EVOLUTIONARY ALGORITHMS APPLIED TO THE SYMMETRIC TSP

Dylan Galea

BSC.MATHEMATICS AND COMPUTER SCIENCE YR3

84296M

Table of Contents

Introduction

The aim of this assignment was to research about Genetic Algorithms and Ant colony optimization algorithms to design and implement one of both, applied to the Symmetric Travel Salesman problem. The Travelling Salesman problem is defined as, given a set of cities and distance between every pair of cities find a tour with minimum weight that visits each city once and that returns to the starting city. TSP is an NP-Hard problem thus there is no polynomial time solution that gives the exact shortest Hamiltonian cycle in the complete graph. In our case since symmetric TSP was considered, the distance between any 2 vertices 'u' and 'v' is the same starting from 'u' or from 'v'. Many solutions were considered to try and approximate the symmetric TSP problem, such as Exact Algorithms, Ant colony optimization algorithms, Genetic Algorithms and many others. However, in this assignment the focus was entirely on Ant colony optimizations and Genetic Algorithms. Genetic Algorithms are called optimization algorithms based on the principles of Genetics and Natural Selection and are very ideal for problems that are required to be estimated and that would take exponential time in order to fully solve. On the other hand, Ant colony optimization algorithms are inspired from the ant's natural behaviour to find the shortest path between its home and a food source. [1], [2], [3]

This report is divided into 3 main sections where the first section is dedicated to the Genetic Algorithm, the second section is dedicated to the Ant Colony Optimization Algorithm and the last section is a comparison and an evaluation of both the genetic algorithm and the ant colony optimization algorithm using different TSP instances, see [2]. The first 2 sections are divided into 2 sub-sections where the first sub-section describes the general setting and general design of the algorithm architecture and the second sub-section describes the design, parameters and implementation of the implemented algorithm and optimizations included in order for the algorithm to find better solutions. The last section is divided in 3 sub-sections where the first is a discussion on the results of the test cases which includes an interpretation of results, the second sub section is a discussion that compares the performance of both algorithms and the last sub-section contains a summary of limitations of both algorithms. For specific programming constructs and implementation details check the source code attached in the same folder this document is found, also to run the executables read the README file for guidelines. The programming language used to implement both algorithms was JAVA since the use of array lists and string regex when reading the input file made it easier to construct the algorithm. The next section discusses the Genetic Algorithm.

Genetic Algorithm

Genetic Algorithm Architecture

As described by Jacobson [4] , Genetic algorithms are based on the idea of Evolution by natural selection, i.e. it mimics how nature works in providing solutions to real life problems. Thus, in the algorithm the fittest individuals survive and may create offspring if they are fit enough. These offspring are assumed to have a better fitness than their current parents so that like in life the algorithm gets better individuals and thus gets closer to the solution. Each individual in the population must have a representation called its chromosomes. The chromosome representation depends on the problem, however classic representations include bit string representation and permutation encoding for problems where order is important in the solution, see [7].

Thus, as already mentioned in every Genetic algorithm there is a population which must be maintained between every generation, this population must be initialized at the start of the algorithm were normally the 2 mechanisms used to generate the population are random generation and heuristic initialization were the population is generated using a known heuristic. Which mechanism to use depends on the problem. Each individual in the population is normally then given a value, this value is called a fitness value where this gives an indication how fit is the individual, hence how likely he is to create offspring, or to make it to the next generation. [4], [5]

Another important characteristic of the Genetic algorithm is the fitness function, the fitness function gives the fitness value to each individual in the population and this is problem specific. Using these fitness values, the selection mechanism of the Genetic algorithm chooses which individuals should pass from each generation to the next, or for mating (Cross Over). There are many different selection mechanisms each could perform differently for different problems. These mechanisms are Tournament Selection, Ranked Based Selection and Roulette Wheel Selection. [4], [6]

The next 2 important mechanisms in the Genetic algorithm are Crossover and Mutation, these 2 mechanisms determine the new offspring's chromosome. Crossover creates new individuals by combining aspects from 2 parents mimicking how reproduction works in real life. The algorithm tries to create a fitter child by combining the best traits from the parents. However, in real life the children do not contain the same genetics as their parents, otherwise they would be an exact copy, thus this scenario is represented by mutation in the genetic algorithm. Mutation works by making slight changes in the chromosome at random. This is very important since it may introduce a new feature in the population that was not present in the previous generation and thus helps the algorithm from converging to a solution which is not good enough. [4], [7].

