Abstract

temp

Contents

1 Introduction

The Travelling Salesperson Problem (TSP) is an NP-Hard problem which is commonly found in combinatorial optimisation, theoretical computer science and operations research. It's a simple problem on the surface which simply states that "Given a list of cities and distances between each pair of them, what is the shortest possile route that visits each city exactly once and returns to the origin city?" [1]

This question appears deceptively simple, however with large sets of cities the search space becomes incredibly large, and as such computation time can be astronomical, since before you even start trying to compute the shortest tour, there are $n!$ different combinations to compute, assuming a complete graph. Clearly this meant another angle had to be taken for this problem.

As such many people decided to go for a more heuristic approach towards solving the TSP problem. So instead of trying to find the optimal solution with an infeasibly long time span, we would settle for a 'good enough' solution with a much lower time complexity, however since TSP is a combinatorial optimisation problem it is much more difficult than continuous optimisation, with higher chances of falling into local optimum rather than the global optimum.

And yet even with this said, there exists an algorithm that has been created more than 30 years ago (1976) that has been proven to be at most 50% worse than the optimal solution for any given TSP problem and is known as Christofides algorithm. This algorithm has remained the best approximation available for over 30 years, and only recently in this year (2020) has another, slightly more efficient algorithm been found [4], although it has yet to be verified even if the general consensus is that it seems correct.

However another way of finding approximate solutions is becoming more and more popular over time, and has its roots deeply set in evolution and natural selection, hence the name they are given: Evolutionary algorithms. In essence these algorithms replicate the generational aspect of natural selection, taking the best individuals from a given population and 'breeding' them, these 'children' can then be tested to see if they perform any better than their parents, and if so replace the worst individuals in the population, and this process continues to repeat until a given goal is complete, whether that turns out to be an actual solution being found, until a good enough solution is found or after a certain number of generations have been looped through.

Genetic algorithms have already been used to try and solve the TSP which shall be discussed later, yet the major point of genetic algorithms is that you start with an entirely random population with which to create new solutions with a large degree of randomness in terms of the search operators. If however we were to replace one of the population with a strong approximate solution (x), how would that affect the algorithm? Theoretically in the worst case scenario this genetic algorithm would simply return x, since if no better one can be found, then x would remain in the population as the best solution, so we are no worse off compared to when we started the algorithm. However in the best case scenario the algorithm would produce a better approximate solution than x and return this instead.

As such, in this thesis I will be investigating the usage of both the mathematical and algorithmic methods of finding a solution to the Travelling Salesman Problem, and thereby seeing if these methods can be combined into a singular algorithm and investigating it's effect.

2 Background

2.1 Travelling Salesman Problem

As explained briefly earlier, the TSP problem is a simple concept, a route that travels to every city in a given set exactly once and then returns to the starting position. This can be done either as an optimisation problem or decision, though the differences between them theoretically remain minimal, in decision we ask if given a length L, there exists a tour through the cities which is less than L, and in optimisation we try to find the shortest possible route around the cities.

TSP has been thoroughly worked on already, and as such plenty of data sets are available for TSP to be used on [15, 16], some of which have yet to be solved for the true optimum, no matter how close some solutions are. However since the sites do contain the optimum solutions, or at very least the closest optimal solution, it makes evaluating our solution very easy since we can simply check the difference between our optimal solution for a given data set and their optimal.

2.2 Genetic Algorithms

Genetic algorithms take deep inspiration from natural selection and evolution in general. The very simplest description of a genetic algorithm is that given a starting population of predetermined size, calculate the fitness of each individual and choose the best of the population to 'reproduce' to create individuals that theoretically have the genes of both of these individuals, potentially giving it a higher fitness than its two parents. This child then has its fitness evaluated and if it is better than the worst individual in the population, it is added in place of it, this repeats for as many generations as we have chosen, and at the end of it we have a population of individuals who may have a better overall fitness than the initial population.

Most genetic algorithms take a very similar form, and though it is often altered for each individual case it is used in, most look very similar to the following pseudocode [10].

Algorithm 1: Pseudocode for a basic Genetic Algorithm

Initial Values: generationNum = 0, generationMax = N[a];
Randomly generate a population of size P individuals to be used initially;
Calculate the fitness of all P individuals;
Sort the population in descending order of fitness;
while *(generationNum < generationMax)* **do**
    Pick the M[b] best individuals from the start of the population;
    for *(each pair of parents in M[c])* **do**
        Apply the crossover and mutation operators to produce two new offspring;
        Evaluate their fitness;
    end
    Replace the worst individuals in the population with the offspring assuming said offspring have a higher fitness than them [d];
    generationNum += 1;
end

[a] Where N is a predetermined number of generations
[b] $2 \leq M \leq P$
[c] Selected randomly with replacement
[d] If all of the population are used to create new individuals then it will be the case that the entire population is replaced.

From the pseudocode it is most likely very clear that genetic algorithm is very well suited for optimisation problems, either maximising or minimising the fitness of the individuals in the population. This means it is definitely a viable option to solve the TSP, since the entire purpose of that is to minimise the weight of the path through all of the cities. However the pseudo code also shows there are a few operators and functions that need to be defined and explained before anything fruitful can be done.

2.2.1 Individuals

More of a clarification point more than anything, we need to decide what the individuals actually are within our problem of the TSP. Simply enough these individuals are solutions to the TSP, that is they are all routes that pass through each city and end back at the starting point, all without visiting a given city twice. The more important part however is how these routes should be stored in a way that the computer can work with, since naturally a picture of the route, whilst technically plausible, would be far more hassle than it would be worth given there exists much simpler encoding methods.

Given that a solution to the TSP requires that all cities within C be used, a solution can simply be seen as an arrangement of these cities, whether this be as an array, list or simply a string. Any one of these formats would be suitable and effective to allow for the crossover and mutation functions to be applied, since regardless of format, all solutions will be of the same size.

2.2.2 Objective Function and Fitness

As stated briefly before, the TSP is an optimisation problem at its core, and genetic algorithms are powerful at solving such problems, and since this is a single objective optimisation problem in this case, the objective function itself is quite simple and can be formed from intuition:

Minimise the following :

$$F(\mathbf{x}) = \sum_{i,j \in \mathbf{x}} d_{i,j}$$

The objective function we want to minimise is $F(\mathbf{x})$ where $\mathbf{x} = \{x_1, x_2, ..., x_N\}$ is a given solution of the TSP problem, or in other words a permutation of all N cities in our problem. $d_{i,j}$ represents the distance between cities i and j that are within $\mathbf{x}$, and we want to sum up the distances between them in order to get the cost of the full route.

So with that said it is blatant to see that the fitness function for any given route is simply the function we are trying to minimise, i.e.

$$\sum_{i,j \in \mathbf{x}} d_{i,j}$$

Which, stated simply, is the addition of all of the weights of the edges connecting our route, regardless of their value or type, i.e. distance, time etc.

