International Baccalaureate

Extended Essay

Computer Science

The Travelling Salesman Problem

How does the Elitist Genetic Algorithm compare to Ant Colony System in terms of time complexity and accuracy when attempting to solve the Travelling Salesman Problem?

Word count: 3,818 words

Table of Contents

1. Introduction

1.1 Background Information

In computational complexity theory, a problem is assigned to the NP (non-deterministic polynomial) class if it can be verified in polynomial time. The *Travelling Salesman Problem* (TSP) is potentially the most famous optimisation problem and it falls under the *NP-hard* category since the existence of a polynomial-time solution for it implies the existence of a polynomial-time solution for every problem in NP.[1] The TSP consists of determining the shortest tour to complete a *Hamiltonian Cycle* – a path through a graph that starts and ends at the same vertex, including every other vertex exactly once; an example is shown in Figure 1.[2]

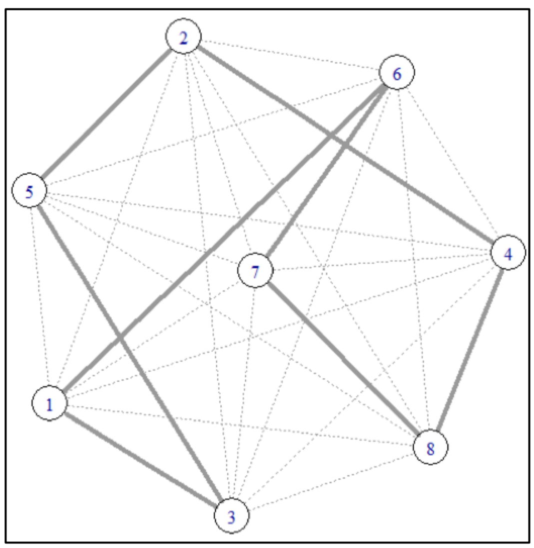


Figure 1: A Hamiltonian cycle with 8 vertices[3]

[1]*NP-hard problems and approximation algorithms - University of Texas at ...* Available at: https://personal.utdallas.edu/~dxd056000/cs6363/unit5.pdf (Accessed: 17 May 2023).

[2] Black, P.E. (2020) *Hamiltonian cycle*, *Dictionary of Algorithms and Data Structures* Available at: https://xlinux.nist.gov/dads/HTML/hamiltonianCycle.html (Accessed: 17 May 2023).

[3] Chatting, M. (2018) 'A Comparison of Exact and Heuristic Algorithms to Solve the Travelling Salesman Problem', The Plymouth Student Scientist, 11(2), p. 53-91.

1.2 Contextual Significance

The TSP has stimulated the development of various problem-solving techniques, algorithms and innovative mathematical models that can be applied beyond its immediate problem space. For instance, this can be applied to the sphere of network and hardware optimisation where the most efficient route for data transmission ought to be found. The formulation as a TSP essentially provides the simplest way to solve problems arising in many different contexts, including computer wiring, vehicle routing, clustering, and job-shop scheduling.

1.3 Scope of Research

There are two variations of the TSP: *asymmetric Travelling Salesman Problem* (ATSP), where the distance from node A to B differs to that from B to A, and *symmetric Travelling Salesman Problem* (STSP), where the graph is undirected.[4] Hence, regarding the STSP, it can be said that $c_{ij} = c_{ji}$; this simply states that regardless of the direction of travel, the cost (or distance) between city i and city j is constant.[5] Numerous algorithms have been generated to approximate a solution to the TSP in a feasible time span since finding the optimal solution for large problem instances is computationally challenging.

The aim of the paper is to determine the most efficient algorithm for solving the STSP by analysing both the time complexity and accuracy of the algorithms. In this case, the most efficient algorithm can be quantified as the one that has the greatest accuracy and shortest execution time.

---

[4] Deep, Kusum., & Mebrahtu, Hadush. (2012), "Variant of partially mapped crossover for the Travelling Salesman problems." International Journal of Combinatorial Optimization Problems and Informatics, Vol.3, num.1, pp.47-69. ISSN: 2007-1558

[5] Ibid.

1.4 Experimental Overview

This paper specifically focuses on comparing the Elitist Genetic Algorithm to Ant Colony System, evaluating which of the two is most efficient at solving the STSP. Random data sets of increasing size will be used to collect a set of execution time periods for each of the algorithms. Furthermore, the accuracy of the algorithms will be obtained by comparing the shortest distance calculated to that determined by a control algorithm; for this experiment, the Brute Force Algorithm will be used - an exact algorithm, so the optimal solution is guaranteed to be found, with time complexity of O(n!).[6] These two factors were then analysed to answer the question: ***"How does the Elitist Genetic Algorithm compare to Ant Colony System in terms of time complexity and accuracy when attempting to solve the Travelling Salesman Problem?"***

[6] Chase, C. *et al.* (no date) *An Evaluation of the Traveling Salesman Problem*. Available at: https://scholarworks.calstate.edu/downloads/xg94hr81q#:~:text=. (Accessed: 14 July 2023).

2. Research

2.1 Genetic Algorithms

Genetic Algorithms (GA) are adaptive, stochastic, metaheuristic search algorithms that are a subclass of evolutionary computing used to solve combinatorial optimisation problems.[7] Combinatorial optimisation is the process of solving for the optimal solution of a finite data set by using combinatorial techniques.[8] Metaheuristics are algorithmic concepts that define heuristic methods applicable to a number of different problems.[9] Darwin's theory of evolution (survival of the fittest and natural selection) forms the backbone of GAs; Figure 2 describes the steps of this process.

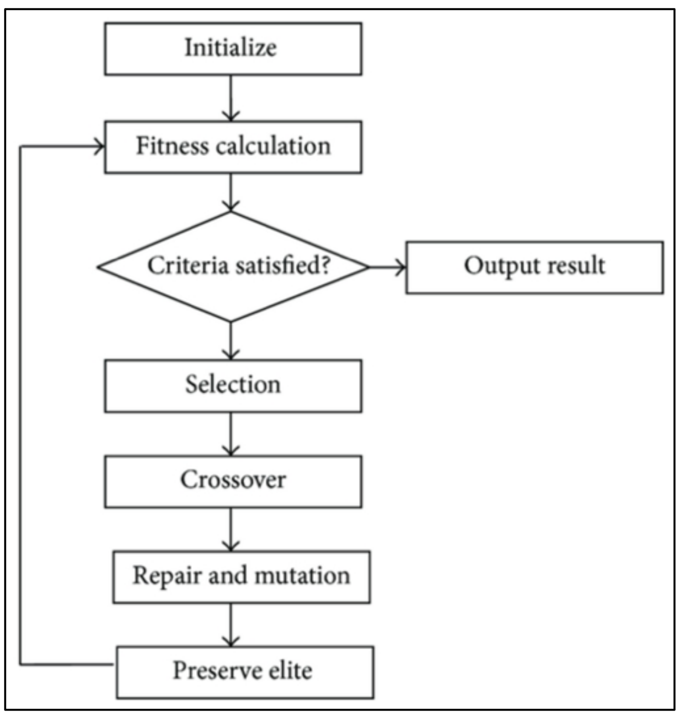


Figure 2: Flowchart for the Elitist Genetic Algorithm[10]

[7] *Genetic algorithms* (2017) *Scribd*. Available at: https://www.scribd.com/document/351623322/Genetic-Algorithms# (Accessed: 15 June 2023).
[8] Ibid.
[9] Dorigo, M. and Stützle, T. (2004) 'The Ant Colony Optimization Metaheuristic', in *Ant colony optimization*. Cambridge, MA: MIT Press, pp. 25–26.
[10] Singh, V.K. and Sharma, V. (2014) 'Elitist genetic algorithm based energy balanced routing strategy to prolong lifetime of wireless sensor networks', *Chinese Journal of Engineering*, 2014, pp. 1–6. doi:10.1155/2014/437625.

These algorithms are more robust and can navigate through larger data sets whilst finding an optimal solution within a sensible timespan. Chromosomes are used to represent *n* number of genes (cities) in the order in which they are visited, typically as a binary string. Figure 3 shows a *chromosome* of a 5-city tour, where the salesman begins at city 3, then travels to city 4 and so on.

3	2	4	1	5

Figure 3: Chromosome of a 5-city tour

2.1.1 Population Initialisation

In terms of the TSP, the city tour is referred to as the *population.* The first process of a GA is to initialise the population – the cities can be randomly generated or set before. If the initial population size is too small, diversity is prohibited which forces some optimisation routines to converge too quickly, causing the population to become homogenous.[11] However, if the population size is too big, it would take a long time for the optimisation routine to converge.[12]

2.1.2 Fitness Evaluation

As each city is situated in a 2D plane, their position can be given by (x_i, y_i). Using Equation (1) and the position coordinates of two cities, the cost c_{ij}, in other words the distance between the ith and jth city, can be found:

$$c_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \quad (1)$$ [13]

[11] Morris, A.T. (1998) *Optimization of the Traveling Salesman Problem and Multivariate Real-Valued Functions using a Genetic Algorithm*. dissertation.
[12] Ibid.
[13] Deep, Kusum., & Mebrahtu, Hadush. (2012), *op. cit.*

The distances between the cities can be represented in an $n \times n$ cost matrix. Below is an example of a symmetric 5-city tour.

$$\begin{array}{c} \\ \text{City 1} \\ \text{City 2} \\ \text{City 3} \\ \text{City 4} \\ \text{City 5} \end{array} \begin{array}{c} \begin{array}{ccccc} \text{City 1} & \text{City 2} & \text{City 3} & \text{City 4} & \text{City 5} \end{array} \\ \begin{bmatrix} 0 & 8 & 14 & 23 & 2 \\ 8 & 0 & 26 & 2 & 19 \\ 14 & 26 & 0 & 2 & 23 \\ 23 & 2 & 2 & 0 & 11 \\ 2 & 19 & 23 & 11 & 0 \end{bmatrix} \end{array}$$

Each chromosome is then evaluated using a fitness function and is assigned a fitness value. The fitness value is determined based on the distance between the two cities using Equation (2):

$$Fitness = \sum_{i=1}^{n} distance(x_i \ to \ x_{i+1}) \quad (2)$$

[14],

where n is the number of cities and i is the city index value.

The shortest distance is given the highest fitness value.

2.1.3 Selection and Elitism

A random sample of chromosomes is placed in a mating pool via the reproductive process, *selection*, by which the fitter chromosomes, those with shorter city tours, are more likely to be reproduced to the next generation.[15] *Elitism* is also considered part of the selection process for this paper, meaning it is guaranteed that the best chromosome, the one with the shortest distance, is copied to the next generation.[16]

[14] Deep, Kusum., & Mebrahtu, Hadush. (2012), *op. cit.*
[15] Morris, A.T. (1998) *Optimization of the Traveling Salesman Problem and Multivariate Real-Valued Functions using a Genetic Algorithm*. dissertation.
[16] Ibid.

2.1.4 Evolution through PMX Crossover

Crossover is a genetic operator by which design characteristics between two parent chromosomes, chosen randomly from the mating pool, are exchanged to form two new superior offspring.[17] For this experiment, the *Partially Mapped Crossover* (PMX) operator will be used as one of the most effective and popular. It is a modification of the basic two-point crossover, however, uses an additional mapping relationship to avoid duplicate values in the offspring that often lead to infeasible results.[18] PMX falls into the category of Inventing Specialised Operators - meaning only valid chromosomes are generated (cities are not missing or repeated).[19]

Below is an example PMX crossover, where P_1 and P_2 are random, parent chromosomes of an 8-city tour. Offspring O_1 and O_2 are formed which each represent a new city tour.