As a result, the Genetic algorithm takes a number of parameters that could affect the quality of the solution. Such parameters are:

- Crossover rate – Fraction of the population to be chosen for mating
- Mutation rate – Fraction of the children that will be mutated
- Population Size – The number of chromosomes to be maintained per generation

- Tournament Size – The size of the Tournament if Tournament Selection is used as selection mechanism. [7]

In addition to these parameters, mechanisms are normally added to the algorithm structure to improve the quality of the solutions. These mechanisms are normally there to enhance diversity. Diversity is important in a population because the algorithm could converge to a solution that is not the optimal solution. Mechanisms to increase diversity could be mutation, fitness sharing which means the individual's fitness is decreased when the population contains similar individuals and elitism which means that the best individuals from the previous population can continue to progress in the same way to the next generation. [7]

Per [4], the Genetic Algorithm is normally implemented as shown below:

1. Initialize Population
2. Evaluate fitness for each individual
3. Select parents for cross over
4. Add children to the next generation
5. Mutate children
6. Use elitism to maintain population size
7. Repeat from step 2 until some termination condition is met.

As seen in the above pseudocode a termination condition is needed in order to stop the algorithm. Per [7], the termination condition depends on the problem, but normally a number of pre-defined iterations are set, or whenever the solution is good enough if the global solution is known. The next section describes the implementation of the genetic algorithm applied to symmetric TSP, where the design and implementation decision of the important mechanism described in this section are discussed.

Genetic Algorithm for Symmetric TSP

This sub-section contains the genetic algorithm design applied to the symmetric TSP instance. Firstly, the representation for each individual had to be designed. Permutation encoding was used as a representation because as discussed by Piwonska [8] this gives the most natural way of encoding since the resultant individual that gives the shortest path is a path itself. This Permutation encoding is a list of cities where is each city in the problem is shown exactly once in a permutation representation. For example, in a 5-city problem the route 1->2->3->4->5 is represented as 12345. In the implementation supplied in the same folder this document is found, an array list is used to represent a route. Having the representation of the individuals the population then was needed to be generated in the algorithm

In this implementation the initial population was generated randomly [7], without any seed or bias. At first in this implementation the initial population was generated half randomly generated and the other half using the Nearest Neighbour algorithm, but after this was tested it gave the same result as having the initial population randomly generated, sometimes giving worse results because the solution would have higher chance of converging to a local optimum. Random generation gives more possibilities and chromosomes that might be needed to combined using cross over with other chromosomes in order to produce a better individual. This highlights again the importance of diversity.

The next mechanism that was implemented was the selection mechanism. The selection mechanism implemented was Tournament Selection since it gives better diversity. Tournament selection works by picking up a number of individuals from the population and the best individual in terms of fitness in this tournament is selected. In order to preserve diversity, the tournament size was chosen to be 2 so that not always the same individuals are selected for cross over, thus increasing the search space of the algorithm. In order to select individuals, a fitness function that gives some fitness value to each individual was implemented. Since the fitness had to be maximized in the genetic algorithm and the tour length had to be minimized, the fitness value was set to be the inverse of the tour length. [7], [9]

The next mechanism that had to be implemented was ordered cross over. Having 2 parents named A and B ordered cross over works by taking a subset from parent A and copying it in the child at the same place as they are present in parent A, then since no repetition is important, those elements from parent 2 that are not present in the child are copied in the order found in the empty elements of the child. In this implementation 2 children are created once starting from parent A and the other time starting from parent B. Ordered cross over was chosen over other cross over mechanisms because although other mechanisms might give better solutions in generation by generation basis, the ordered cross over is much faster and thus allows more processing of generations that could result in a faster convergence to a solution. [10]

In order not to create a population with similar individuals and thus avoiding local optima, the mutation operator had to be implemented. The mutation used was Inversion Mutation, see [11]. In inversion mutation a subset is chosen from the permutation representation of the route and this is

reversed to create a new mutated route. For example, if from the route 12345, 234 is chosen then the resulting permutation is 14325.

The Genetic Algorithm implementation for TSP was designed to have some parameters. These parameters were population size, tournament size, cross over rate, mutation rate, termination condition. The specific values for each of the mentioned parameters depended on the TSP instance used from [2]. Thus, a better explanation on the parameters could be seen from the evaluation section. However, the termination condition was always set to be 10,000 as from testing it was noticed that for small and big instances the solution always converges to the optimal or sub-optimal solution within 10,000 iterations.