2.2.3 Mutation

Mutation occurs after crossover has occurred, though in general is a much easier process, and this holds true for the genetic algorithm for TSP. There are many ways in which we can do mutations (but there are two that are used very commonly), one of which is to simply swap around two points in the tour (known as twors mutation [12]), for example given:

Solution: 123456

Lets say we wish to swap towns 2 and 6, the solution would become:

Solution: 163452

This solution should be equally valid since the graph of the towns should be complete, that is every town is connected to every other town, so the route is still valid since an edge still exists.

Another method of mutation is to instead reverse a sub-tour within the tour itself (known as Reverse Sequence Mutation (RSM) [12]), for example given the same example above, reversing the tour from 2 to 6 would give us:

Solution: 165432

Naturally reversing a subtour is more computationally difficult than simply swapping 2 cities at random, however research has suggested that using RSM produces much stronger results compared to the other common methods, whilst simultaneously not being that much more theoretically difficult to understand [12]. As such using RSM as the mutation operator for my own genetic algorithm would seem to be greatly beneficial.

It is important to note that mutation occurs by random chance, and so is not guaranteed to happen for any given child, otherwise the children might be too far spread across the search space and unable to close in properly to an optimum solution (either local or global).

2.2.4 Crossover

Crossover is another operator which heavily takes its inspiration from the process in meiosis with the same name. Within meiosis a pair of homologous chromosomes pair up during prophase 1, and form a bivalent, when this happens the chromosomes overlap at a given point known as the chiasma, and the two paired chromosomes exchange genetic material at this point. This given point is identical on both separate chromosomes, so in other words the exact same genes have been swapped between the chromosomes, yet their alleles may well be different after the swap.

To put this in simpler terms, let's say the segment of genetic material being swapped is for eye colour, in each chromosome the same section determines eye colour, and we are swapping those sections during meiosis, so after the 'swap' both chromosomes still have the gene for eye colour, yet the actual eye colour it will determine may differ.

In normal genetic algorithms this process is identical to its biological counterpart, since the assumption is usually made that we are crossing over at identical points along both solutions, and as such no 'genes' are being lost or repeated. However in a solution for TSP, we cannot make this assumption, if we are to cross at a given point between two solutions, it may well be the case that two cities are repeated, and obviously this is not a possibility we can afford to have in a problem which strictly states every city must be visited exactly once. As an example of what I refer to take the following example:

Solution 1: 153426
Solution 2: 654321

Now let us say we wish to perform a crossover between the solutions between positions 4 and 5 like so:

Solution 1: 1534|26
Solution 2: 6543|21

If we now swap the solutions after this point accordingly we get:

Solution 1: 153421
Solution 2: 654326

Underlined you can see we now have repeated cities as one error in the solution and another error in the sense that we are no longer visiting each city, so this is certainly not a valid solution to any TSP problem.

As such we must look at a different method of crossover, which takes the same core concept, yet avoids this issue detailed above. As a matter of fact there are a fair amount of different crossover operators that have been designed exactly for this purpose and shall be detailed shortly.

3 Literature Review

3.1 Different Crossover Operators

3.1.1 Partially Mapped Crossover

In this crossover operator [13], proposed by Goldberg and Lingle, two random cut points are used as its basis, instead of doing a singular crossover point like demonstrated above. After the cuts have been made one of the parents has its subroute mapped onto the other parent's subroute before the other remaining information is exchanged between the two parents. As an example take the following two parents who have the crossover points marked in the same way as before [13]:

$$\text{Solution 1: } (348|271|65)$$
$$\text{Solution 2: } (425|168|37)$$

In this scenario the mapping is between 2 and 1, 7 and 6, 1 and 8. So we swap these around and make two new children based off of them:

$$\text{Child 1: } (XXX|168|XX)$$
$$\text{Child 2: } (XXX|271|XX)$$

The next stage is to fill in the Xs with the appropriate cities from their corresponding parent, checking of course first that a city is not being repeated. (So Child 1 will receive Parent 1's cities and vice versa):

$$\text{Child 1: } (34X|168|X5)$$
$$\text{Child 2: } (4X5|271|3X)$$

From here we need to fill in the remaining Xs that have conflicts with other cities, the first X in child one would have originally been 8 from parent 1, though since there is already an 8 in child 1, we check the mapping which shows 8 is mapped to 1, so we then check is 1 is in the child, which turns out to be true, so once again we check the mapping and see 1 maps to 2, we check if 2 is in the child and it is not, hence the first X becomes a 2. We do the same for the second X in child 1, which should have been a 6 yet it cannot be, the mapping shows 6 is mapped to 7 so we test 7 and it does not conflict, therefore we have Child 1 being equal to:

$$\text{Child 1: } (342|168|75)$$

We compute child 2 in exactly the same way which then gives us:

$$\text{Child 2: } (485|271|36)$$

And therefore we get two output children to test the fitness with and add to the population if necessary.

3.1.2 Order Crossover

Another crossover operator is the order crossover proposed by Davis [13], and it builds offspring by choosing a subtour from the parents and preserving the relative order of cities in the other parent. As an example:

$$\text{Solution 1: } (147|258|36)$$
$$\text{Solution 2: } (482|653|71)$$

The offspring are created first by the subtour of their respective parents:

$$\text{Child 1: } (XXX|258|XX)$$
$$\text{Child 2: } (XXX|653|XX)$$

Then from the opposite parent of each solution, we take the list of cities starting from the second cut point, and removing any already present in the 1st child, e.g. in this case we take the tour 7,1,4,8,2,6,5,3 then remove 2,5 and 8 to give 7,1,4,6,3 which is then placed in the 1st child starting from the 2nd cut point to give:

$$\text{Child 1: } (463|271|71)$$

And child 2 is made in the same way:

$$\text{Child 2: } (728|653|14)$$

3.1.3 Cycle Crossover

Cycle crossover is an operator which takes a different approach compared to the other two detailed above. It works by splitting the parents into 'cycles' and then alternating adding each one to their children. It is an algorithm which is best described through an example, however we must use a different example from the one used previously for reasons that shall be discussed afterwards [13]:

$$\text{Solution 1: } (12345678)$$
$$\text{Solution 2: } (85213647)$$

We start by picking one of the starting cities, in this case 1 or 8. In this case we will pick 1, and this is added to the first child:

$$\text{Child 1: } (1XXXXXXX)$$

We then look at the city in the corresponding position of parent 2 and go to that appropriate position in parent 1, so that we can add it to child 2, in this case the parent 2 city is 8, so we go to position 8 whilst also adding it to child 1:

$$\text{Child 1: (1XXXXXX8)}$$

We continue this, next getting a 7:

$$\text{Child 1: (1XXXXX78)}$$

And then a 4:

$$\text{Child 1: (1XX4XX78)}$$

From here the corresponding point is 1, however since we already have used 1 a the very beginning, it is already in the list and so that is the end of this 'cycle'. At this point we fill in the remaining missing cities with the ones from the second parent to give:

$$\text{Child 1: (15243678)}$$

We can then do the same for child 2 which would give us:

$$\text{Child 2: (82315647)}$$

As I mentioned earlier we had to use a separate example from the one we used for the other crossover operators in this case, this was because of the drawback of this technique. If you used this operator on our other examples, you would find that both children would end up exactly the same as their parents, which is clearly a very large waste of computation.