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

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

A substring is selected using two random crossover points (marked with “|”):

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

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

A Two-Point Crossover is performed:

$$O_1 = (x\ x\ x\ |\ 3\ 6\ 2\ |\ x\ x),$$

$$O_2 = (x\ x\ x\ |\ 5\ 7\ 3\ |\ x\ x).$$

[17] Hasançebi, O. and Erbatur, F. (2000) ‘Evaluation of crossover techniques in genetic algorithm based optimum structural design’, *Computers & Structures*, 78(1–3), pp. 435–448. doi:10.1016/s0045-7949(00)00089-4.

[18] Deep, Kusum., & Mebrahtu, Hadush. (2012), *op. cit.*

[19] Üçoluk, G. (2002) ‘Genetic algorithm solution of the TSP avoiding special crossover and mutation’, *Intelligent Automation & Soft Computing*, 8(3), pp. 265–272. doi:10.1080/10798587.2000.10642829.

The mapping systems are determined:

$$5 \leftrightarrow 3, 7 \leftrightarrow 6, 3 \leftrightarrow 2.$$

Bits that are not conflicting are filled:

$$O_1 = (4\ 1\ x \mid 3\ 6\ 2 \mid x\ 8),$$

$$O_2 = (1\ x\ 8 \mid 5\ 7\ 3 \mid 4\ x).$$

Using the mapping relationships, the offspring can be fully filled:

$$O_1 = (4\ 1\ 5 \mid 3\ 6\ 2 \mid 7\ 8),$$

$$O_2 = (1\ 2\ 8 \mid 5\ 7\ 3 \mid 4\ 6).$$

2.1.5 Evolution through Mutation

Mutation is a unary genetic operator which produces spontaneous, random changes on one parent chromosome.[20] Only one type of mutation, the *swap mutation*, will be used to ensure reliability in the results as this acts as a control variable. Two genes (cities) are selected at random, and their positions are swapped.

An example swap mutation is shown, where P_1 is the parent chromosome of an 8-city tour.

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

Two cities are chosen:

$$P_1 = (4\ \mathbf{1}\ 2\ 5\ 7\ \mathbf{3}\ 6\ 8).$$

Their values are interchanged:

$$O_1 = (4\ \mathbf{3}\ 2\ 5\ 7\ \mathbf{1}\ 6\ 8).$$

[20] GeeksforGeeks. (2018). *Mutation Algorithms for String Manipulation (GA)*. [online] Available at: https://www.geeksforgeeks.org/mutation-algorithms-for-string-manipulation-ga/.

2.2 Ant Colony Optimisation

Ant Colony Optimisation (ACO) is another stochastic, population-based, metaheuristic search algorithm that simulates the foraging behaviour of ants to solve combinatorial optimisation problems.[21] The concept of using swarm intelligence, the collective behaviour of decentralised, self-organised natural or artificial systems, was first introduced by Gerado Beni and Jing Wang in 1989.[22]

2.2.1 Real Ant Behaviour

Ants use stigmergy meaning they indirectly communicate with each other by altering their surrounding environment. They have collective intelligence as they lay a *pheromone trail* while searching for food to communicate with each other to find the shortest path; other ants can sense this chemical, influencing their choice of path. Pheromone is a particularly volatile substance that starts to evaporate after the ant marches over the path. Hence, for shorter paths, the pheromone density remains high as pheromone accumulation is faster.[23] Figure 4 shows how the distribution of pheromones is dependent on path length.

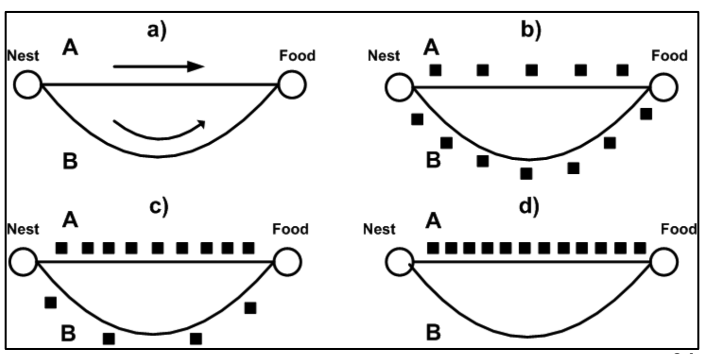


Figure 4: Pheromone trails of path A and B[24]

[21] Ahmed, Z.E. *et al.* (2020) 'Energy optimization in low-power wide area networks by using heuristic techniques', *LPWAN Technologies for IoT and M2M Applications*, pp. 199–223. doi:10.1016/b978-0-12-818880-4.00011-9.

[22] Ibid.

[23] Ranjith, K.A. (2010) *Ant Colony Optimization*. rep., pp. 1–16.

[24] Nguyen, K.-H. and Ock, C.-Y. (2011) 'Word sense disambiguation as a traveling salesman problem', *Artificial Intelligence Review*, 40(4), pp. 405–427. doi:10.1007/s10462-011-9288-9.

Initially, there is an equal probability p that an ant will travel via path A or B when in search of food. As path A is shorter than path B, in a specific time period t, path A will be travelled more times; therefore, path A will have a higher pheromone density. As t increases, more ants will follow path A whilst the pheromone trails in path B will all evaporate - eventually all ants follow path A.[25]

2.2.3 Ant Colony System

There are many variations of ACO algorithms including Rank-Based Ant System (ASrank), Max-Min Ant System (MMAS) and Ant Colony System (ACS). In this paper, the ACS algorithm will be investigated - a set of cooperating agents, *ants*, indirectly communicate with each other through the deposited pheromones on the edges of the TSP graph whilst finding the optimal solution.[26] All ants perform the local pheromone update after every step rather than after a completed tour meaning the next edge is chosen purely based on the updated pheromone value. The process stops when the best solution is found or there are no more pheromone updates.[27] During ACS, an ant completes a tour around the map n number of times. After every iteration, the ant's global memory is reset; meaning, they have no knowledge of the journey they took prior.

---

[25] Ranjith, K.A. (2010), Op. cit,

[26] Dorigo, M. and Gambardella, L.M. (1997) 'Ant Colony System: A cooperative learning approach to the traveling salesman problem', *IEEE Transactions on Evolutionary Computation*, 1(1), pp. 53–66. doi:10.1109/4235.585892.

[27] Mulani, M. and Desai, V.L. (2018) 'Design and Implementation Issues in Ant Colony Optimization', in *International Journal of Applied Engineering Research*. 16th edn. Research India Publications, pp. 12877–12882.

Figure 5 below describes the steps of an ACS where the evaluation stop condition is when n number of iterations have taken place.

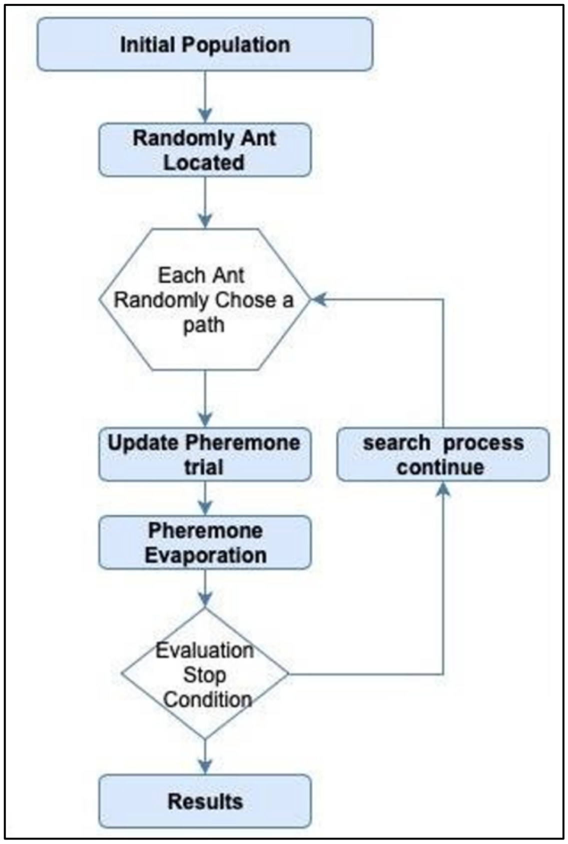


Figure 5: Flowchart for ACS[28]

2.2.3 ACO Algorithms for the TSP

When solving the TSP, this algorithm assumes that ants are always able to determine the shortest path to the food sources by detecting the pheromones laid by other ants. In other words, cities further away are less visible meaning there is a lower probability of being chosen. The greater the intensity of the pheromone trail, the greater the probability that the ant will choose that edge.

[28] M. Almufti, S., Boya Marqas, R. and Ashqi Saeed, V. (2019) ‘Taxonomy of bio-inspired optimization algorithms’, *Journal of Advanced Computer Science & Technology*, 8(2), p. 23. doi:10.14419/jacst.v8i2.29402.

Pheromone trails describe the desirability of an ant visiting node j after node i. As evident in Equation (3), the heuristic desirability, η_{ij}, of an ant going from node i to node j is inversely proportional to the distance, d_{ij}, between the nodes:

$$\eta_{ij} = 1/d_{ij}.^{30} \tag{3}$$

At each node, the ant plans based on its local memory – this stores information about the adjacent nodes. Nodes that have not been visited by the nth ant are defined as $allowed_n$. The probability of the nth ant choosing one node is given by Equation (4):

$$p_{ij}^{n}(t) = \begin{cases} \frac{[\tau_{ij}(t)]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{n \in allowed_n}[\tau_{ij}(t)]^{\alpha}[\eta_{ij}]^{\beta}} & j \in allowed_n \\ 0 & else, \end{cases} \tag{4}$$

where τ_{ij} is the evaporation rate, determining the amount of pheromone present between node i and j, $\alpha \geq 0$ is a parameter in control of the influence of τ_{ij}, η_{ij} is the desirability of the transition from i to j and $\beta \geq 1$ is a parameter in control of the influence of η_{ij}.[31] These parameters must be selected properly otherwise the convergence speed may be too high causing the algorithm to fall rapidly into local optima. Likewise, if the parameters are set so that the convergence speed is slow, time complexity increases.

[30] Dorigo, M. and Stützle, T. (2004) 'Ant Colony Optimization Algorithms for the Traveling Salesman Problem', in *Ant colony optimization*. Cambridge, MA: MIT Press, pp. 67–68.

[31] Danu, M.S. (2013) *Ant colony optimization algorithms*, *Scribd*. Available at: https://www.scribd.com/document/136679005/Ant-colony-optimization-algorithms# (Accessed: 27 June 2023).

2.3 Parameters for Analysis

2.3.1 Time Complexity

The first parameter analysed for each algorithm is the time complexity. For a given problem, the *time complexity* is defined as the maximum time the algorithm requires to find a solution for each possible input size, n.[32] This is alternatively referred to as the worst-case time complexity. Big-O notation is typically used to describe time complexity for the function $f(n)$ as it gives asymptotic upper bounds for the worst-case scenario:

$$f(n) = O\big(g(n)\big) \quad for\ n \to \infty \quad and \quad f(n), g(n) \in R$$

where $g(n)$ represents the big-O notation.[33] Equation 4 is only valid if there exist constants c and n_0 such that

$$|f(n)| \leq c|g(n)| \quad for\ all \quad n > n_0.$$[34]

This effectively means that the big-O notation, denoted by $g(n)$, is always greater than or equal to the number of steps. Determining the time complexities of different algorithms enables their efficiency to be compared and analysed.