As further additions to the algorithm to enhance diversity and to try and approximate the optimal solution better, more mechanisms were included in this implementation. The first enhancement to the algorithm was that the parents, the children and the mutated children were all merged into one population were in order to maintain the population size, then the best individuals were selected in terms of fitness from the merged population. This was done so that from one generation to the next, the best-found individuals are always kept. Also, in order for these best individuals not to create same copies of themselves, in the selection mechanism the parent cannot be mated with itself to create a child since it would create a copy of the parent. Also, if after mating 2 different parents, the result would be a child with the same fitness as some other individual in the next generation or previous generation, this child is mutated until it has unique fitness. This was done to prevent exact copies of individuals that would result into local optima converging.

In order to keep on avoiding local optimal converging and enhance diversity, after choosing the fittest individuals from the merged population, the less fit of the chosen population are then replaced with randomly generated tours. These tours were important to be generated because if after certain generations we get individuals with similar characteristics, the randomly generated tours may add that extra characteristic that was missing from the population, in fact the algorithm performed much better with this mechanism than without it. As noted above the algorithm converged in 10,000 iterations, however due to a lot of checking of uniqueness of individuals the algorithm performed slower, however the optimal solution was always reached for the majority of TSP instances from [2].

The following is the algorithm pseudocode of the genetic algorithm implemented:

```
Initialize the population randomly
while(numberOfGenerations doesn't exceed maximum size)
    Evaluate fitness of population
    for i=0 i<crossOverRate*populationSize i++ do
        parent1 = TournamentSelection()
        remove parent 1 from population list
        parent2 = TournamentSelection()
        child1,child2 = crossOver(parent1,parent2)
        if
            child1 and child2 are unique add them to next generation
        else
            mutate invalid child untill it is unique and add to next generation
    end

    for i=0 i<mutationRate*nextGenerationSize i++ do
    routeToBeMutated = randomly select route
    mutate(routeToBeMutated) until it is unique in the next and previos generation list
    add routeToBeMutated to next generation
    end

merge all individuals into 1 generation and select the best individuals
Remove randomRouteRate*populationSize individuals from population list and include random tours
end
```

FIGURE 1-PSEUDOCODE FOR GENETIC ALGORITHM FOR TSP

The next section describes the Ant Colony optimization.

Ant Colony Optimization

General Architecture of the Ant Colony Optimization Algorithm

Ant Colony Optimization algorithms are based on the natural behaviour of ants when they construct a root from home to a food source and back. Ants have no visibility; however, they are very capable of adapting to environmental changes if for example some obstacle is put between the path and the food source. This can be done due to the ant's deposit of pheromone trail on the ground whenever they are moving, in addition to that, ants prefer probabilistically to move in a direction where the pheromone levels are higher than other directions. Suppose, whenever an obstacle is put in a path between home and a food source there are only 2 choices of paths, then, half of the ants moving this path probabilistically choose to go one direction and half of the other ants probabilistically choose to go to the other direction. However, those ants that chose the shortest path from the 2 will re-constitute the interrupted path much faster compared to those that choose the longest path and thus the shortest path will get more pheromone trail and thus ants would later probabilistically prefer this path when going for the food source, since this would contain much more pheromone trail. It is important to note that although an ant prefers a path with higher level of pheromone it may take a completely different random path having a very low level of pheromone, this because a random variable is used to model the choice of ants. This would be clearer when applied to the symmetric TSP problem in the next section. However, these pheromone trails evaporate over time and thus trails which do not lead to food sources will gradually get eliminated over time, and thus the ants would later use better paths. These observations of ants are used to construct a strategy to find an optimal path for TSP. [12], [14]

Thus, due to the observation of Ant's behaviour an Ant Colony optimization algorithm normally takes the following approach:

- Initialize pheromone trails and parameters
- Construct an ant solution through the search space
- Update Pheromone
- Repeat from bullet point 1 until some condition is met. [13]

The bullet points in the above paragraph are a high-level abstraction on what happens in an ACO algorithm, thus as seen in the next sub-section this high-level abstraction is made clearer when seen how these were applied to the symmetric Travelling Salesman Problem. Dorigo [12], suggests that each Ant colony optimization algorithm has its own parameters for example, the rate of pheromone deposited by each ant in the path, how the ant moves and the importance of pheromone level and the rate of evaporation on each path. However, these depend highly on the application, thus a better explanation will follow in the next section. The next section describes how Ant colony optimization approach used in order to approximate TSP.