In a paper by Abid Hussain et al. [14], they investigate the performance of each of these operators along with many others to see which would lead to the best solutions in a given number of iterations, the results of this experiment showed that partially mapped crossover and cycle crossover both performed very similarly, but were a significant margin away from the performance of the order crossover on the BERLIN52 instance which this experiment was run on. As such, due to the relative simplicity of order crossover compared to the other two examples, and cycle crossovers drawback of potentially producing identical children, it strongly suggests that the most suitable crossover operator to use in our genetic algorithm would be the order crossover.

3.2 Different Algorithms for TSP

3.2.1 Brute Force Algorithm

Much like the name suggests, a brute force algorithm doesn't really solve the problem in a clever way, it just brute forces a solution by testing every single solution within the search space of

the optimisation problem, however this by no means implies these types of algorithms are bad. When the problem we are dealing with is small, this algorithm works exceedingly well, and since it is just iterating through every possible solution, it is very easy to implement, and for smaller problems it isnt really worth creating a more complex solution when a brute force approach will find an optimal solution perfectly well [17]. As we said the algorithms main downside is its complexity, as the size of the problem increases, this algorithm becomes inefficient incredibly fast, though given enough time this algorithm is guaranteed to find the optimal solution within the search space given that it exists.

This algorithm isn't really suited for this project however, since the problems I wish to attempt within this will be very large by nature, and a brute force approach is definitely not well designed for that.

3.2.2 Greedy Algorithm

Greedy algorithms are a step above brute force algorithms, they have a much smaller time complexity ($O(n)$) and are equally as easy to implement. They are still better suited for smaller search spaces however but not for the same reasons as the brute force algorithm. Greedy algorithms take a very short sighted approach to finding a solution, at each stage it takes the shortest route it can to another city (assuming it hasn't been visited already) until all of the cities have been visited and it returns to the starting point. The main downside of the algorithm is its tendency to get stuck in local optima, which is best described with a diagram.

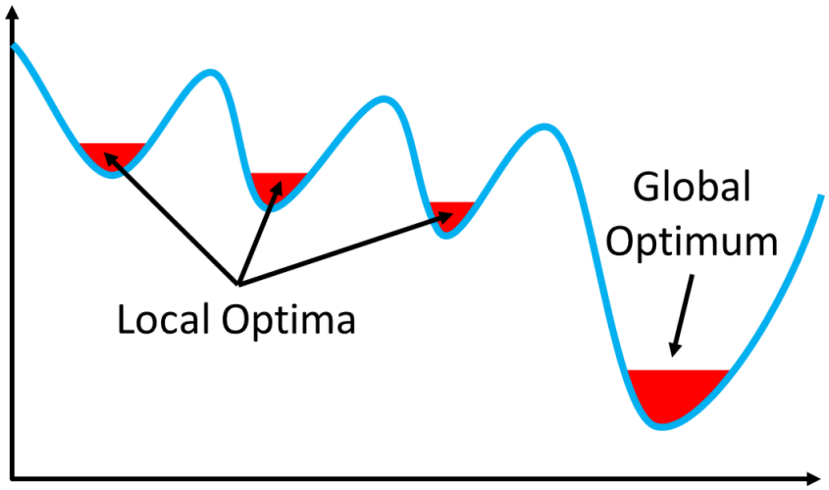

Figure 1: Full Graph

When the greedy algorithm travels down the line,it will continue to do so until it reaches an optima, be that local or global, it does not travel across the whole search space so it easily gets stuck in one of these 'dips' and will not reach the global optimum. In other words by taking the shortest route every time, we may miss a better route which is shorter overall but takes a longer path between some cities. This is much more of an issue for larger search spaces than smaller ones, but since we will be using larger search spaces, it makes this algorithm unsuitable for this experiment.

3.2.3 Held-Karp Algorithm

The Held-Karp algorithm is currently the best available for finding the optimum solution with a time complexity of $O(n^2 2^n)$, which whilst not great from a practical stand point, is leagues ahead of its brute force counterpart. This being said with its complexity still being quite high, it is unsuitable for very large data sets as it suffers greatly at these levels, especially so with its inefficient space complexity of $O(n * 2^n)$ which is not useful for solving the very large TSP problems without a very high-tech machine on which to run the algorithm, otherwise memory issues can occur as early as 30 city problems [2].

To try and explain the Held-Karp algorithm as simply as possible, it calculates suboptimal shortest paths that satisfy:

1. Each route must traverse an explicit set of nodes.
2. Each route must end on a specific node.

There is a recursive way of using this algorithm which uses the following formula:

$$cost(x_i, S) = \min_{x_j}\{cost(x_j, S \setminus \{x_i\}) + D_{ji}\}$$

(Credit to: [3])

Where:

1. V is the set of vertices
2. D is the distance matrix, so D_{ij} is the distance between points i and j
3. $S \subset V \setminus \{x_1\} \equiv \{x_2, x_3, ..., x_n\}$, $x_i \in S$
4. $cost(x_i, S)$ is the length of the shortest path from x_1 to x_i which visits each of the vertices within S exactly once
5. $x_j \in S \setminus \{x_i\}$.

In the case of S having size 1 we have the base case of:

$$cost(x_i, S) = D_{1i}$$

With this recursive formula we can create the cost function for S for the appropriate sizes, and since it is recursive each step is the basis for the next recursive loop. When we reach the case that $S = V \setminus \{x_1\}$ we have to then find

$$\min_{x_i}\{cost(x_i, V \setminus \{x_1\}) + D_{i1}\}$$

with $x_i \in S \equiv V \setminus \{x_1\}$

As stated before even though this is the best for finding the optimal solution to TSP, its time complexity is still exceptionally large and its space complexity also a huge issue. Furthermore for the purposes of this project it is unsuitable since it is already guaranteed to find the optimum,and combining it with any other method will be of no use, only serving to push the time required even further.

3.2.4 Christofides Algorithm

Even with the algorithm having been created over 30 years ago in 1976, there has yet to be another verified algorithm at this time that can produce more optimal approximate solutions to the TSP problem than this. This algorithm has been proven to be at most 50% worse than the optimal solution whilst having an $O(n^3)$ time complexity where n is the number of cities for which we are trying to solve the problem. [5] If we instead look at the best algorithm currently available to solve the TSP (aka the Held-Karp Algorithm) we can see that its complexity is $O(n^2 2^n)$ [2] which, whilst astronomically better than the original $O(n!)$, is significantly worse than the approximate solution that Christofides algorithm provides.