[32] *Big O notation - mit - massachusetts institute of technology* (no date) *Big O Notation*. Available at: https://web.mit.edu/16.070/www/lecture/big_o.pdf (Accessed: 26 June 2023).
[33] Ibid.
[34] Ibid.

For example, as seen in Figure 6, an algorithm with a time complexity of $O(\log n)$ is significantly more efficient than one which has a time complexity of $O(n!)$.

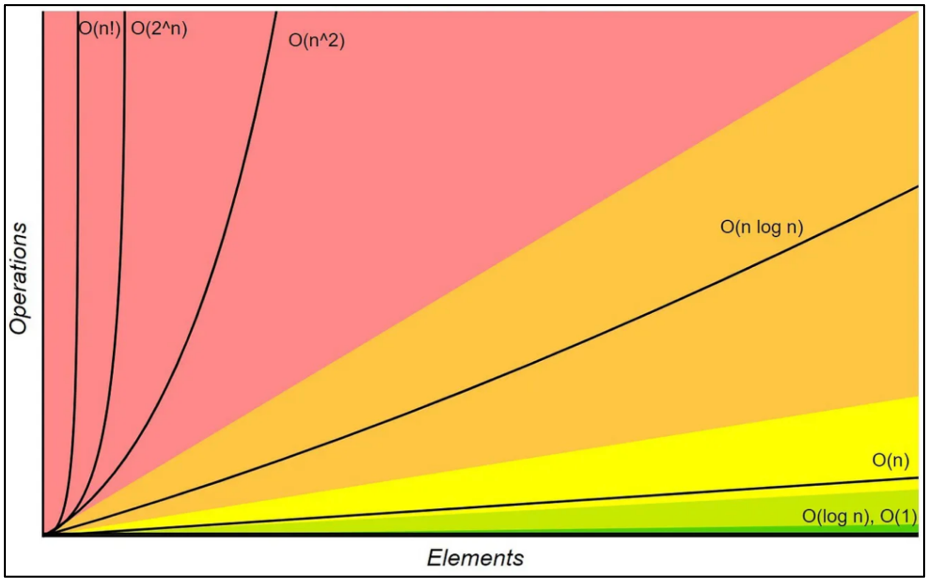


***Figure 6: A graph to show different time complexities*[35]**

The execution time of NP-hard problems increases exponentially with the data size; consequently, so does the time complexity. For instance, the TSP has a time complexity of $O(n!)$. Although finding the solution for low values of n is manageable, this is not the case for most scenarios. Hence heuristic approaches are used which have a better time complexity even though the solution may not be optimal.

2.3.2 Accuracy

Accuracy is the second parameter investigated in this paper as it is used to validate and assess the performance of a specific algorithm. It is defined as an evaluation metric that measures the closeness of the obtained values to the accepted or correct value. For this

[35] Prado, K.S. do (2020) *Understanding time complexity with python examples*, *Medium*. Available at: https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7 (Accessed: 26 June 2023).

experiment, the accuracy states the error between the optimal path generated by an algorithm and the actual, shortest path available. As mentioned previously, exact algorithms, including the Brute Force Algorithm, always find the shortest route for the TSP despite being extremely inefficient. Hence, the percentage accuracy of the Elitist GA and ACS will be found using Equation (5).

$$percentage\ accuracy = \frac{shortest\ path\ generated\ by\ control\ brute\ force\ algorithm}{shortest\ path\ generated\ by\ the\ elitist\ GA\ or\ ACS} \times 100 \qquad (5)$$

2.4 Hypothesis

Although the two algorithms fall under the same combinatorial optimisation category (as they are both population-based, metaheuristic and stochastic), this paper hypothesises that the elitist GA will have a lower time complexity compared to ACS. This is because the additional parameter, elitism, ensures that the best chromosomes are always transferred to the next generation, so the optimal solution should be obtained faster. However, ACS should be more accurate since there are several parameters that can be adjusted to acquire the desired convergence speed – this includes the desirability and the constants α and β. Therefore, the hypothesis for this investigation is: ***When using the elitist GA and ACS to find the shortest tour for the TSP, the Elitist GA should have a lower time complexity whilst the solution generated by ACS should have a higher level of accuracy.***

3. Experimentation

3.1 Methodology

3.1.1 Variables

The two independent variables for this investigation are the algorithms used, Elitist GA or ACS, and the data set size. As evident in Appendix 4, the number of cities to visit increases by 4 each time up until 20 as this provides a variety of results that can later be interpreted.

As stated in the research question both the accuracy and efficiency of each algorithm will be measured. Hence, the two dependent variables are the shortest distance calculated and the time taken, in seconds. This enables the accuracy of each algorithm to be calculated using Equation (5) as well as the time complexities to be compared.

One key control variable is the type of control algorithm used since it will produce the shortest distance of each city tour which will then be compared to those calculated by ACS and GA . This way the accuracy can be calculated. Hence, it was decided that the Brute Force algorithm will act as the control and the Python code to run this will be obtained from https://github.com/Joseph bakulikira/Traveling-Salesman-Algorithm.

Table 1 discusses all the control variables for this experiment.

Controlled Variable	Why is it controlled?	How is it controlled?
Computer and Operating System	The hardware can impact the speed of each algorithm, depending on factors such as processor speed and memory. Therefore, these elements must be kept constant to ensure repeatability.	The same hardware was used throughout the experiment: - Computer model: 21.5-inch 2019 iMac - Processor: 3GHz 6-Core Intel Core i5 - Memory: 16GB 2667 MHz DDR4 - macOS: Ventura 13.4.1 All applications will be closed when running code to keep the amount of RAM available constant.
Integrated Development Environment (IDE)	The specifications and differing features of an IDE can have an effect on the results obtained.	The same IDE was used: - Type: Visual Studio Code - Version: 1.80.1
Data points for each data set	If different points are used, the shortest distance will be different so the accuracy of each algorithm cannot be calculated.	The points used were chosen in the preliminary experiment and used throughout the experiment (see Appendix 4)
Parameters for each algorithm	The specific parameters of each algorithm have effects on factors like termination criteria, convergence speed and the exploration of space.	The mutation rate for the Elitist GA was 0.01%. The parameters for ACS were fixed: - α: 1 - β: 3
Initial population size	The population size influences the convergence speed and so must be controlled.	The population size for both algorithms was set at 150.
Python code for ACS, GA, and Brute Force	The same code must be used for each algorithm to ensure reproducibility and consistency in the results.	All algorithms were taken from: https://github.com/Josephbakulikira/Traveling-Salesman-Algorithm (Accessed on 21st July 2023).

Table 1: Control Variables

3.1.2 Experiment

The following steps were used to obtain a set of results:

1. Run the Brute Force Algorithm with 4 data points and record the shortest distance calculated (see Appendix 2).
2. Run the elitist GA using the same 4 data points and record the shortest distance calculated and the time taken (see Appendix 3).
3. Repeat step 2 using ACS (see Appendix 4).
4. Repeat steps 2 and 3 two more times to collect repeat readings.
5. Repeat all steps using a larger data set (see Appendix 5).
6. Calculate the mean time taken for each set of results.

The default_timer() function was imported from the timeit module in Python to obtain the time taken for the optimal solution to be calculated as seen in Figure 7.

```
from timeit import default_timer as timer
```

Figure 7: Code to import default_timer()

After program execution, the time and shortest distance were outputted – see Figures 8 and 9 respectively.

Figure 8: Example time output

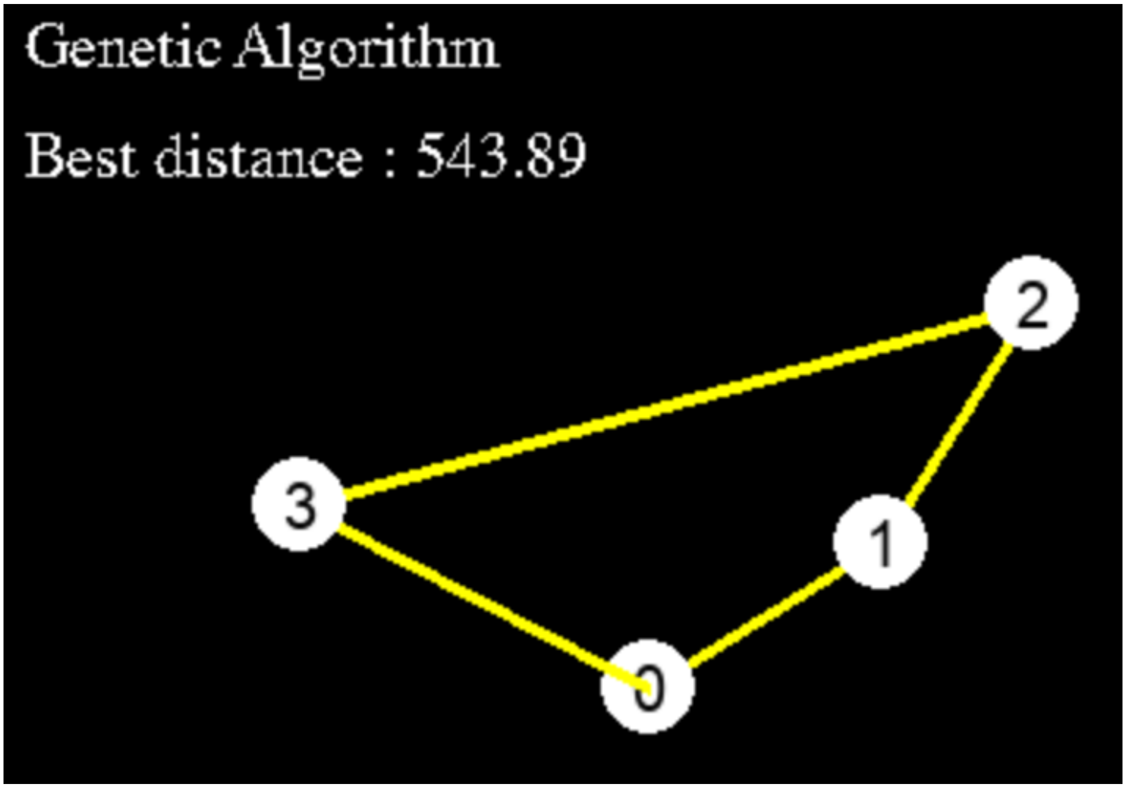


Figure 9: Example best distance output

3.2 Results

Measured using the timer function in Figure 7, the results in Tables 2 and 3 show the time taken to calculate the shortest distance for each algorithm and data set. The mean time taken was found using Equation (6):

$$mean\ time = \frac{time\ 1 + time\ 2 + time\ 3}{3} \tag{6}$$

An example calculation using Equation (6) is shown below.

$$\frac{0.233+0.214+0.196}{3} = 0.214\dot{3}$$

$$= 0.214\ (3\ s.f)$$

Number of Cities	Time taken to find the shortest distance (s)			Mean time (s)
	Attempt 1	Attempt 2	Attempt 3	
4	0.233	0.214	0.196	0.214
8	1.554	1.219	0.805	1.193
12	12.02	11.68	12.76	12.15
16	21.43	19.33	22.46	21.07
20	29.04	30.21	28.65	29.30

Table 2: Time taken to find shortest distance using the Elitist GA

Number of Cities	Time taken to find the shortest distance (s)			Mean time (s)
	Attempt 1	Attempt 2	Attempt 3	
4	0.246	0.178	0.197	0.207
8	0.565	0.689	0.713	0.656
12	4.595	5.103	4.992	4.897
16	7.456	7.685	8.103	7.748
20	12.31	13.85	13.59	13.25

Table 3: Time taken to find shortest distance using ACS

The results from Tables 2 and 3 were plotted to produce Figure 10.