Ant Colony Optimization Algorithm for Symmetric TSP

This sub-section contains a detailed description on how an ant colony optimization algorithm was designed in order to approximate symmetric TSP. Since the ants are the driving force of the algorithm, a Class Ant was created in order to model an artificial ant. Since in TSP the ant cannot visit a city that has already been visited, this ant was designed to have a List in order to memorize the cities that have already been visited. In addition to that an ant was designed to have a variable that stores the starting index, so that when it completes the route it can go back to the starting vertex and thus calculate the Hamiltonian path's cost. Thus, in this algorithm the strategy that was taken was that, first a number of ants are placed randomly on a starting city where the starting city may be the same for ants. Each ant then moves to one city at every time step and modifies the pheromone levels between those edges used termed as local trail updating. Note that in this implementation a 2-D array was used in order to store the pheromone level between city 'k' and city 'j', although this wastes more memory, this was done this way since in this implementation the graph has no edges, since the graph was assumed to be complete. The ant keeps on moving to new cities until all ants complete the Hamiltonian cycle. When they are ready, the ant that gave the shortest tour in that iteration adds an amount of pheromone that is inversely proportional to the tour length. Finally, if the ant with the best path in some iteration gives a shorter path than previous iterations this is stored in order to not lose the shortest path. Note that as shown in figure 2 below, a number of ants are destroyed and created again within each iteration, this was done in this way because the createAnts () method creates an ant and instantly puts it on a vertex since the constructor to create the ant takes the starting city index as parameter. [12]

The following is a pseudocode of the strategy undertaken:

```
Initialize GeneticAlgorithmParameters : alpha,beta,qo,numberOfAnts,t0,pheromoneMatrix
        for(a predefined number of iterations) do
            createAnts();
            while(!allAntsCompletedTour) do
                for(int i=0;i<numberOfAnts;i++)do
                    moveAnts+update pheromone locally
                end
            end
            Get the ant given the shortest path and globally update the pheromone globally
            If this path is shorter than the previously saved shortest path save it.
        end
```

FIGURE 2-PSEUDOCODE OF STRATEGY TAKEN

In order to move from one city to the next the ant decides probabilistically which city to visit. The following was the formula used in the implementation as suggested by Dorigo [12]:

$$s = \begin{cases} \arg\max_{u \in M_k} \left\{ [\tau(r,u)] \cdot [\eta(r,u)]^{\beta} \right\} & \text{if } q \le q_0 \\ S & \text{otherwise} \end{cases}$$

FIGURE 3-FORMULA SUGGESTED BY DORIGO [12]

$$p_k(r,s) = \begin{cases} \dfrac{[\tau(r,s)]\cdot[\eta(r,s)]^{\beta}}{\sum_{u \notin M_k}[\tau(r,u)]\cdot[\eta(r,u)]^{\beta}} & \text{if } s \notin M_k \\ 0 & \text{otherwise} \end{cases}$$

FIGURE 4-FORMULA SUGGESTED BY DORIGO [12]

As seen in figure 3 above, some ant chooses to move from city r to city s depending whether q<=q0. This can be compared to a biased coin where q0 is some value given to the algorithm as parameter and q is a randomly generated number. τ (r, s) represents the amount of pheromone level between city 'r' and city 's' and η (r, s) represents the inverse distance between 'r' and 's'. Also, β acts as a controller on the importance of pheromone level and on vicinity. Thus, whenever q<=q0 the ant chooses to move to that city which has the highest combination of pheromone level and vicinity. This suggests that q0 must have a large value so that the ant would have a higher probability of choosing close cities which would have high level of pheromone and thus constitute shorter paths since shorter paths have higher level of pheromone [12].