This is often the case with such computationally expensive problems, a balance must often be made between finding the optimal solution, and finding a solution in an appropriate amount of time. The Held-Karp algorithm focuses more on finding the optimum answer rather than finding it in a feasible amount of time, whereas Christofides Algorithm was designed to find as best an approximation as it can whilst still having a relatively feasible computation time which is what can make it so appealing to use.

One of the strong points about Christofides algorithm is just how simple it is, having only 4 or 5 major steps (many sources combine steps 4 and 5 whilst others do not [6, 7]) even if these steps are more likes processes themselves.

Nonetheless the steps to this algorithm are as follows:

1. Find a minimum spanning tree T.
2. Find a minimum weight perfect matching M for the odd degree vertices in T.
3. Calculate $M \bigcup T$.
4. Find an Euler tour of $M \bigcup T$.
5. Remove any repeated vertices.

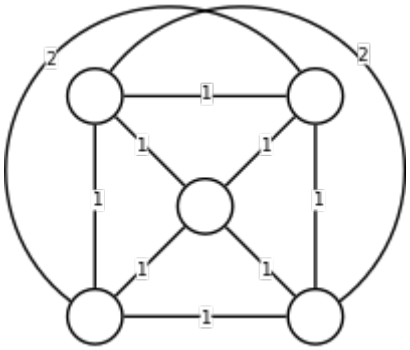

Figure 2: Full Graph

As stated the process is compressed into few steps and is quite simple to understand, and yet creating this algorithm within a programming language is more complex than one would initially assume. Finding a minimum spanning tree (Figure 3) of the full graph (Figure 2) is quite simple, the two most common algorithms known for solving this are Prims and Kruskals, both of which are effective methods with time complexities of $O(n)$ and $O(n^2)$ respectively [8] though choosing one or the other is mostly affected by the number of edges within the graph we are finding the minimum spanning tree for, with Prims being more suitable if there are a large number of edges, and Kruskals if not.

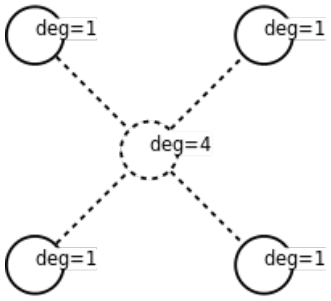

Figure 3: Minimum Spanning Tree

The minimum weight perfect matching of T (Figure 4) is accomplished by finding $\frac{T}{2}$ edges which 'link' together every vertex in T by the shortest edge possible. For this one of the most common algorithm in use appears to be the blossom algorithm and has a time complexity of $O(n^2m)$ in the worst case where n is the number of vertices and m is the number of edges [9].

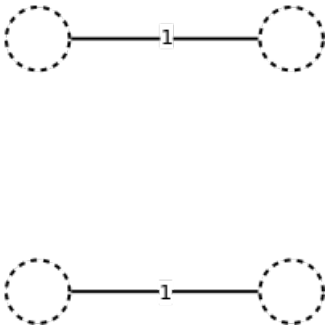

Figure 4: Minimum Weight Perfect Matching

From here we simply calculate the union of M and T (Figure 5), including any of the repeated edges that may be present, before moving onto the Euler tour. A Euler tour is simply a tour of the graph which visits each edge exactly once, regardless of starting or ending vertex (Figure 6).

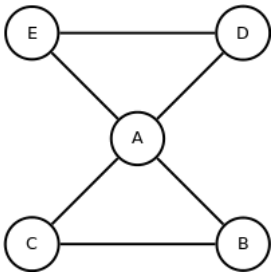

Figure 5: $M \bigcup T$

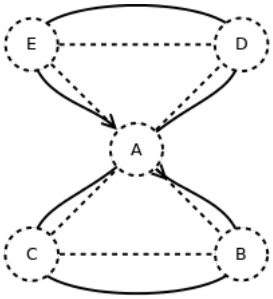

Figure 6: Euler Tour

Then need to remove any repeated vertices within our tour and replace them with the direct edge between those vertices, and once this is done we have the algorithms output (Figure 6).

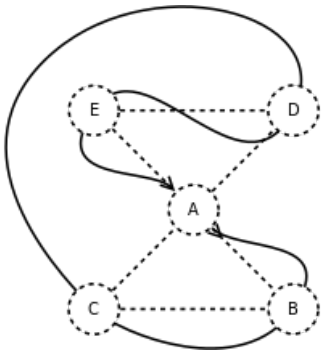

Figure 7: Christofides Algorithm Output

(Figures 1-6 taken from: Wikipedia: Travelling Saleman Problem [1])

With this algorithm output we now have a solution to the TSP for our specific graph, and this solution is guaranteed to be within 50% of the optimum [5], which is significantly better than trying to calculate the true optimum for a very large graph in an infeasible amount of time. However there is clear room for improvement, since for these very large graphs, a solution that is 50% worse is still drastically longer than the optimum, though is certainly a lot better than the alternative brute force approach.

This point is either a positive or negative depending on which angle you look at it from. If you look at it from the view that, for a monumental time complexity decrease of $O(n!)$ to $O(n^3)$ you are guaranteed to only be 50% worse, it looks at it in a positive light, though if you look at it in terms of it being 50% worse compared to the optimum for very large sets of cities which can have incredibly high distances, it looks quite disadvantageous. I am of the viewpoint that this is a positive thing, as having a solution that is already very good in such a short amount of relative time gives a good basis point for other work to be done with it, exactly as I plan to do within this project.

4 Implementation

4.1 Data Sets

For an investigative project such as this, it makes sense to use a variety of different data sets which will appropriately represent the different possible situations this algorithm could be used in, as such I have used three data sets of varying sizes to analyse my algorithm with. These data sets have all been taken from the same source [15] which also contain the optimum lengths of the tours for each data set available on the site, however there are no units for said distances yet this point is negligible for the purposes of this project. I have chosen three data sets of varying sizes to see how the algorithm will fair in each of them, these data sets are the cities of Qatar, Zimbabwe and Oman which have 194, 929 and 1979 cities respectively. Each of these data sets are encoded as qa194 (Qatar), zi929 (Zimbabwe) and mu1979 (Oman) and shall be referred to as such from now on for simplicities sake.

The optimum lengths for the datasets are as follows:

Dataset	Optimum Length
qa194	9352
zi929	95345
mu1979	86891

Each of these data sets is stored in a file format known as tsplib which gives some basic details on the data set along with the points of said data set.The details of the data structure are not really relevant, but the important aspects are simply that each city is stored with a number identifying it (quite simply a list from 1 to n) and next to said number is its coordinates, and from this it is simple to calculate the distances between the cities by just using the required distance metric (which is also specified within the file, and in the case of these data sets, it is all euclidean distance). As such converting from this file format into a data structure is exceedingly simple.