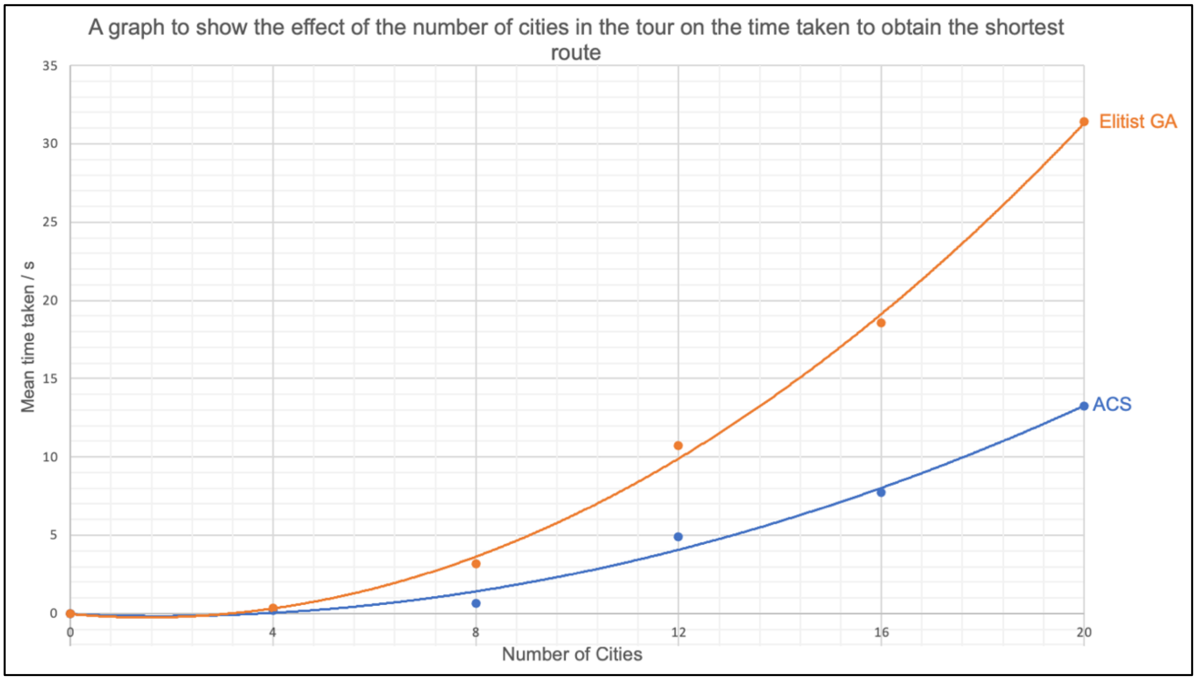


Figure 10: A graph to show mean time taken (s) against number of cities

To evaluate the accuracy of each algorithm, the shortest distance obtained by each of the algorithms was noted down in Tables 4 and 5.

Number of Cities	Shortest distance obtained			Mean shortest distance
	Attempt 1	Attempt 2	Attempt 3	
4	543.890	543.890	543.890	543.890
8	1121.46	1121.46	1121.46	1121.46
12	1253.88	1261.41	1247.21	1254.17
16	1531.68	1542.35	1546.54	1540.19
20	2035.21	1986.42	1964.53	1995.39

Table 4: Shortest distance obtained using the Elitist GA

Number of Cities	Shortest distance obtained			Mean shortest distance
	Attempt 1	Attempt 2	Attempt 3	
4	543.890	543.890	543.890	543.890
8	1121.46	1121.46	1121.46	1121.46
12	1242.87	1239.34	1242.87	1241.69
16	1515.34	1513.43	1511.76	1513.51
20	1687.48	1693.78	1701.54	1694.27

Table 5: Shortest distance obtained using ACS

The actual shortest distance was found using the control Brute Force Algorithm (see Appendix 2).

Number of Cities	Actual shortest distance
4	543.890
8	1121.46
12	1237.83
16	1493.76
20	1632.06

Table 6: Actual shortest distance obtained using Brute Force

The data in Tables 4-6 were processed using Equation (5), enabling the percentage accuracy of each algorithm to be found (see Table 7). The example calculation shows how the percentage accuracy of ACS was determined for 16 cities:

$$\frac{1493.76}{1513.51} \times 100 = 98.69508626$$

$$= 98.70\% \ (4\ s.f)$$

Number of Cities	Elitist GA Percentage Accuracy (%)	ACS Percentage Accuracy (%)
4	100.0	100.0
8	100.0	100.0
12	98.70	99.69
16	96.99	98.70
20	81.79	96.33

Table 7: Percentage accuracy calculated using Equation (6)

The data from Table 7 were plotted to produce Figure 11.

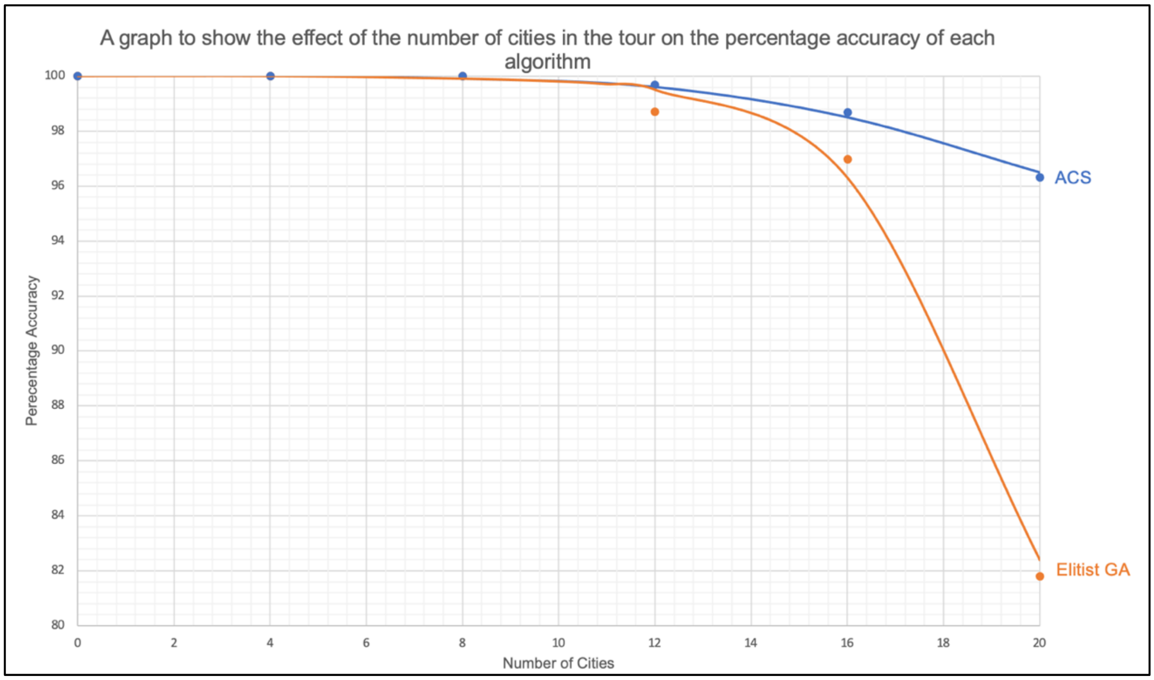


Figure 11: A graph to show percentage accuracy against number of cities

3.3 Interpretation

Figure 10 shows that for both algorithms, the time taken to find the optimal route increased exponentially as the size of the city tour increased. However, the time taken by the Elitist GA increased at a faster rate than ACS suggesting it generally has a greater time complexity. When the algorithms navigated through a map of 4 cities or less, both algorithms had a similar time complexity as it roughly took them a mean time of 0.2 seconds for a solution to be generated.

There were no distinct anomalous results in Tables 2 and 3 since the points in Figure 10 were all located close to the line of best fit. Extrapolation was used to create a curve between 0 and 4 cities for both algorithms, therefore, there was a chance that these points could be unreliable. Nevertheless, this was most likely not the case since background theory supports the claim that the time complexity will increase with city tour size.

Figure 11 clearly illustrates that as the number of cities in the tour increased, the accuracy of the distance generated decreased. Both algorithms are proven to be 100% accurate for tours that consisted of 8 cities or less; if the city tour was greater, the accuracy decreased at a non-linear rate. When experimenting with 20 cities, the Elitist GA had an accuracy rate of 81.79% which is significantly lower than the accuracy of ACS which was 96.33%. Overall, the Elitist GA was shown to be more inaccurate in comparison to ACS as the percentage accuracy fell by a faster rate, especially when the city tour was greater than 16.

It is also evident that there were no anomalous results in Tables 4 and 5 as the points in Figure 11 only deviated slightly from those on the line of best fit, hence, the results were valid and reliable overall. However, the values obtained from the Elitist GA are located further away from its line of best fit in contrast to ACS. This implies that the results from Table 4 had a greater error uncertainty than those in Table 5. Despite this, the error uncertainty did not have an effect on the overall relationship between the accuracy of each algorithm and the number of cities.

4. Conclusion

4.1 Research Question Analysis

This exploration aimed to utilise the theory behind metaheuristic algorithms to formulate an experiment, applicable to evaluate each algorithm's time complexity and percentage accuracy when solving an NP-hard problem. This investigation showed that ACS has a lower time complexity but a higher level of accuracy in comparison to the Elitist GA when finding the shortest route to solve the TSP.

It was proven that accuracy is inversely proportional to the city tour size since larger data sets lead to a lower percentage accuracy. On the other hand, the time taken for either algorithm to obtain a solution increased with the number of cities visited in the tour. A greater time complexity represents a lower level of efficiency; therefore, reinforcing the inversely proportional nature between efficiency and data set size.

4.2 Hypothesis Analysis

The hypothesis made was supported by the results to a partial extent as ACS was shown to be more accurate, however, the Elitist GA actually has a higher time complexity than ACS. After further research, relevant supporting evidence was found. After every iteration, the entire population (city tour) changes with ACS whereas with the Elitist GA, only one city is altered enabling the optimal solution to be found quicker. [36]

This paper's results are reproducible as a similar experiment, carried out by Alexander, A. and Sriwindono, H. had the same findings.[37] As shown in Figure 12, the distance obtained by ACS is always much shorter than that found by the GA, implying that ACS is more accurate. The accuracy of the GA tended to decrease at a faster rate than ACS like in this paper.