However, whenever the bias test fails,i.e. q>q0 a random variable S is selected as can be seen in figure 3 above ,whose probability distribution depends on that shown in figure 4.The probability of an ant moving from city r to city s is also determined according to the level of pheromone and vicinity if the city is not already visited .If the city has already been visited it must not be visited again hence it has 0 probability of being visited. These probability values were then needed in order to implement the Roulette Wheel Selection mechanism which chose the city in the case when q>q0. Roulette Wheel selection works by generating a random number between 0 and 1 and summing the probabilities one by one together. Whenever the sum exceeds the random number value, that city which exceeded the random number is chosen. Thus, although those cities with highest combination of pheromone and vicinity have a greater chance of being chosen, this gives a higher chance to the ants to visit unexplored areas in the search. Thus, this preserves stagnation and local optima convergence. [6], [12],

As can be seen in figure 2 above in addition to moving to a new city the ant must update the pheromone level between the departing city and the new destination. In this implementation local pheromone updating was done as suggested by Dorigo [12] below:

$$\tau(r, s) = (1- \alpha) \cdot \tau(r, s) + \alpha \cdot \tau 0$$

Where α is the evaporation rate of pheromone, and τ0 is the level of pheromone to be deposited on the matrix of pheromones in one step. This step was important in order to avoid a strong edge being visited by all the ants, this is because α acts as an evaporation of pheromone and thus this enhances more exploration of different paths and avoid stagnation on local optima. [12]

As can be seen in figure 2 above, after each ant completed a tour, the ant that completes the shortest tour in that iteration has the ability to update the pheromone levels of the edges that make up the tour (in this implementation the matrix of pheromone). Global pheromone update rewards the path that the ant found with more pheromone and thus guides the ants to find shorter paths in the algorithm. Per [12], the global updating formula was given by:

$$\Phi(r, s) = (1-\alpha) * \Phi(r, s) + \alpha * \Delta(r, s)$$

$$\Delta(r, s) = (\text{shortest tour})-1$$

Thus, as seen from the formula above, the shorter trails get a higher level of pheromone update, however the longer trails get a love level of pheromone update, thus guiding the search to find the optimal shortest path in the complete graph. [12]

As can be seen in the above formulas and discussion, the algorithm takes a number of parameters. However, these shall be discussed in the next section since these are specific to the instance being benchmarked from [2]. The next section is the evaluation section which compares both algorithm's performance on benchmark TSP instances including parameters used and discusses some limitations of both approaches.

Evaluation

Results and Discussion on Parameters

This sub section will contain all the results when testing both algorithms with TSPLIB instances from [2]. At first all the results, parameters and expected outcomes will be given in a separate table for each algorithm each getting an interpretation of results. The 2 tables below summarize the results that were encountered during the experimentation. Note that the Metric type describes how the distance between 2 cities was defined since in TSPLIB there where distances of type GEO which means that the cities are defined on the globe and the distance is given as shown in [15], and there are also distance of types EUC_2D meaning the distance is Euclidean distance. Thus, when running the program, it is important to set the matrix of distances with the appropriate metric as it is discussed in the README file.

Genetic Algorithm									
Instance File	**Metric Type**	**Optimal Solution**	**Best Solution Result**	**Population Size**	**Cross Over Rate**	**Mutation Rate**	**Average Iterations Needed to Converge**	**Random Route Rate**	**Tournament Size**
Burma14.tsp	GEO	3323	3323	30	0.9	0.01	46	0.1	2
Ulysses22.tsp	GEO	7013	7013	30	0.9	0.01	174	0.1	2
Eil51.tsp	EUC_2D	426	435	30	0.9	0.1	1807	0.1	2
Kroa100A.tsp	EUC_2D	21282	21996	30	0.9	0.2	5382	0.1	2
Gil262.tsp	EUC_2D	2378	2759	30	0.9	0.2	16953	0.1	2

FIGURE 5-TEST DATA AND RESULTS FOR THE GENETIC ALGORITHM

As shown in the test data figure 5 above, the files burma14.tsp, ulysses22.tsp, eil51.tsp, kroa100A.tsp, gil262.tsp where tested with the given parameters as shown on the right-hand side of the table. The testing procedure for parameters was carried out by setting the number of iterations to 10,000. Then starting from the Cross Over rate parameter all the values between 0 to 1 in steps of 0.1 were tested in order to check which value gives faster convergence and best result. Then this procedure was repeated for mutation rate followed by random route rate and finally tournament size. As shown in the table above, the number of iterations for the solution to converge depend entirely on the size of the TSP instance. This is the case because the more cities there are in the instance, the more there are routes to be explored (factorial time). However, in order to make sure that the results are converging, the program is given a pre-defined number of iterations for example 100,000 where then if after a lot of iterations, the result is the same the program is terminated abruptly. This could be done because in each generation the shortest path is displayed in the program. It was concluded that for large instances after approximately 6000 iterations the solution converges.