4.2 Data Structures

The travelling salesman problem involves dealing with multiple graphs of cities and the roads connecting them, as well as keeping track of multiple individuals which each represent a given tour around said graph. As such it made perfect sense to use some specific data structures for these to allow the usage of them to be as intuitive as possible. The graph itself was stored as a list of nodes, a 2D array of the edges and a string representing its name. This is nothing particularly outlandish but allows for the graph to be accessed easily enough. The edges were best stored as a 2D array since each pair of cities had an edge between them, as such the entirety of the array would be filled (bar the diagonals, which would represent the edge of a given city to itself). It would be possible to calculate the distance between each city as needed instead of storing them all at the beginning, however this would be less efficient in my opinion due to the number of times a given edge would have to be calculated, whereas storing them outright would sacrifice some space in order to reduce execution time. The nodes were simply small data structures which contained the x and y coordinates of each city along with its identifier (number in this case), exactly as shown in the screenshot above.

The other major data structure would be the "Individual" class, which is what a given solution to the travelling salesman problem would be stored as. The individual contains a list of the nodes in the order of which they would be visited, along with the 'fitness' of the individual (aka the length of the tour) which is calculated upon an instance of an individual being created. The data structure does not store the edges of the individual as these can be found out very quickly from the tour itself, simply searching the 2D array with the indexes being two consecutive nodes in the tour, this was to prevent any more space being used than necessary.

4.3 Parameters

Genetic algorithms come with a fair few parameters to tweak with, all of which can make quite the significant difference upon the results obtained. In my case said parameters were: "NUM_OF_CALCULATIONS", "POPULATION_SIZE", "PARENT_PERCENTAGE", "MUTATION_CHANCE" and "CROSSOVER_CHANCE". Going through them individually, "NUM

_OF_CALCULATIONS" is simply the value of the number of generations * the number of individuals, this was to keep the overall amount of 'calculations' done in a single genetic algorithm about the same, and allowed me to adjust the number of individuals and generations accordingly without changing this overall value. "POPULATION_SIZE" is simply the number of individuals in the population. "PARENT_PERCENTAGE" is used to calculate how many parents will be in the population, as an example with a population size of 1000, parents, we would take 0.2 of this (20%) to get 200 parents. "MUTATION_CHANCE" represents the chance that a given individual will undergo mutation and similarly "CROSSOVER_CHANCE" is the chance that a pair of individuals will have the crossover operator applied to them.

4.3.1 Generations and Population

In regards to optimising these parameters, the main example will be with the population size and the max number of generations (which is calculated by the number of calculations / population size).

Dataset	Population: 200 Generations: 25000	Population: 1000 Generations: 5000	Population: 2000 Generations: 2500
qa194	10540.6556793449	10543.3664673219	10109.16049177478
zi929	138258.542848618	124442.958821215	222572.64170034693
mu1979	634606.350605636	351729.696875395	2703446.74385851

Table 1: The results in the table are for the pure genetic algorithm, not the hybrid algorithm

Before discussing the results above, it is important to note that usually for genetic algorithms, smaller population sizes with larger generation numbers usually do better than larger population sizes. With that being said we can see here that the datasets actually react in the opposite way, the qa194 data set gives incredibly similar results for both, so they can be disregarded in this instance, however for the zi929 and mu1979 datasets we can see that the smaller population does worse on both accounts, and extremely so in the case of the mu1979 data set where is it around 300,000 worse. For a population size of 2000 it was once again very similar for the qa194 data set, but interestingly for the zi929 data set it was significantly worse than both whilst the mu1979 was a significant amount better instead. However since a larger population leads to a larger number of parents, it means the execution time of the program is significantly greater, and as such for this reason I have decided to stick with the 1000 population size as to balance out the results for all 3 datasets as well as decrease the running time of the program.

4.3.2 Mutation and Crossover

The other important parameters are the mutation and crossover chance, the mutation operator is applied after the crossover and applies to the children when they are generated, the crossover chance is basically the chance of two of the parents 'mating' to create two new children, if the crossover chance is not met then the two parents are simply carried forward as the children, essentially copying them.

Before going into the values used, I will mention that the RSM (Reverse Sequence Mutation) operator was used for mutation where a sub-tour within the overall tour is reversed, the reasoning behind this is quite simply that from past research papers the results have shown that this mutation operator performs better than the alternative [12] which is where we swap two cities in the tour around, and example of both is shown below for convenience.

TWOR Mutation (swapping two cities):

[123456] mutates to [153426]

RSM Mutation (reversing a sequence):

[123456] mutates to [154326]

In regards to the crossover operator it is true in much the same way, with order crossover having the best results within past research [14] and as such it was clearly the most logical choice to choose this crossover operator for the genetic algorithm.

In regards to optimising them I used the same method as used for the population and generation sizes, as shown in the table below:

Dataset	Mutation: 0.1	Mutation: 0.4	Mutation: 0.9
qa194	10220.556448011208	10042.534388187263	10307.448956457112
zi929	180637.17314356696	154080.72513430865	289831.6821760032
mu1979	1273721.9043312161	1461183.9174761462	2738641.7549757827

Table 2: Once again the results in the table are for the pure genetic algorithm, not the hybrid algorithm

Here we can see that a mutation of 0.9 is significantly worse on the larger datasets than the other two mutation values, and this is to be expected since mutation rates are never supposed to be as high as this, we can also see that the differences between 0.1 and 0.4 aren't as large as expected, however between the two a mutation of 0.4 does better and even if it does worse on the largest data set, there is a significant amount of variation within the genetic algorithm, so the difference may be due to that. As such I shall choose a mutation of 0.4 for the results of the hybrid.

And then below we see the same for optimisation of the crossover chances. In general the chance for crossover should be quite high whilst the mutation percentages tend to be a lot lower and we can see that these chances definitely reflect this general trend.

Dataset	Crossover: 0.1	Crossover: 0.4	Crossover: 0.9
qa194	10536.952603005655	10152.56527247845	10412.855764138969
zi929	193926.6438033023	161403.49362711725	153977.29243174416
mu1979	2539262.528786441	1874666.6607498457	1451559.7225074822

Table 3: Results in the table are for the pure genetic algorithm, not the hybrid algorithm

For the above data we can see that a crossover of 0.1 does significantly worse on the largest data set and does moderately worse for the medium sized one, the smallest data set once again doesn't really tell us much in all three cases since the variation is always incredibly minor, yet the other two show that a crossover of 0.9 has the best results in both cases compared to a crossover of 0.4, this matches what is expected since in general a higher chance of crossover is usually better. As such can see the best values for mutation and crossover are 0.4 and 0.9 respectively and therefore these are the values I used for my hybrid algorithm.

4.4 Christofides Algorithm

The implementation of Christofides algorithm was thankfully quite straight forward, the Individual steps within the algorithm were fairly easy to program except for a couple.

