ToList();
    for (int i = 0; i < availableDestinations.Count; i++)
    {
        float pheromoneLevel =
            pheromone[city.IndexOf(availableDestinations[i])];
        float distance = 1.0f / availableDestinations[i];
        float product = pheromoneLevel * distance;
        denominator += product;
    }
    for (int i = 0; i < availableDestinations.Count; i++)
    {
        float pheromoneLevel =
            pheromone[city.IndexOf(availableDestinations[i])];
        float distance = 1.0f / availableDestinations[i];
        float probability = (pheromoneLevel * distance) / denominator;
        probabilities.Add(city.IndexOf(availableDestinations[i]),
            probability);
    }
    var ordered = probabilities.OrderByDescending(x =>
        x.Value).ToDictionary(x => x.Key, x => x.Value);
    List<int> keys = ordered.Keys.ToList();
    List<float> values = ordered.Values.ToList();
    List<float> updatedProbabilityValues = new List<float>();
    for (int i = 0; i < keys.Count; i++)
    {
        float total = values.Aggregate((a, c) => a + c);
        updatedProbabilityValues.Add(total);
        values.RemoveAt(0);
    }
Dictionary<float, int> finalProbabilities = new Dictionary<float, int>();

for (int i = 0; i < keys.Count; i++)
{
    finalProbabilities.Add(updatedProbabilityValues[i], keys[i]);
}

double rouletteProbability = new Random().NextDouble();
float test = (float)rouletteProbability;

for (int i = 0; i < finalProbabilities.Count - 1; i++)
{
    {
        return finalProbabilities[finalProbabilities.Keys.ToList()[i]];
    }
}

public float ShortestDistance()
{
    List<int> order = new List<int>();
    float shortestRoute = 999999;

    //For time experiment uncomment below and comment out for loop on line 99
    //var watch = new Stopwatch();
    //watch.Start();
    //while (watch.ElapsedMilliseconds < 200)
    for (int i = 0; i < 100; i++)
for (int j = 0; j < 200; j++)
{
    order.Clear();
    int count = 0;
    var currentCity = distanceMatrix[new Random().Next(0,5)];
    order.Add(distanceMatrix.IndexOf(currentCity));

    while (count < distanceMatrix.Count() - 1)
    {
        var nextNodeIndex = nextDestination(currentCity, 
                                   pheromoneMatrix[distanceMatrix.IndexOf(currentCity)], order);
        currentCity = distanceMatrix[nextNodeIndex];
        order.Add(distanceMatrix.IndexOf(currentCity));
        count++;
    }
    order.Add(distanceMatrix.IndexOf(distanceMatrix[order[0]]));
    float routeLength = CostCalculator(order);
    if (routeLength < shortestRoute)
    {
        shortestRoute = routeLength;
    }
    updatePheromone(routeLength, order);
}

return shortestRoute;
C# GA code

```csharp
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TSP
{
    class GeneticAlgorithm
    {
        public List<List<int>> DistanceMatrix = new Matrices().GenerateDistanceMatrix();

        public List<int> RandomOrder()
        {
            List<int> order = new List<int>();

            while (order.Count() < DistanceMatrix.Count)
            {
                var rng = new Random().Next(0, 11);
                if (!order.Contains(DistanceMatrix.IndexOf(DistanceMatrix[rng])))
                {
                    order.Add(rng);
                }
            }
            return order;
        }

        public List<List<int>> GenerateSolution()
        {
            var solution = RandomOrder().Select(x => DistanceMatrix[x]).ToList();
            solution.Add(solution[0]);

            return solution;
        }
    }
}
```
public List<List<List<int>>> GeneratePopulation(int size)
{
    List<List<List<int>>> population = new List<List<List<int>>>();
    foreach (int value in Enumerable.Range(0, size))
    {
        population.Add(GenerateSolution());
    }
    return population;
}

public List<List<int>> CrossOver(List<List<int>> first, List<List<int>> second)
{
    List<List<int>> firstInterim = first.Take(5).ToList();
    foreach (var city in second)
    {
        if (!firstInterim.Contains(city))
        {
            firstInterim.Add(city);
        }
    }
    firstInterim.Add(firstInterim[0]);
    return firstInterim;
}

public List<List<List<int>>> Mutate(List<List<List<int>>> popToMutate)
{
    foreach (var city in popToMutate)
    {
        var rng = new Random().Next(0, 1000);
        if (rng < 900)
        {
            int oldIndex = new Random().Next(1, 10);
            int newIndex = new Random().Next(1, 10);
            city.Insert(newIndex, city[oldIndex]);
            if (newIndex <= oldIndex) ++oldIndex;
            city.RemoveAt(oldIndex);
        }
    }
}
city.Insert(oldIndex, city[newIndex]);
if (oldIndex <= newIndex) ++newIndex;
city.RemoveAt(newIndex);
}
}
return popToMutate;

public List<List<List<int>>> NextGeneration(List<List<List<int>>> currentGeneration)
{
    List<List<List<int>>> nextGen = new List<List<List<int>>>();
    for (int i = 0; i < 50; i++)
    {
        nextGen.Add(currentGeneration[i]);
    }
    var keepForCrossOver = currentGeneration.Take(150).ToList();
    for (int i = 0; i < keepForCrossOver.Count() - 1; i++)
    {
        var x = CrossOver(keepForCrossOver[i], keepForCrossOver[i + 1]);
        nextGen.Add(x);
    }
    nextGen = Mutate(nextGen);
    return nextGen;
}

public int CalculateFitness()
{
    int fitness = 9999999;
    List<List<int>> bestSolution = new List<List<int>>();
    var population = GeneratePopulation(200);

    // For time experiment uncomment below and comment out for loop on line 114
    // var watch = new Stopwatch();
    // watch.Start();


// while (watch.ElapsedMilliseconds < 1000)
for (int j = 0; j < 100; j++)
{
    var orderedSolutions = new Dictionary<List<List<int>>, int>();
    foreach (var item in population)
    {
        int currentSolution = 0;
        for (int i = 0; i < item.Count - 1; i++)
        {
            int next = DistanceMatrix.IndexOf(item[i + 1]);
            currentSolution += item[i][next];
        }
        if (currentSolution < fitness)
        {
            fitness = currentSolution;
            bestSolution = item;
        }
        if (!orderedSolutions.Keys.Contains(item))
        {
            orderedSolutions.Add(item, currentSolution);
        }
    }
    orderedSolutions = orderedSolutions.OrderBy(x => x.Value).ToDictionary(x => x.Key, x => x.Value);
    var x = orderedSolutions.Keys.ToList();
    population = NextGeneration(x);
    // watch.Stop(); uncomment this if using timed experiments
}
return fitness;
}
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TSP
{
    public class AllPerms
    {
        public static IEnumerable<IEnumerable<T>> GetPermutations<T>(IEnumerable<T> list, int length)
        {
            if (length == 1) return list.Select(t => new T[] { t });
            return GetPermutations(list, length - 1)
                .SelectMany(t => list.Where(o => !t.Contains(o)),
                    (t1, t2) => t1.Concat(new T[] { t2 }));
        }
    }
}