As seen in the above table, the population size was set to 30 in all of the cases. This was done in this way because as discussed by [7], although increasing the population size increases the diversity and hence the chance to get better results this may come to a greater computational cost. Thus, a

balance had to be kept between solving a number of generations fast and in keeping good diversity so that the best possible solution is still outputted. Per [16], if the population size is set to a low value, only a fraction of the search space is evaluated, however if it is too large the GA slows down. Also, whenever the population size exceeds some limit value it does not compute the result faster. Thus, for each instance in figure 5 the algorithm was run by different population size parameter values and it was noted that the algorithm does not improve the solution for population sizes greater than 30. In addition to that, the algorithm slows down significantly for population sizes greater than 100. This is the case because in this algorithm in order to preserve diversity each chromosome created by crossover, mutation or random generation is checked whether there is one already present. Thus, although the algorithm gives us very good approximation values, these come to a computational cost.

Observing the remaining parameters in figure 5 it was concluded that the cross over rate needed to be high for both small instances and large instances for faster conversion and good solution results. This had to be the case because in this algorithm, since a merged population approach is taken, if in an iteration more children are created then there is more chance that there are fitter chromosomes in the next generation since the best are taken. It was also observed from figure 5 that as the TSP instance increases the mutation rate must also be increased. This happens because as the TSP instances increase the more chance there is to be stuck in local optima since there are more routes to be considered, thus higher mutation rate increases diversity, see [7]. As shown in figure [5] random route generation and tournament size were constant. This was the case because although random route generation is important to add diversity, if it is increased further the quality of the results diminishes. This is the case because some features can get lost if too much randomness is added to the population, hence it would be difficult to converge. Finally, tournament size was set to 2 because the best results where given for this parameter value. This reason happened because as discussed by [7] if the tournament size increases, then there is more chance that only the fittest individual/s are chosen thus leads to a greater chance of starvation and crowding.

From the above instances it was concluded that the genetic algorithm gives optimal results for small test cases and gives a near optimal solution with an average percentage difference of 3% for medium distances, and an average percentage difference of 8% for larger distances. Thus, as the number of cities increase, the quality of the solutions diminishes as expected.

Ant Colony Optimization Algorithm									
Instance File	**Metric Type**	**Optimal Solution**	**Best Solution Result**	**Number of Ants**	**q0**	**Beta**	**alpha**	**t0**	**Average iterations needed to converge**
Burma14.tsp	GEO	3323	3323	10	0.9	2	0.1	1/(n*NNH)	8095
Ulysses22.tsp	GEO	7013	7013	10	0.9	2	0.1	1/(n*NNH)	1329
Eil51.tsp	EUC_2D	426	426	10	0.9	2	0.1	1/(n*NNH)	65000
Kroa100A.tsp	EUC_2D	21282	21517	10	0.9	2	0.1	1/(n*NNH)	15962
Gil262.tsp	EUC_2D	2378	2536	10	0.9	2	0.1	1/(n*NNH)	11030

FIGURE 6-RESULTS FOR ACO APPLIED TO SYMMETRIC TSP

n = number of cities

NNH = tour length using nearest neighbour heuristic algorithm.

As seen in figure 6 above the parameters supplied to the algorithm are all the same for both the small instances and the large instances. This was done this way since according to Dorigo [12], these were the most robust in all different instance sizes. Strengthening this claim, whenever the algorithm was tested with different parameters, the quality of the results diminished. It was also noted and observed from figure 6 above, that as the number of cities increase, the more the result deviates from the optimal solution and the less iterations are computed in some unit time. For example, in eil51.tsp 65,000 iterations could be computed, however in Kroa100A less iterations were computed. This was due to the fact that an ant needed more time to complete a path, since there are more cities involved.

It was concluded from this test case that; the ant colony optimization algorithm gives optimal solutions for both small distances and medium distances. However, as the tsp instances get larger although it gives very near optimal solutions at an average percentage difference of 6%, the algorithm computes much less iterations per unit time, however the algorithm converges to a solution faster. It was also noted that a lot of generations had to be passed in order for the algorithm to not be stuck on local minima. For example, in eil51.tsp 65000 generations had to be passed for the ant to find the optimal solution. This enforces the claim suggested by [17], that although the ACO gives better results it is much more prone to fail in local optima especially for large instances.