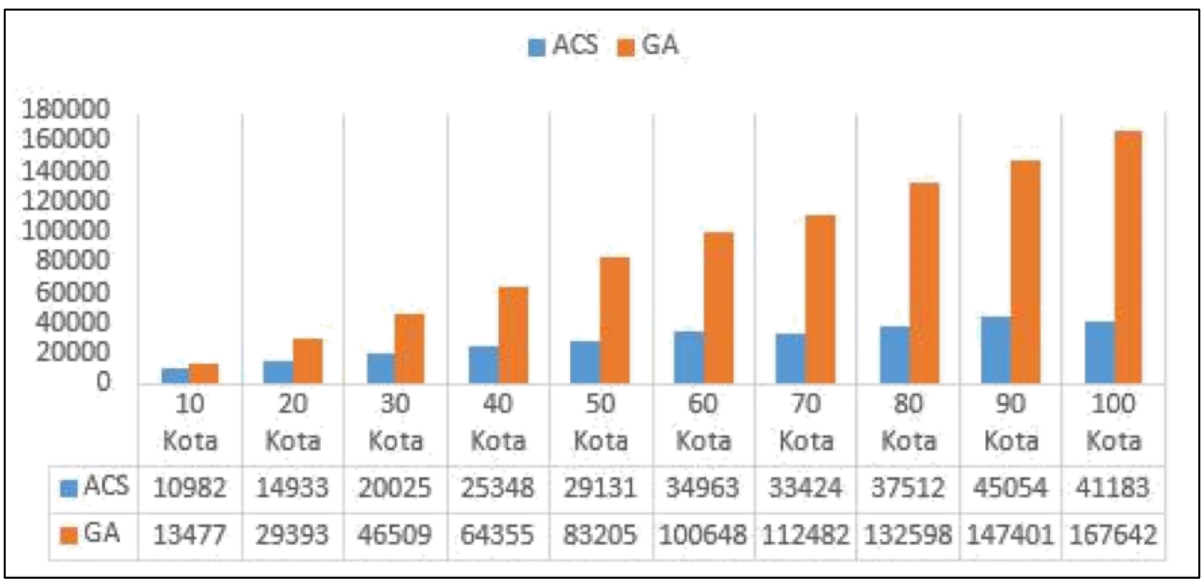


Figure 12: Distance found by each of the two algorithms with shortest distance on the y-axis and city tour size on the x-axis[38]

[36] Alexander, A. and Sriwindono, H. (2019) 'The comparison of genetic algorithm and ant colony optimization in completing travelling salesman problem', *Proceedings of the 2nd International Conference of Science and Technology for the Internet of Things, ICSTI 2019, September 20th, Yogyakarta, Indonesia* [Preprint]. doi:10.4108/eai.20-9-2019.2292121.

[37] *Ibid.*

[38] *Ibid.*

Alexander, A. and Sriwindono, H also recorded the time taken by each algorithm – this is shown in Figure 13.

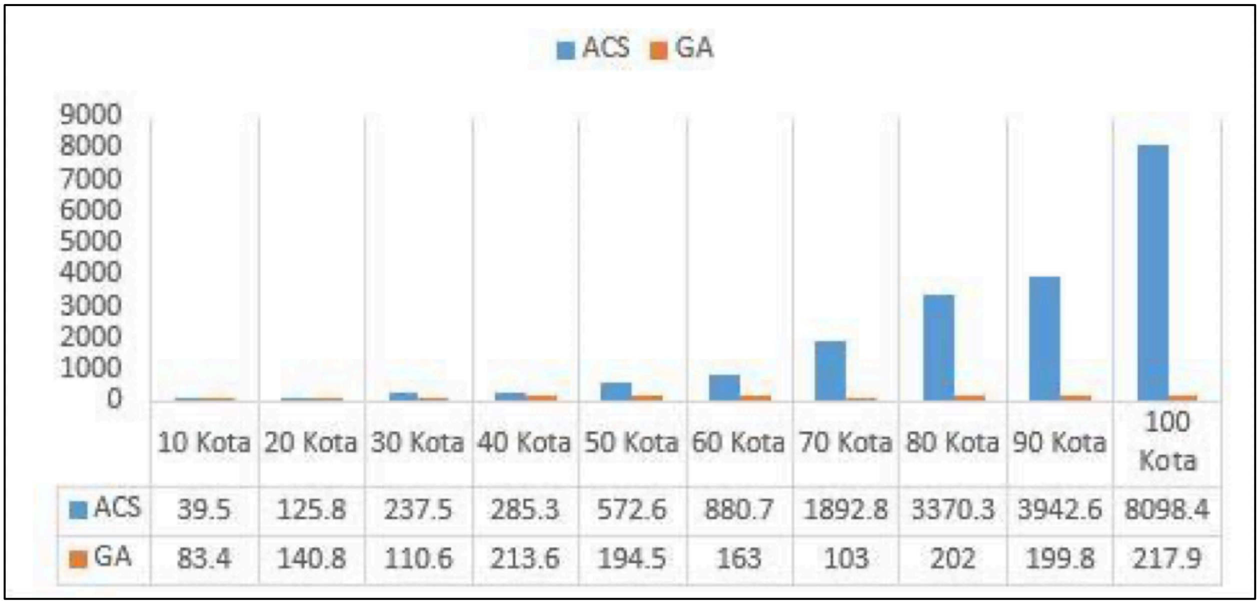


Figure 13: Average time taken by each of the two algorithms with time, in seconds, on the y-axis and city tour size on the x-axis [39]

Their data supported the concept that ACS is faster than the GA when navigating through a city tour consisting of 20 or fewer. However, when the city tour size exceeds 20, the GA was faster than ACS. Therefore, to improve this investigation, data sets larger than 20 should be used to see whether ACS will have a greater time complexity than GA.

[39] Alexander, A. and Sriwindono, H. (2019), *op. cit.*

4.3 Relevance of Data

Both algorithms are powerful metaheuristic methods of solving NP-hard problems, so comparing, and evaluating the algorithms are beneficial to see which one would outperform the other. This experiment proved that ACS is superior to the Elitist GA when solving the TSP, a notable optimisation problem. Based on the results and analysis presented in this paper, for cities consisting of 20 or fewer, ACS always produced a shorter distance in a quicker time period.

These metaheuristic approaches are applicable beyond the TSP as they can be used to solve other NP-hard problems including the Knapsack Problem, Job Shop Scheduling Problem and Vehicle Routing Problem. They can approximate a solution more efficiently than exact algorithms; hence, this data is insightful.

Outside of computational complexity theory, these algorithms have real-life applications, for instance, in telecommunications network optimisation and circuit board manufacturing. In telecommunications, metaheuristic algorithms are used to determine how data packets should be routed to improve network performance and reduce latency. On the other hand, when manufacturing circuitry, the optimal component arrangement is vital to eliminate signal interference. For these reasons, this data is relevant as it can be applied to these scenarios to save resources, such as money, and increase efficiency.

4.4 Evaluation

Seeing that the findings of this experiment aligned with the results of other research papers (see Section 4.2), this essay was proven to be successful and had many strengths. The accuracy of the results was ensured through repeat readings and calculating a mean. In addition, the absence of anomalies supports emphasises the validity of the results. Despite this, several improvements would eliminate potential sources of error since the investigation was carried out in a home environment – these are presented in Table 8.

Source of Errors	Effect and Importance	Improvements
Brute Force Algorithm: For large data sets, the time taken to generate a solution was extremely slow since it has a time complexity of $O(n!)$.	The program may have stopped too soon, meaning the final solution generated was inaccurate. As this value was used to calculate the percentage accuracy, this would have had an effect on the values in Table 15.	Other exact algorithms with a lower time complexity could be used: - Brand and Bound: $O(2^n)$ - Dynamic Programming: $O(n^2 \times 2^n)$
Random Time Error: The programs' performance can be affected by other computer processes running simultaneously.	Although all other applications were closed during program execution, background processes are still being carried out. This would affect the accuracy of the default_timer(), impacting the results in Tables 2 and 3.	A computer with a better processor and larger RAM could eliminate this source of error.
Systematic Error from Source Code: The program code was found online so there may be logic errors.	This error would have an effect on most of the results obtained if the code does not directly mirror the algorithms.	One could program their own algorithms or use code that they are certain to contain no logic errors.

Table 8: Potential errors and corresponding improvements

After evaluating the data acquired in this investigation, it has been concluded that ACS is both more accurate and has a lower time complexity than the Elitist GA when it comes to solving the TSP for 20 cities or fewer. Therefore, the research question ***"How does the Elitist Genetic Algorithm compare to Ant Colony System in terms of time complexity and accuracy when attempting to solve the Travelling Salesman Problem?"*** has been answered, consequently underscoring the beneficial nature and indispensability of this essay.

5. Bibliography

5.1 Books

[1] Ahmed, Z.E. *et al.* (2020) ‘Energy optimization in low-power wide area networks by using heuristic techniques’, *LPWAN Technologies for IoT and M2M Applications*, pp. 199–223. doi:10.1016/b978-0-12-818880-4.00011-9.

[2] Chatting, M. (2018) 'A Comparison of Exact and Heuristic Algorithms to Solve the Travelling Salesman Problem', The Plymouth Student Scientist, 11(2), p. 53-91.

[3] Deep, Kusum., & Mebrahtu, Hadush. (2012), "Variant of partially mapped crossover for the Travelling Salesman problems." International Journal of Combinatorial Optimization Problems and Informatics, Vol.3, num.1, pp.47-69. ISSN: 2007-1558

[4] Dorigo, M. and Gambardella, L.M. (1997) ‘Ant Colony System: A cooperative learning approach to the traveling salesman problem’, *IEEE Transactions on Evolutionary Computation*, 1(1), pp. 53–66. doi:10.1109/4235.585892.

[5] Dorigo, M. and Stützle, T. (2004) ‘The Ant Colony Optimization Metaheuristic’, in *Ant colony optimization*. Cambridge, MA: MIT Press, pp. 25–26.

[6] Dorigo, M. and Stützle, T. (2004) ‘Ant Colony Optimization Algorithms for the Traveling Salesman Problem’, in *Ant colony optimization*. Cambridge, MA: MIT Press, pp. 67–68.

[7] Lenstra, J.K. and Kan, A.H. (1975) ‘Some simple applications of the travelling salesman problem’, *Operational Research Quarterly (1970-1977)*, 26(4), pp. 717–733. doi:10.2307/3008306.

[8] M. Almufti, S., Boya Marqas, R. and Ashqi Saeed, V. (2019) ‘Taxonomy of bio-inspired optimization algorithms’, *Journal of Advanced Computer Science & Technology*, 8(2), p. 23. doi:10.14419/jacst.v8i2.29402.

[9] Nguyen, K.-H. and Ock, C.-Y. (2011) ‘Word sense disambiguation as a traveling salesman problem’, *Artificial Intelligence Review*, 40(4), pp. 405–427. doi:10.1007/s10462-011-9288-9.

[10] Singh, V.K. and Sharma, V. (2014) ‘Elitist genetic algorithm based energy balanced routing strategy to prolong lifetime of wireless sensor networks’, *Chinese Journal of Engineering*, 2014, pp. 1–6. doi:10.1155/2014/437625.

5.2 Research Papers

[1] Alexander, A. and Sriwindono, H. (2019) 'The comparison of genetic algorithm and ant colony optimization in completing travelling salesman problem', *Proceedings of the 2nd International Conference of Science and Technology for the Internet of Things, ICSTI 2019, September 20th, Yogyakarta, Indonesia* [Preprint]. doi:10.4108/eai.20-9-2019.2292121.

[2] Chase, C. *et al.* (no date) *An Evaluation of the Traveling Salesman Problem*. Available at: https://scholarworks.calstate.edu/downloads/xg94hr81q#:~:text=. (Accessed: 14 July 2023).

[3] Danu, M.S. (2013) *Ant colony optimization algorithms*, *Scribd*. Available at: https://www.scribd.com/document/136679005/Ant-colony-optimization-algorithms# (Accessed: 27 June 2023).

[4] Hasançebi, O. and Erbatur, F. (2000) 'Evaluation of crossover techniques in genetic algorithm based optimum structural design', *Computers & Structures*, 78(1–3), pp. 435–448. doi:10.1016/s0045-7949(00)00089-4.

[5] Morris, A.T. (1998) *Optimization of the Traveling Salesman Problem and Multivariate Real-Valued Functions using a Genetic Algorithm*. dissertation.

[6] Mulani, M. and Desai, V.L. (2018) 'Design and Implementation Issues in Ant Colony Optimization', in *International Journal of Applied Engineering Research*. 16th edn. Research India Publications, pp. 12877–12882.