4.4.1 Find the minimum spanning tree

This was far from a difficult task, and is a very basic and common function that is used on a significant number of graphs. The only decision that had to be made with this function was whether to use Prim's or Kruskal's algorithm for finding the minimum spanning tree. From previous research [8] it was clear that Prim's was the better of the two to implement so I did exactly that.

4.4.2 Find a minimum weight perfect matching

This would have been a difficult algorithm to implement, and even more so to do in an efficient manner, as such for this stage and one other I used a library that is freely available [18] which contains a function to calculate this exactly, the only awkward part of this was manipulating my graph data structure into a form that the algorithm would accept and be able to use properly, however this was more time consuming than actually difficult to accomplish.

4.4.3 Calculate the union of the minimum spanning tree and minimum weight perfect matching

Very self explanatory, simply adding all of the edges from the minimum weight perfect matching to the list of edges in the minimum spanning tree was enough to complete this step in an efficient manner.

4.4.4 Find a euler tour of the union graph

In the same manner as the minimum weight perfect matching, calculating a euler tour for the union graph would have been difficult to implement quickly and efficiently, and implementing this isnt the main part of this project, and as such another library was used [19] to calculate the tour. The calculated tour then needed to be converted back into my own data structure which was not a complicated process at all, it simply required converting the edges into a list of cities instead.

4.4.5 Remove repeated vertices

This step is incredibly easy to do if you decide to do it simply, just go through the list checking each city and seeing if it appears again further down the line, then you just remove any that were found afterwards and you have your completed tour of the cities, however part of my project was to test to see if multiple individuals being added to the population would lead to a better or worse result, and it is this stage here in which the crux of that investigation begins

and how this function can become quite a bit more complex, and that shall be discussed in more detail later.

Overall these steps together will create a singular tour which will be at most 1.5 times worse than the optimum tour around a given graph. This individual can then be plugged directly into the population of the genetic algorithm and we can then allow to generations to run as many times as we deem necessary to achieve fruitful results. Before that however:

4.5 Multiple Christofides Individuals

As stated just a little bit ago we can change the simple step of simply removing repeated vertices to be quite a bit more complex in order to generate multiple individuals from it. Notably each of these multiple individuals will be worse than that of the singular individual, however the entire purpose of a genetic algorithm is to work with worse individuals and create better ones from them, so perhaps having multiple worse individuals that are spread more over the search space will allow the algorithm to find its way to the global optima (or somewhere close) much more quickly.

With this in mind the following code is what was produced to create said individuals, the specifics are not incredibly important and only the more important aspects shall be explained below.

```
public List<List<Node>> findAllPossibleCycles(List<Node> initialCycle) {
    //from any given repeat we can create 4 different tours, this subroutine can calculate all of them given enough time
    final int FREQUENCY_LIMIT = initialCycle.size()/12;
    List<List<Node>> allCycles = new ArrayList<>();
    List<List<Node>> cyclesStack = new ArrayList<>();
    int frequency = FREQUENCY_LIMIT; //the growth of cycles is incredibly large, only taking every nth one of this
                                     //allows for a much shorter running time
    cyclesStack.add(initialCycle);
    List<Node> reverseCycle = new ArrayList<>(initialCycle);
    Collections.reverse(reverseCycle);
    cyclesStack.add(reverseCycle);
    while (!cyclesStack.isEmpty() && allCycles.size() < 10) { //we only want a max of 10 individuals
        List<Node> cycle = cyclesStack.get(cyclesStack.size()-1);
        cyclesStack.remove(cycle);
        List<Node> cleanedCycle = new ArrayList<>(); //cycle without repeats
        for (int i = 0; i < cycle.size(); i++) {
            if (i != 0 && i != cycle.size() - 1) { //the first and last node have to be repeats by definition of hamiltonian cycle
                if (!cleanedCycle.contains(cycle.get(i))) {
                    cleanedCycle.add(cycle.get(i));

                } else {
                    if(frequency++ >= FREQUENCY_LIMIT) { //every nth cycle we will add to the stack to generate more off of
                        List<Node> cleanedCycle2 = new ArrayList<>(cleanedCycle);
                        Collections.reverse(cleanedCycle2.subList( i: cleanedCycle2.indexOf(cycle.get(i)) + 1, cleanedCycle2.size()));
                        cleanedCycle2.addAll(cycle.subList( i: i + 1, cycle.size()));
                        cyclesStack.add(cleanedCycle2);
                        frequency = 0;
                    }
                }
            } else {
                cleanedCycle.add(cycle.get(i));
            }
        }
        if(!allCycles.contains(cleanedCycle))
            allCycles.add(cleanedCycle);
    }
    return allCycles;
}
```

Figure 8: Code for multiple Christofides individuals.

The first important thing to note is how this algorithm works overall, for any given repeat we

encounter we can remove either the first instance or the second, doing this will lead to two separate tours which we can then iterate on again to remove the other repeats. Furthermore, not only can we remove either one, we can also reverse the tour and remove each of them. As such for every repeat we can create 4 different tours, and therefore it is obvious to see that the number of tours we can create will grow exponentially fast, because of this the time taken by the algorithm would be far too long in most tour sizes to be of any use.

With this in mind my code works by adding each of these newly created tours onto a stack of tours which will be iterated on. WE still have to tackle the issue of the exponential growth, and to do this I used a "FREQUENCY_LIMIT" which decided how many tours would be added to the queue and how many would simply be discarded. e.g. if the frequency limit was 5 then only every 5th tour generated would be added back onto the stack, and as such a lot of the tours could be discarded, and we do not need many of them in the first place, since the idea of this is to add a few individuals to the population, instead of entirely overriding it, and with this in mind there is a condition on the while loop used to empty the stack which also stops once 10 tours have been reached.

When every nth tour is reached, the function will create a copy of the tour and reverse it, then add it to the stack to be worked on each, and the program then continues on its current tour to remove all of its repeats before in order to add to the final list of tours which we can then use. The frequency chosen here was the size of the tour divided by 12, this was to try and make said size dynamic, such that smaller tours would have the chosen tours closer together and the larger tours further away, and this in turn was to create as large a variety of tours as possible, instead of ending up with many tours that were incredibly similar, as this would simply lead to getting caught in a local optimum very quickly.

Once execution of this is finished all 10 of the tours are to be added to the generation, with the other 990 (to give 1000 individuals) being randomly generated which will then be used by the genetic algorithm for 5000 generations on each data set.

5 Results

With everything now in place we can look into the results achieved by the algorithms and explain how and why they are the way they are.

5.1 Algorithm Evolution

The first results I will show are the evolutionary graphs of the singular hybrid algorithm alongside the pure genetic algorithm, so that we can see how their individuals are improving as the generations go by. The full tables of data for each of the graphs can be found in appendix A.