Comparison of the performance between ACO and the GA

From the previous section, many observations were taken. From the quality of the results it can be observed that the Ant colony optimization algorithm outperform the genetic algorithm, in fact the Genetic algorithm only gave optimal solutions for the small instances. In fact, as the number of cities increased the genetic algorithm started to deviate faster from the optimal solution than ACO. On the other hand, the Ant colony optimization algorithm gave the optimal solution for all small and medium size instances and gave better near optimal solutions to large instances compared to the Genetic algorithm. One must also mention, that for smaller instances the ACO optimization algorithm on average does not always give the optimal solution, however the Genetic Algorithm always gave the optimal solution if the parameters in figure 5 were given.

As discussed by [17] and from the previous section, however the Ant colony optimization algorithm is more prone to the local minima problem. Thus, if the TSP problem instance has many local minima, the ant colony optimization algorithm may not be the appropriate algorithm to choose. The Genetic algorithm proposed in this document is robust in avoiding local minima due to always merging the population, avoid having crowding with validation mechanisms and mutations. Due to these mechanisms the Genetic Algorithm guarantees that over time, the quality of the solutions increases linearly, however this may take a lot of generations to pass. In comparison the ACO converges earlier to a near optimal solution, however the local minima problem may stagnate this fast conversion.

Another important observation was that in terms of computational power the Ant colony optimization performed more generations than the Genetic algorithm for small and medium instances, however for large instances this performance diminished . However , still the quality of the solution favoured more ACO than the Genetic Algorithm for a large number of cities.

Therefore, it was concluded that for small instances since on average the Genetic Algorithm always gives the optimal solution, it is the most ideal one to use compared to the ACO algorithm. On the

other hand, for medium size instances the ACO algorithm is much faster and computes better results than the GA, hence for medium size instances the ACO algorithm must be chosen. Finally, for large instances since the ACO algorithm gives better results it is suggested to use the ACO algorithm even though it computes less iterations since the ACO algorithm converges both to a better and faster solution compared to GA.

Limitations and Advantages of GA and ACO algorithm

As mentioned in the previous section, the Genetic Algorithm for TSP's main weakness is that although a genetic algorithm designed properly might guarantee that the quality of the solution improves linearly, it might take a lot of generations for the GA to converge to a near optimal solution or to the optimal solution. This makes the termination condition very difficult to design, in fact in this implementation a lot of iterations were given to the solution in order to observe if the solution converged or not, see [18]. Another limitation is that if the algorithm is to avoid crowding of some individual and multiple copies in the population, then this validation in the algorithm comes at a great computational cost, thus although the quality of the solution increases, the number of generations that can be computed by the algorithm decreases. However as already mentioned in the previous section the main advantage of the Genetic Algorithm is that if designed correctly it does not get trapped in local minima easily. Also, for TSP instances with a small number of cities the genetic algorithm on average always gives the optimal solution in a few number of generations and computational speed, thus making it the ideal algorithm to choose.

On the other hand, the Ant Colony Optimization Algorithm's limitations are that on average for small instances the Algorithm gives near optimal results. This is the case because the algorithm is more susceptible to failing in local minima, in our table although optimal solutions were given for the small instances, these were only the best results, on average the result was more likely not to be optimal. Thus, for smaller instances this does not make the ACO algorithm the best from the 2. However, as the number of cities increase, the ACO gives better near optimal results better than the GA thus making it the most ideal algorithm to use. Also, since ACO computes less iterations per unit time as the number of cities increase, this makes it much harder to determine whether the algorithm converged to a solution or not. Also, per [19], the ACO algorithm is much more adequate for dynamic applications where the environment is susceptible to change frequently, this is the case because the ants always update the pheromone levels and the pheromone levels evaporate over time, thus bad solutions are less likely to be taken.

Conclusion

Thus, it was concluded that both algorithms were implemented successfully and applied to benchmark problems from [2] of different sizes. As a result, the performance of both algorithms and their strength + weaknesses in all of the instances could be identified. In fact, the Genetic Algorithm performed better in average quality performance in terms of quality of the solution and fast conversion to solution, however the ACO algorithm performed better for larger instance problems in terms of quality of the solution and fast conversion to the solution. However, a limitation in ACO is that as the number of cities increase the less iterations could be done and thus it made it more difficult to design a termination condition. A further improvement that could have been done to both algorithms is to set the termination condition according to which file is being run, but this was not done this way so that if the user wants to run another file he does not need to modify the source code. Also, the termination condition was set to be a pre-defined number of iterations to find it easier to observe converging of results and to see after how many iterations the algorithms are converging.