[7] *NP-hard problems and approximation algorithms - University of Texas at ...* Available at: https://personal.utdallas.edu/~dxd056000/cs6363/unit5.pdf (Accessed: 17 May 2023).

[8] Ranjith, K.A. (2010) *Ant Colony Optimization*. rep., pp. 1–16.

[9] Üçoluk, G. (2002) 'Genetic algorithm solution of the TSP avoiding special crossover and mutation', *Intelligent Automation & Soft Computing*, 8(3), pp. 265–272. doi:10.1080/10798587.2000.10642829.

5.3 Websites

[1] *Big O notation - mit - massachusetts institute of technology* (no date) *Big O Notation*. Available at: https://web.mit.edu/16.070/www/lecture/big_o.pdf (Accessed: 26 June 2023).

[2] Black, P.E. (2020) *Hamiltonian cycle*, *Dictionary of Algorithms and Data Structures* [online] Available at: https://xlinux.nist.gov/dads/HTML/hamiltonianCycle.html (Accessed: 17 May 2023).

[3] *Genetic algorithms* (2017) *Scribd*. Available at: https://www.scribd.com/document/351623322/Genetic-Algorithms# (Accessed: 15 June 2023).

[4] Prado, K.S. do (2020) *Understanding time complexity with python examples*, *Medium*. Available at: https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7 (Accessed: 26 June 2023).

6. Appendix

All the code was accessed on 21st July 2023 from https://github.com/Josephbakulikira/Traveling-Salesman-Algorithm. See the following pages for screenshots of the code used.

6.1 Appendix 1 – Main Code

A > main.py > ...

```python
import pygame
from point import *
from manager import *
from random import randint
from UI.setup import *
from utils import SumDistance
from timeit import default_timer as timer

pygame.init()

manager = Manager()
antColonyTypes = ["ACS", "ELITIST", "MAX-MIN"]
#manager.ChangeAntColonyVariation("ELITIST")

selectedIndex = 2

pause = True
started = False
rightMouseClicked = False
GenerateToggle = False
reset = False

PauseButton.state = pause
ResetButton.state = reset
RandomButton.state = GenerateToggle

showUI = False
run = True
while run:
    manager.Background()

    delta_time = manager.SetFps()
    manager.UpdateCaption()
    # handle Events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            run = False
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                run = False
                end = timer()
                print('Time in seconds:',end - start)
            if event.key == pygame.K_SPACE:
                pause = not pause
                started = True
                start = timer()
            if event.key == pygame.K_RETURN:
                showUI = not showUI

        if event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:
                rightMouseClicked = True
    # Choose one method between the 3 below: bruteForce, lexicagraphic order, genetic algorithm
    if selectedIndex == 0:
        if pause == False:
            manager.BruteForce()
        manager.DrawPoints()
        manager.DrawShortestPath()
        #manager.Percentage(manager.PossibleCombinations)
```

```python
    elif selectedIndex == 1:
        if pause == False:
            manager.Lexicographic()
        manager.DrawPoints()
        manager.DrawShortestPath()
        manager.Percentage(manager.PossibleCombinations)
    elif selectedIndex == 2:
        if pause == False:
            manager.GeneticAlgorithm()
        manager.DrawPoints()
        manager.DrawShortestPath()
    else:
        manager.AntColonyOptimization(pause)
        # print(selectedIndex-3)
        manager.ChangeAntColonyVariation(antColonyTypes[selectedIndex-3])
        manager.Percentage(iterations)
    manager.ShowText(selectedIndex, started)
    # UI
    if showUI:
        panel.Render(manager.screen)
        AlgorithmChoice.Render(manager.screen, rightMouseClicked)
        if pause != PauseButton.state:
            PauseButton.state = pause

        PauseButton.Render(manager.screen, rightMouseClicked)
        ResetButton.Render(manager.screen, rightMouseClicked)
        RandomButton.Render(manager.screen, rightMouseClicked)

        pause = PauseButton.state
        reset = ResetButton.state

        if reset == True:
            reset = False
            ResetButton.state = False
            temp = manager.Points.copy()
            manager = Manager(temp)
            manager.OptimalRoutes = manager.Points.copy()
            manager.recordDistance = SumDistance(manager.Points)
            manager.ResetAntColony(manager.antColony.variation)
            manager.ResetGenetic()

        GenerateToggle = RandomButton.state
        if GenerateToggle == True:
            manager.RandomPoints()
            GenerateToggle = False
            RandomButton.state = False
        if pause == True:
            PauseButton.text = "Continue"
        else:
            PauseButton.text = "Pause"
        if rightMouseClicked:
            selectedIndex = AlgorithmChoice.currentIndex
    # point scale animation increment
    manager.scaler += 1
    if manager.scaler > manager.max_radius:
        manager.scaler = manager.max_radius
    pygame.display.flip()
    rightMouseClicked = False
pygame.quit()
```

6.2 Appendix 2 – Brute Force Algorithm Code

```
A > manager.py > Manager > ResetAntColony
import pygame
import random
from random import randint, sample
from point import Point
from utils import *
from genetic import Genetic
from antColony import *
from ant import *

offset          = 100
width, height   = 1920, 1080
populationSize  = 150
n = 10
colony_size = 150
iterations = 300
pygame.font.init()

class Manager(object):
    size            = (width, height)
    fps             = 30
    screen          = pygame.display.set_mode(size)
    clock           = pygame.time.Clock()
    scaler          = 1
    max_radius      = 15
    Black           = (0, 0, 0)
    White           = (255, 255, 255)
    Yellow          = (255, 255, 0)
    Gray            = (100, 100, 100)
    Highlight       = (255, 255, 0)
    LineThickness   = 4
    showIndex       = True
    n_points        = n
    algorithms        = ["Brute Force", "Lexicographic Order", "Genetic Algorithm", "Ant Colony ACS", "Ant Colony Elitist", "Ant Colony Max-Min"]

    genetic         = Genetic([sample(list(range(n)), n) for i in range(populationSize)], populationSize)

    PossibleCombinations = Factorial(n_points)
    print("possible combinations : {}".format(Factorial(n_points)))

    Order           = [i for i in range(n_points)]
    counter         = 0

    def __init__(self):
        Points = [Point(372, 201), Point(298, 247), Point(420, 125), Point(187, 189), Point(458, 325), Point(398, 289), Point(198, 456), Point(346, 150), Point(162, 157), Point(201, 317)]
        self.Points           = Points
        self.recordDistance  = SumDistance(self.Points)
        self.OptimalRoutes   = self.Points.copy()
        self.currentList     = self.Points.copy()

        # --- Ant Colony ---
        self.antColony = AntColony(variation="ACS", size=colony_size, max_iterations = iterations,
                                   nodes=self.Points.copy(), alpha=1, beta=3, rho=0.1, pheromone=1, phe_deposit_weight=1)

    def ResetGenetic(self):
        self.genetic = Genetic([sample(list(range(n)), n) for i in range(populationSize)], populationSize)

    def ChangeAntColonyVariation(self, name):
        self.antColony.variation = name
```

```
    def ResetAntColony(self, name="ACS"):
        self.recordDistance  = SumDistance(self.Points)
        self.antColony = AntColony(variation=name, size=colony_size, max_iterations = iterations,
                                   nodes=self.Points.copy(), alpha=1, beta=3, rho=0.1, pheromone=1, phe_deposit_weight=1)
    def SetFps(self):
        return self.clock.tick(self.fps)/1000.0

    def UpdateCaption(self):
        frameRate = int(self.clock.get_fps())
        pygame.display.set_caption("Traveling Salesman Problem - Fps : {}".format(frameRate))

    def Counter(self):
        self.counter += 1
        if self.counter > self.PossibleCombinations:
            self.counter = self.PossibleCombinations

    def BruteForce(self):
        if self.counter != self.PossibleCombinations:
            i1 = randint(0, self.n_points-1)
            i2 = randint(0, self.n_points-1)
            self.Points[i1], self.Points[i2] = self.Points[i2], self.Points[i1]
```

```
            dist = SumDistance(self.Points)
            if dist < self.recordDistance:
                self.recordDistance  = dist
                self.OptimalRoutes   = self.Points.copy()
                #print("Shortest distance : {}" .format(self.recordDistance))

            self.DrawLines()
        def Lexicographic(self):
            self.Order = LexicalOrder(self.Order)
            nodes = []
            for i in self.Order:
                nodes.append(self.Points[i])

            self.Counter()

            dist = SumDistance(nodes)
            if dist < self.recordDistance:
                self.recordDistance = dist
                self.OptimalRoutes  = nodes.copy()
                #print("Shortest distance : {}" .format(self.recordDistance))
            self.DrawLines()

        def GeneticAlgorithm(self):
            self.genetic.CalculateFitness(self.Points)
            self.genetic.NaturalSelection()

            # self.Counter()
            for i in range(self.n_points):
                self.currentList[i] = self.Points[self.genetic.current[i]]
            if self.genetic.record < self.recordDistance:
                for i in range(self.n_points):
                    self.OptimalRoutes[i] = self.Points[self.genetic.fitest[i]]
                self.recordDistance = self.genetic.record
```

```
            self.DrawLines(True)

        def AntColonyOptimization(self, pause):
            if pause == False:
                self.counter += 1
                if self.counter > self.antColony.max_iterations:
                    self.counter = self.antColony.max_iterations

                if self.counter < self.antColony.max_iterations:
                    self.antColony.Simulate(self.counter)

            self.antColony.Draw(self)
            self.recordDistance = self.antColony.best_distance

        def RandomPoints(self):
            self.Points = [Point(167,92), Point(73,5), Point(56,24), Point(6,41)]
            self.ResetAntColony(self.antColony.variation)
            self.recordDistance  = SumDistance(self.Points)
            self.OptimalRoutes   = self.Points.copy()
            self.currentList     = self.Points.copy()

        def Percentage(self, val):
            percent = (self.counter/val) * 100
            textColor   = (255, 255, 255)
            # textFont    = pg.font.Font("freesansbold.ttf", size)
            textFont    = pygame.font.SysFont("Arial", 20)
            textSurface = textFont.render(str(round(percent, 4)), False, textColor)
            self.screen.blit(textSurface, (width//2, 50))
```

```python
    def ShowText(self, selectedIndex, started = True):
        textColor   = (255, 255, 255)
        # textFont    = pg.font.Font("freesansbold.ttf", size)
        textFont    = pygame.font.SysFont("Times", 20)
        textFont2    = pygame.font.SysFont("Arial Black", 40)

        textSurface1 = textFont.render("Best distance : " + str(round(self.recordDistance,2)), False, textColor)
        textSurface2 = textFont.render(self.algorithms[selectedIndex], False, textColor)
        textSurface3 = textFont2.render("... Press ' SPACE ' to start ..." ,False, textColor)

        self.screen.blit(textSurface1, (100, 70))
        self.screen.blit(textSurface2, (100, 35))
        if started == False:
            self.screen.blit(textSurface3, (width//2, height-200))

    def DrawShortestPath(self):
        if len(self.OptimalRoutes) > 0:
            for n in range(self.n_points):
                _i = (n+1)%self.n_points
                pygame.draw.line(self.screen, self.Highlight,
                                (self.OptimalRoutes[n].x, self.OptimalRoutes[n].y),
                                (self.OptimalRoutes[_i].x, self.OptimalRoutes[_i].y),
                                self.LineThickness)
                self.OptimalRoutes[n].Draw(self, self.showIndex, True, n)

    def DrawPoints(self, selected_index = 0):
        for point in self.Points:
            point.radius = self.scaler
            point.Draw(self)
```

```python
    def DrawLines(self, drawCurrent=False):
        if drawCurrent == True:
            for i, point in enumerate(self.currentList):
                _i = (i+1)%len(self.currentList)
                pygame.draw.line(self.screen, self.Gray, (point.x, point.y), (self.currentList[_i].x, self.currentList[_i].y), 1)
        else:
            for i, point in enumerate(self.Points):
                _i = (i+1)%len(self.Points)
                pygame.draw.line(self.screen, self.Gray, (point.x, point.y), (self.Points[_i].x, self.Points[_i].y), 1)

    def Background(self):
        self.screen.fill(self.Black)
```