It is important that the layout of the graphs is first explained. There are two lines on each graph, the blue line represents the results of **just** the genetic algorithm and the orange line represents the results of the hybrid algorithm, and the x-axis represents the generation number. Most importantly is the scales, the results for genetic algorithm and hybrid algorithm were vastly different, so two different scales have had to be used in order to show both lines on one graph, and this was the best way to represent the data as it most clearly shows how the trends of both differ. The scale on the **left** of each graph represents the genetic algorithm (blue line) and the scale on the **right** represents the hybrid algorithm (orange line).

5.1.1 Qa194 Data Set

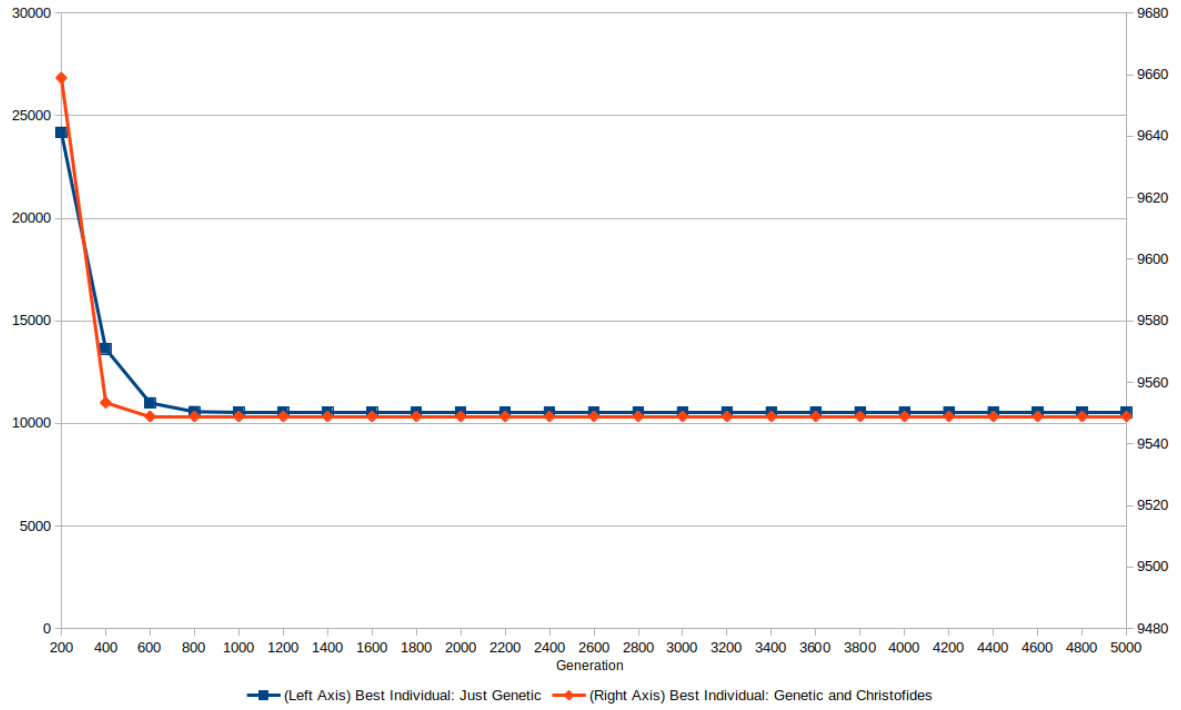

Figure 9: qa194 data set evolution

The first thing that immediately jumps out on this graph is how quickly both of these algorithms plateau, and deceptively it appears like both have plateaued very close to one another, however one look at the scale shows that there is a significant difference between the two values they have stuck with. The values specifically are 10543.3664673219 for the genetic algorithm and 9548.91872392524 in this case, however the general results of the algorithm will be discussed later. As we can see the hybrid algorithm hits the plateau about the same time as the genetic algorithm, however said plateau is hit at a tour which is 1000 better than the genetic algorithm, which means that even if they have stopped improving at the same time, the local optima that they have both become trapped in are certainly different (and in the hybrids case, better).

As such we can see in this small data sets case, after the same number of generations (approximately 800) we have a much better result with the hybrid compared to the genetic algorithm, and on top of this still we can see that the hybrid algorithm has been able to reach around 200 from the global optimum (9352) compared to the genetic algorithm which is around 1200 off, which is significantly better result.

5.1.2 Zi929 Data Set

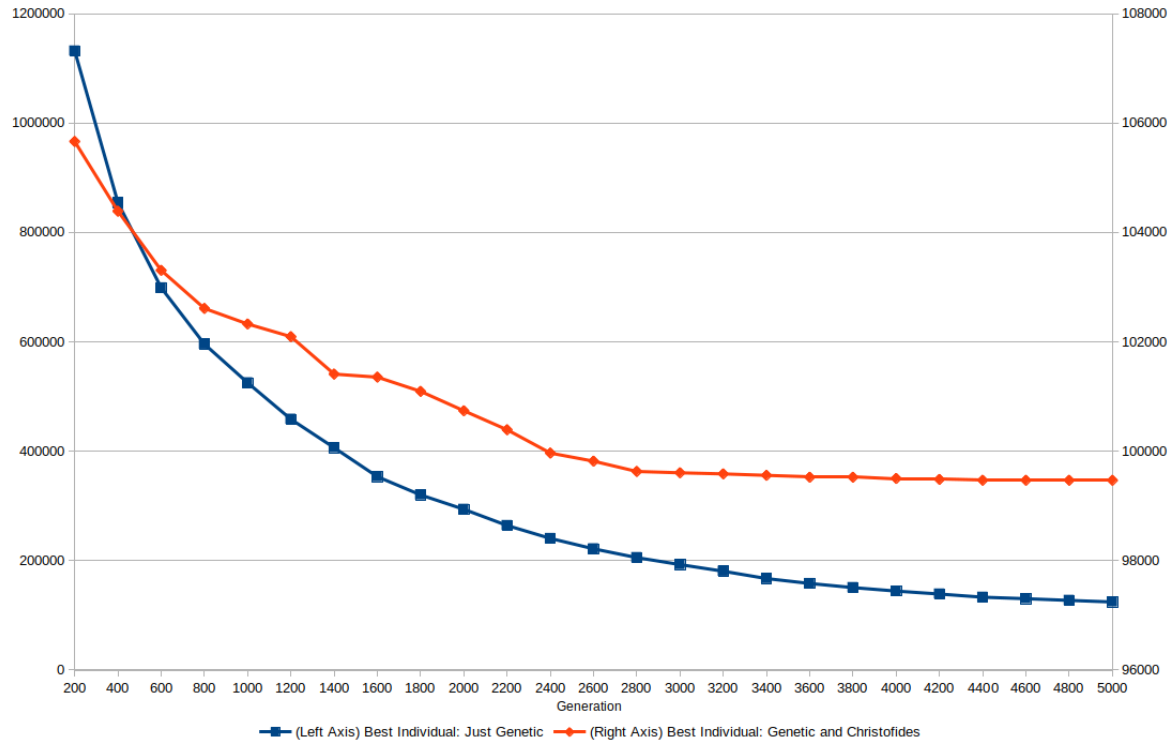

Figure 10: zi929 data set evolution

This evolutionary graph shows much more interesting data compared to the previous one, as we can see both algorithms take a larger number of generations before they start to plateau, and the genetic algorithm only just starts to plateau at the 5000th generation, whereas the hybrid seems to have plateaued closer to the 3000th generation.