More improvements could be done to both algorithms, for example in the Genetic Algorithm the population could have been seeded in order to converge faster. In the ACO algorithm the ants could have also had some local optimization of the path before updating the global pheromone. However, these were not done since for the test cases chosen the algorithm already gave a good near optimal result.

References

[1] "Traveling Salesman Problem (TSP) Implementation - GeeksforGeeks", GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/.

[2] "TSPLIB", Elib.zib.de, 1997. [Online]. Available: http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html.

[3] "Genetic Algorithms Introduction", www.tutorialspoint.com. [Online]. Available: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_introduction.htm.

[4] Jacobson, "Creating a genetic algorithm for beginners", Theprojectspot.com, 2012. [Online]. Available: http://www.theprojectspot.com/tutorial-post/creating-a-genetic-algorithm-for-beginners/3.

[5] "Genetic Algorithms Population", www.tutorialspoint.com. [Online]. Available: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_population.htm.

[6] "Genetic Algorithms Parent Selection", www.tutorialspoint.com. [Online]. Available: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm .

[7] "Genetic Algorithms", Cs.ucc.ie. [Online]. Available: http://www.cs.ucc.ie/~dgb/courses/tai/notes/handout12.pdf.

[8] A. Piwonska, GENETIC ALGORITHM FINDS ROUTES IN TRAVELLING SALESMAN PROBLEM WITH PROFITS. Bialystok: Bialystok University of Technology, 2010.

[9] F. salesman, "Fitness function in genetic algorithm for the travelling salesman", Stackoverflow.com, 2012. [Online]. Available: https://stackoverflow.com/questions/9649553/fitness-function-in-genetic-algorithm-for-the-travelling-salesman.

[10] "Order 1 Crossover Operator Tutorial", Rubicite.com. [Online]. Available: http://www.rubicite.com/Tutorials/GeneticAlgorithms/CrossoverOperators/Order1CrossoverOperator.aspx.

[11] B. Keresztury, Genetic algorithms and the Traveling Salesman Problem. Eötvös: Eötvös Loránd University, 2017.

[12] M. Dorigo and L. Gambardella, Ant colonies for the traveling salesman problem. Université Libre de Bruxelles, 1996.

[13] Shyamala and Prabha, "An Ant Colony Optimization approach to solve Travelling Salesman Problem", Pdfs.semanticscholar.org, 2014. [Online]. Available: https://pdfs.semanticscholar.org/c106/344de5da919624d55dc1aab02cf3de46e5e1.pdf.

[14] L. Jacobson, "Ant Colony Optimization For Hackers", Theprojectspot.com, 2015. [Online]. Available: http://www.theprojectspot.com/tutorial-post/ant-colony-optimization-for-hackers/10.

[15] "TSPLIB: Questions and Answers", Comopt.ifi.uni-heidelberg.de, 2000. [Online]. Available: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/TSPFAQ.html.

[16] "Parameters of GA - Introduction to Genetic Algorithms - Tutorial with Interactive Java Applets", Obitko.com. [Online]. Available: http://www.obitko.com/tutorials/genetic-algorithms/parameters.php.

[17] S. Haroun, B. Jamal and E. Hicham, "A Performance Comparison of GA and ACO Applied to TSP", 2015. [Online]. Available: https://www.researchgate.net/profile/Ahmed_Sabry15/publication/277017426_Article_A_Performance_Comparison_of_GA_and_ACO_Applied_to_TSP/links/5595ee1708ae21086d20814c/Article-A-Performance-Comparison-of-GA-and-ACO-Applied-to-TSP.pdf.

[18] D. Thomas, "What are the disadvantage of genetic algorithm?". [Online]. Available: https://www.quora.com/What-are-the-disadvantage-of-genetic-algorithm.

[19] N. Abreu, M. Ajmal, Z. Kokkinogenis and B. Bozorg, "Ant Colony Optimization", Paginas.fe.up.pt, 2017. [Online]. Available: https://paginas.fe.up.pt/~mac/ensino/docs/DS20102011/Presentations/PopulationalMetaheuristics/ACO_Nuno_Muhammad_Zafeiris_Behdad.pdf.

Statement of Completion

Item	Completed
Implemented GA	yes
Implemented ACO	yes
Evaluated GA vs ACO for selected instances	yes
Described GA architecture in report	yes
Described ACO architecture in report	yes