6.3 Appendix 3 – Elitist Genetic Algorithm Code

A > genetic_algorithm_TSP.py > createRoute

```python
import numpy as np, random, operator, pandas as pd, matplotlib.pyplot as plt

class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, city):
        xDis = abs(self.x - city.x)
        yDis = abs(self.y - city.y)
        distance = np.sqrt((xDis ** 2) + (yDis ** 2))
        return distance

    def __repr__(self):
        return "(" + str(self.x) + "," + str(self.y) + ")"

class Fitness:
    def __init__(self, route):
        self.route = route
        self.distance = 0
        self.fitness= 0.0

    def routeDistance(self):
        if self.distance ==0:
            pathDistance = 0
            for i in range(0, len(self.route)):
                fromCity = self.route[i]
                toCity = None
                if i + 1 < len(self.route):
                    toCity = self.route[i + 1]
                else:
                    toCity = self.route[0]
                pathDistance += fromCity.distance(toCity)
            self.distance = pathDistance
        return self.distance

    def routeFitness(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.routeDistance())
        return self.fitness

def createRoute(cityList):
    route = random.sample(cityList, len(cityList))
    return route

def initialPopulation(popSize, cityList):
    population = []

    for i in range(0, popSize):
        population.append(createRoute(cityList))
    return population

def rankRoutes(population):
    fitnessResults = {}
    for i in range(0,len(population)):
        fitnessResults[i] = Fitness(population[i]).routeFitness()
    return sorted(fitnessResults.items(), key = operator.itemgetter(1), reverse = True)
```

```python
def selection(popRanked, eliteSize):
    selectionResults = []
    df = pd.DataFrame(np.array(popRanked), columns=["Index","Fitness"])
    df['cum_sum'] = df.Fitness.cumsum()
    df['cum_perc'] = 100*df.cum_sum/df.Fitness.sum()

    for i in range(0, eliteSize):
        selectionResults.append(popRanked[i][0])
    for i in range(0, len(popRanked) - eliteSize):
        pick = 100*random.random()
        for i in range(0, len(popRanked)):
            if pick <= df.iat[i,3]:
                selectionResults.append(popRanked[i][0])
                break
    return selectionResults

def matingPool(population, selectionResults):
    matingpool = []
    for i in range(0, len(selectionResults)):
        index = selectionResults[i]
        matingpool.append(population[index])
    return matingpool

def breed(parent1, parent2):
    child = []
    childP1 = []
    childP2 = []

    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))

    startGene = min(geneA, geneB)
    endGene = max(geneA, geneB)

    for i in range(startGene, endGene):
        childP1.append(parent1[i])

    childP2 = [item for item in parent2 if item not in childP1]

    child = childP1 + childP2
    return child

def breedPopulation(matingpool, eliteSize):
    children = []
    length = len(matingpool) - eliteSize
    pool = random.sample(matingpool, len(matingpool))

    for i in range(0,eliteSize):
        children.append(matingpool[i])

    for i in range(0, length):
        child = breed(pool[i], pool[len(matingpool)-i-1])
        children.append(child)
    return children

def mutate(individual, mutationRate):
    for swapped in range(len(individual)):
        if(random.random() < mutationRate):
            swapWith = int(random.random() * len(individual))
```

```python
            city1 = individual[swapped]
            city2 = individual[swapWith]

            individual[swapped] = city2
            individual[swapWith] = city1
    return individual

def mutatePopulation(population, mutationRate):
    mutatedPop = []

    for ind in range(0, len(population)):
        mutatedInd = mutate(population[ind], mutationRate)
        mutatedPop.append(mutatedInd)
    return mutatedPop

def nextGeneration(currentGen, eliteSize, mutationRate):
    popRanked = rankRoutes(currentGen)
    selectionResults = selection(popRanked, eliteSize)
    matingpool = matingPool(currentGen, selectionResults)
    children = breedPopulation(matingpool, eliteSize)
    nextGeneration = mutatePopulation(children, mutationRate)
    return nextGeneration

def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):
    pop = initialPopulation(popSize, population)
    print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))

    for i in range(0, generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)

    print("Final distance: " + str(1 / rankRoutes(pop)[0][1]))
    bestRouteIndex = rankRoutes(pop)[0][0]
    bestRoute = pop[bestRouteIndex]
    return bestRoute

cityList = []

for i in range(0,25):
    cityList.append(City(x=int(random.random() * 200), y=int(random.random() * 200)))

print(cityList)

geneticAlgorithm(population=cityList, popSize=150, eliteSize=20, mutationRate=0.01, generations=500)

def geneticAlgorithmPlot(population, popSize, eliteSize, mutationRate, generations):
    pop = initialPopulation(popSize, population)
    progress = []
    progress.append(1 / rankRoutes(pop)[0][1])

    for i in range(0, generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)
        progress.append(1 / rankRoutes(pop)[0][1])

    plt.plot(progress)
    plt.ylabel('Distance')
    plt.xlabel('Generation')
    plt.show()
geneticAlgorithmPlot(population=cityList, popSize=100, eliteSize=20, mutationRate=0.01, generations=500)
```

```python
from utils import *
from random import randint

class Genetic:
    def __init__(self, population=[], populationSize=0):
        self.population = population
        self.size = populationSize
        self.fitness = [0 for i in range(populationSize)]
        self.record = float("inf")
        self.currentDist = float("inf")
        self.current = None
        self.fitest = []
        self.fitestIndex = 0
        self.mutation_rate = 0.01

    def CalculateFitness(self, points):
        for i in range(self.size):
            nodes = []
            for j in self.population[i]:
                nodes.append(points[j])
            #print(nodes)
            dist = SumDistance(nodes)
            if dist < self.currentDist:
                self.current = self.population[i]

            if dist < self.record :
                self.record = dist
                self.fitest = self.population[i]
                self.fitestIndex = i
                #print(f"Shortest distance: {dist}")
            self.fitness[i] = 1/ (dist+1)
        self.NormalizeFitnesss()

    def NormalizeFitnesss(self):
        s = 0
        for i in range(self.size):
            s += self.fitness[i]
        for i in range(self.size):
            self.fitness[i] = self.fitness[i]/s

    def Mutate(self, genes):
        for i in range(len(self.population[0])):
            if (randint(0, 100)/100) < self.mutation_rate:
                a = randint(0, len(genes)-1)
                b = randint(0, len(genes)-1)
                genes[a], genes[b] = genes[b], genes[a]

    def CrossOver(self, genes1, genes2):
        start = randint(0, len(genes1)-1)
        end   = randint(start-1, len(genes2)-1)
        try:
            end = randint(start+1, len(genes2)-1)
        except:
            pass
        new_genes = genes1[start:end]
        for i in range(len(genes2)):
            p = genes2[i]
            if p not in new_genes:
                new_genes.append(p)
        return new_genes

    def NaturalSelection(self):
        nextPopulation = []
        for i in range(self.size):
            generation1 = PickSelection(self.population, self.fitness)
            generation2 = PickSelection(self.population, self.fitness)
            genes = self.CrossOver(generation1, generation2)
            self.Mutate(genes)
            nextPopulation.append(genes)
        self.population = nextPopulation
```

6.4 Appendix 4 – Ant Colony System Code

A > antColony.py > ...

```python
import pygame
from ant import *
from utils import translateValue
pygame.font.init()
textColor   = (0, 0, 0)
# textFont    = pg.font.Font("freesansbold.ttf", size)
textFont    = pygame.font.SysFont("Arial", 20)

class AntColony(object):
    def __init__(self, variation="ACS", size=5, elitist_weight=1.0, minFactor=0.001, alpha=1.0, beta=3.0,
                 rho=0.1, phe_deposit_weight=1.0, pheromone=1.0, max_iterations=100, nodes=None, labels=None):
        self.variation = variation
        self.size = size
        self.elitist_weight = elitist_weight
        self.minFactor = minFactor
        self.alpha = alpha
        self.rho = rho
        self.phe_deposit_weight = phe_deposit_weight
        self.max_iterations = max_iterations
        self.n_nodes = len(nodes)
        self.nodes = nodes
        self.edges = [[None for j in range(self.n_nodes)] for i in range(self.n_nodes)]
        for x in range(self.n_nodes):
            for y in range(self.n_nodes):
                heuristic = math.sqrt(
                    math.pow(self.nodes[x].x-self.nodes[y].x, 2) +
                    math.pow(self.nodes[x].y-self.nodes[y].y, 2)
                )
                self.edges[x][y] = self.edges[y][x] = Edge(x, y, heuristic, pheromone)
        self.ants = [Ant(self.edges, alpha, beta, self.n_nodes) for i in range(self.size)]

        # global Best route
        self.best_tour = []
        self.best_distance = float("inf")

        self.local_best_route = []
        self.local_best_distance = float("inf")

    def AddPheromone(self, tour, distance, heuristic=1):
        pheromone_to_add = self.phe_deposit_weight / distance
        for i in range(self.n_nodes):
            self.edges[tour[i]][tour[(i + 1) % self.n_nodes]].pheromone += heuristic

    def ACS(self):
        # for step in range(self.max_iterations):
        for ant in self.ants:
            self.AddPheromone(ant.UpdateTour(), ant.CalculateDistance())
            if ant.distance < self.best_distance:
                self.best_tour = ant.tour
                self.best_distance = ant.distance

        for x in range(self.n_nodes):
            for y in range(x + 1, self.n_nodes):
                self.edges[x][y].pheromone *= (1 - self.rho)
```

```
    def Simulate(self, counter):
        if self.variation == "ACS":
            self.ACS()
        elif self.variation == "ELITIST":
            self.ELITIST()
        elif self.variation == "MAX-MIN":
            self.MAX_MIN(counter)

    def Draw(self, manager):
        # Draw Best Routes
        for i in range(len(self.best_tour)):
            a = self.nodes[self.best_tour[i]]
            b = self.nodes[self.best_tour[(i+1) % len(self.best_tour)]]
            pygame.draw.line(manager.screen, manager.Highlight, a.GetTuple(), b.GetTuple(), manager.LineThickness)
        # Draw Pheromones
        if self.variation == "MAX-MIN":
            for ant in self.ants:
                for edge in ant.edges:
                    for e in edge:
                        r = g = b = int(min((e.pheromone)*math.pow(10, 5), 255))
                        thickness = int(translateValue(e.pheromone, 0, 255, 1, 8))
                        pygame.draw.line(manager.screen, (r, g, 0), self.nodes[e.a].GetTuple(), self.nodes[e.b].GetTuple(), thickness)
        else:
            for ant in self.ants:
                for edge in ant.edges:
                    for e in edge:
                        r = g = b = int(min((e.pheromone)*2, 255))
                        thickness = int(translateValue(e.pheromone, 0, 255, 1, 8))
                        pygame.draw.line(manager.screen, (r, g, 0), self.nodes[e.a].GetTuple(), self.nodes[e.b].GetTuple(), thickness)


        for node in self.nodes:
            pygame.draw.circle(manager.screen, manager.White, node.GetTuple(), manager.scaler)

        for i in self.best_tour:
            textSurface = textFont.render(str(i), True, textColor)
            textRectangle = textSurface.get_rect(center=(self.nodes[self.best_tour[i]].x, self.nodes[self.best_tour[i]].y))
            manager.screen.blit(textSurface, textRectangle)
```

```
A > ant.py > ...
import pygame
import math
import random

class Edge:
    def __init__(self, a, b, heuristic, pheromone):
        self.a = a
        self.b = b
        self.heuristic = heuristic
        self.pheromone = pheromone

class Ant:
    def __init__(self, edges, alpha, beta, n_nodes):
        """
        alpha -> parameter used to control the importance of the pheromone trail
        beta  -> parameter used to control the heuristic information during selection
        """
        self.edges = edges
        self.tour = None
        self.alpha = alpha
        self.beta = beta
        self.n_nodes = n_nodes
        self.distance = 0.0
```

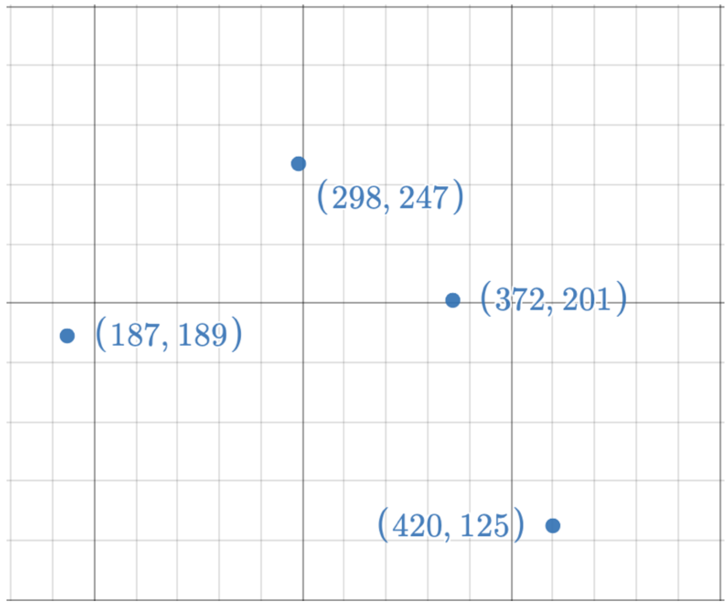
```python
    def NodeSelection(self):
        """
        Constructing solution
        an ant will often follow the strongest
        pheromone trail when constructing a solution.
        state -> is a point on a graph or a City
        Here, an ant would be selecting the next city depending on the distance
        to the next city, and the amount of pheromone on the path between
        the two cities.
        """
        roulette_wheel = 0
        states = [node for node in range(self.n_nodes) if node not in self.tour]
        heuristic_value = 0
        for new_state in states:
            heuristic_value += self.edges[self.tour[-1]][new_state].heuristic
        for new_state in states:
            A = math.pow(self.edges[self.tour[-1]][new_state].pheromone, self.alpha)
            B = math.pow((heuristic_value/self.edges[self.tour[-1]][new_state].heuristic), self.beta)
            roulette_wheel += A * B
        random_value = random.uniform(0, roulette_wheel)
        wheel_position = 0
        for new_state in states:
            A = math.pow(self.edges[self.tour[-1]][new_state].pheromone, self.alpha)
            B = math.pow((heuristic_value/self.edges[self.tour[-1]][new_state].heuristic), self.beta)
            wheel_position += A * B
            if wheel_position >= random_value:
                return new_state

    def UpdateTour(self):
        self.tour = [random.randint(0, self.n_nodes - 1)]
        while len(self.tour) < self.n_nodes:
            self.tour.append(self.NodeSelection())
        return self.tour

```


```python
    def UpdateTour(self):
        self.tour = [random.randint(0, self.n_nodes - 1)]
        while len(self.tour) < self.n_nodes:
            self.tour.append(self.NodeSelection())
        return self.tour
```

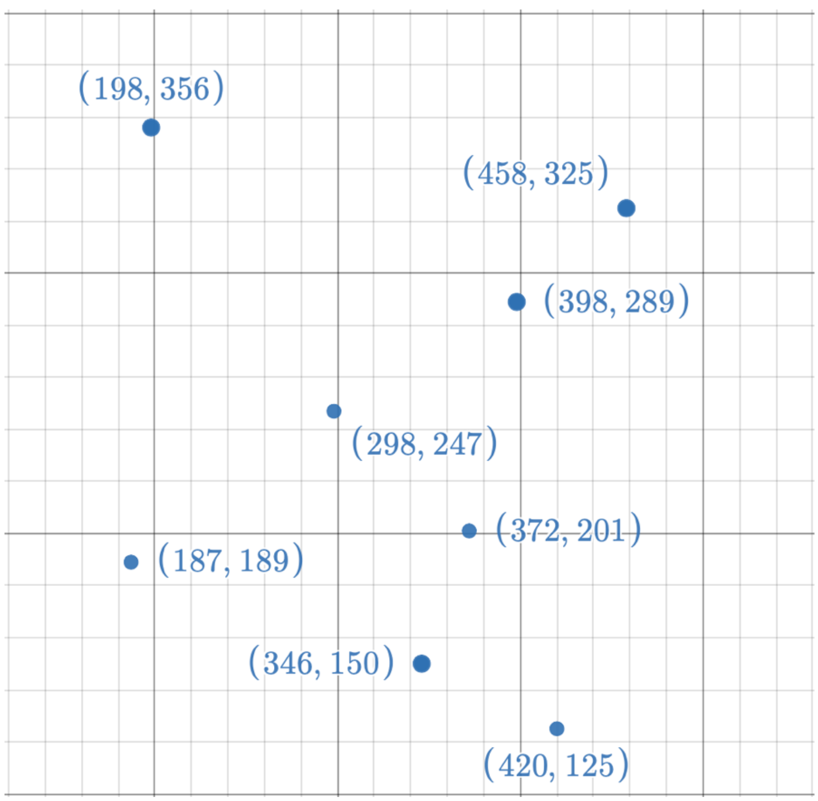

```python
    def CalculateDistance(self):
        self.distance = 0
        for i in range(self.n_nodes):
            self.distance += self.edges[self.tour[i]][self.tour[(i+1)%self.n_nodes]].heuristic
        return self.distance
```

6.5 Appendix 5 – Data Sets

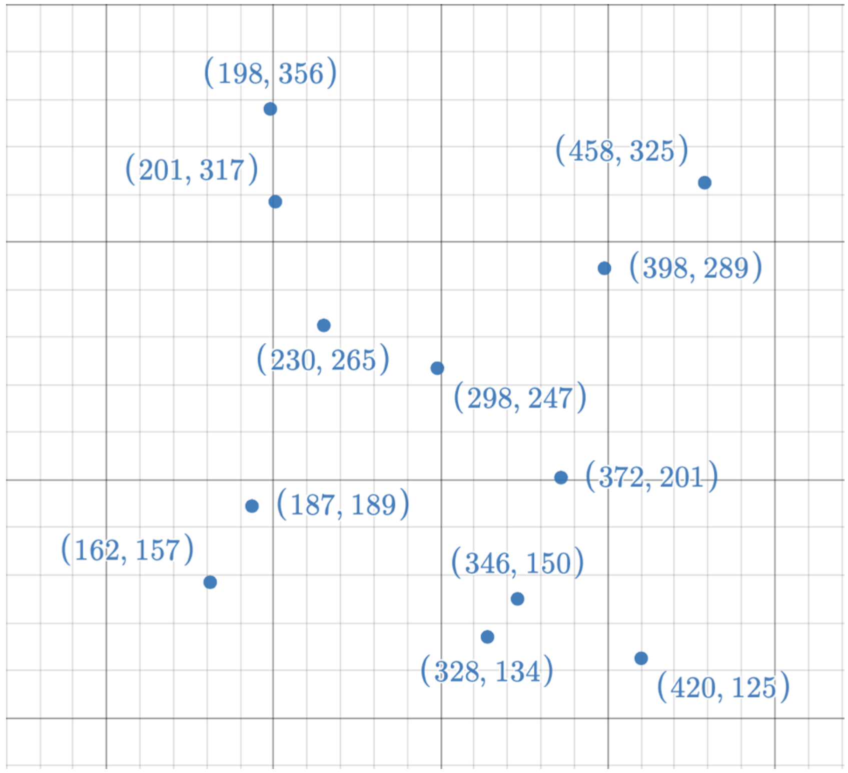
6.5.1 Four City Map



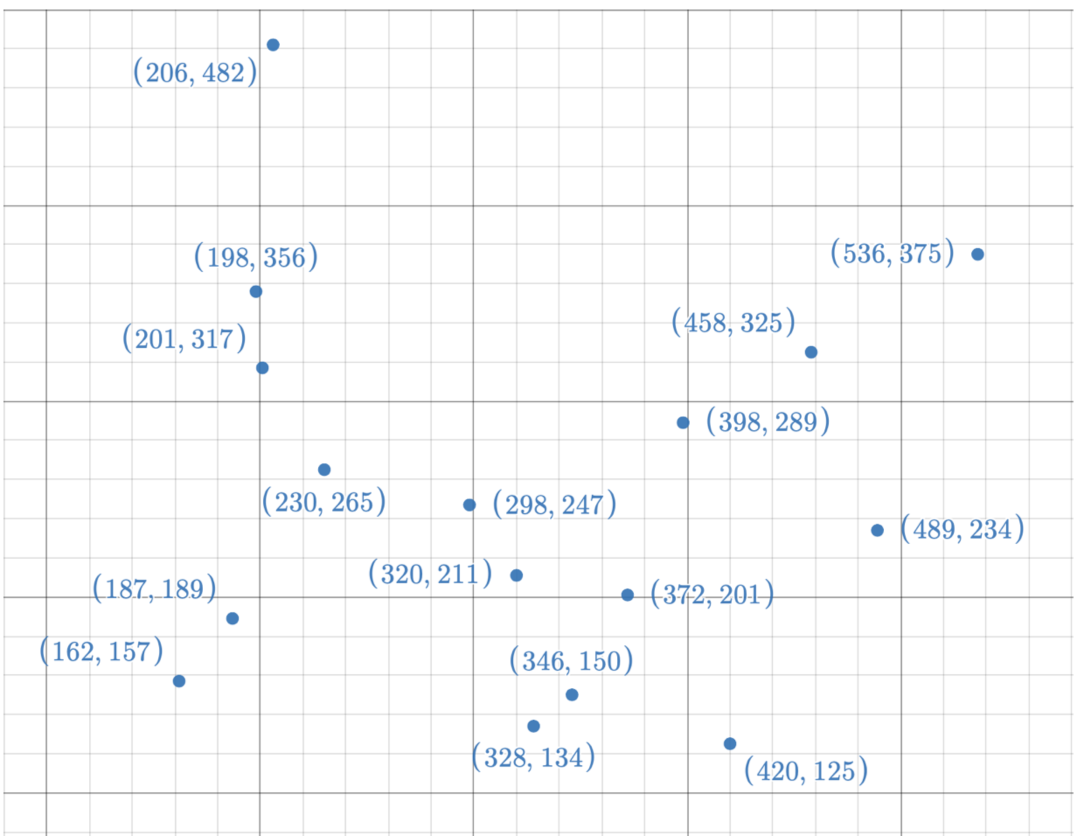
6.5.2 Eight City Map



6.5.3 Twelve City Map



6.5.4 Sixteen City Map



6.5.4 Twenty City Map