If we look at the general trend of them, we can see that the genetic algorithm starts decreasing very quickly and then trails off, decreasing less and less until it stops, and this is the expected trend, however the hybrid algorithm has a stranger shape to it, despite the algorithm solely the genetic part at this point, the rate at which the individuals improve is not as consistent as the pure genetic algorithm, instead decreasing at strange intervals, before eventually trailing off at around the 3000th generation. That being said, even if this is the case the results of the genetic algorithm are still significantly worse than that of the hybrid algorithm. The genetic algorithm had a final result of 124442.958821215 whereas the hybrid algorithm had a result of 99467.6109972511 which is around 30,000 better. In comparison to the global optimum which is known to be 95345 we can see that once again the hybrid algorithm gets very close respective to the size of the tours, getting within 4000 whereas the genetic algorithm was around $29,000$ off, which is obviously significantly worse.

With this being said it appears neither of these had quite reached a true plateau, as the table within appendix A2 shows that they were still improving, if only marginally, as such given more time it would be interesting to see at what point they would both plateau fully.

5.1.3 Mu1979 Data Set

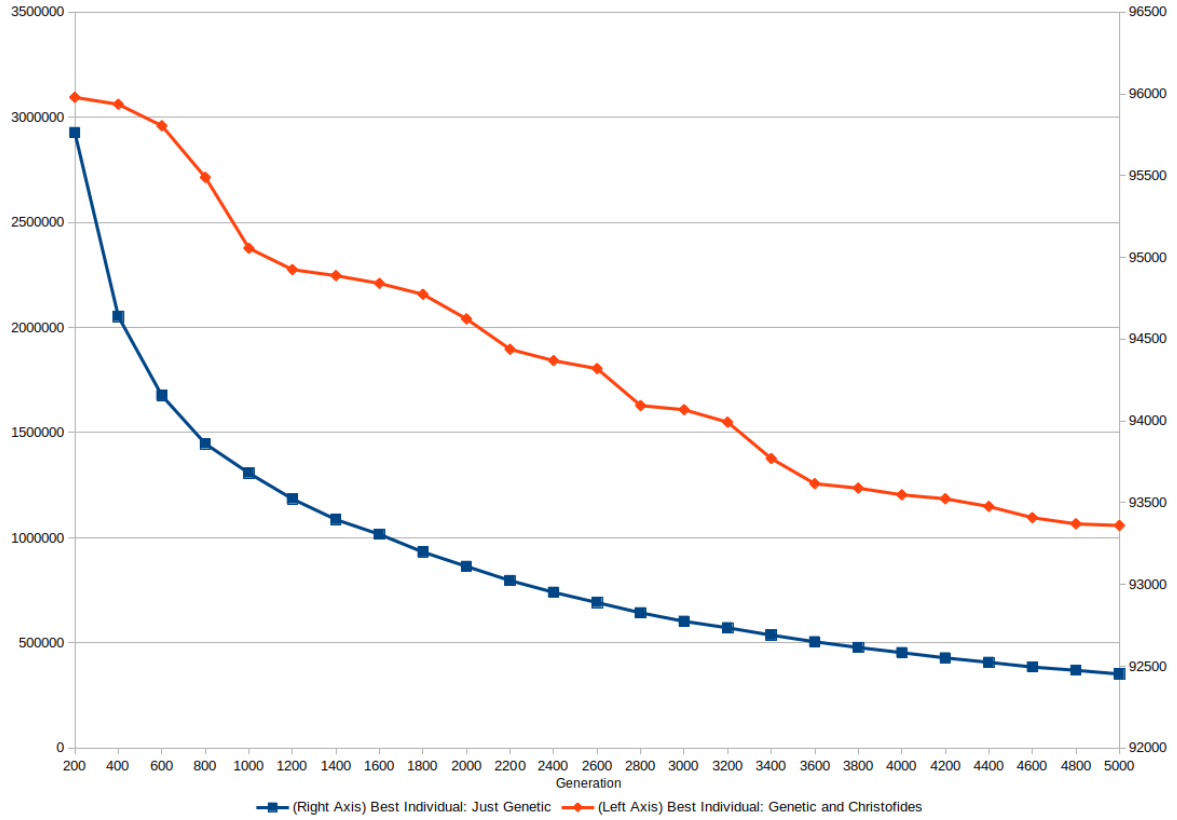

Figure 11: Mu1979 data set evolution

Finally we have the largest data set, and we can notice that the trends for each algorithm are fairly similar to what they were on the previous data set, the genetic algorithm starts decreasing a lot quicker before slowing down, whilst the hybrid algorithm is significantly more unpredictable to find a consistent trend with. What else we can see is that neither have plateaued yet, and as such this data could definitely do with being extended in the future if more time was available, as the genetic algorithm and the hybrid algorithm both have decent capacity for further improvement.

Nevertheless analysing what data is here we can see that the scaled this time are ludicrously different, the starting point for the genetic algorithm is around $3,000,000$ whereas for the hybrid it is around $96,000$ and as such the hybrid algorithm is already at a significant advantage before any generations have been calculated. Despite this we can see that the genetic algorithm does improve a lot, down below $500,000$ and the hybrid algorithm also improves, though not at all to the same degree.

However if we look at the final results in this case we can see there is no contest between the two; the genetic algorithm had a result of $351,729.696875395$ and the hybrid algorithm had a result of 93360.7690694728. With the optimum being $86,891$ we can clearly see that the hybrid algorithm is insurmountably better, with it being only $7,000$ worse compared to the genetic algorithm being more than 4 times worse than optimum, this isn't a one off occurrence either as we shall see later on.

References

[1] Wikipedia: Travelling Salesman Problem
https://en.wikipedia.org/wiki/Travelling_salesman_problem (Accessed 20 October 2020)

[2] Hutchinson, C. et al. (no date) CMU Traveling Salesman Problem, p. 25. Available at: https://www.math.cmu.edu/~af1p/Teaching/OR2/Projects/P58/OR2_Paper.pdf (Accessed: 27 October 2020)

[3] Nguyen, Q. N. (no date) Travelling Salesman Problem and Bellman-Held-Karp Algorithm, p. 5. Available at: http://www.math.nagoya-u.ac.jp/~richard/teaching/s2020/Quang1.pdf (Accessed: 12 November 2020)

[4] Klarreich, E. (no date) Computer Scientists Break Traveling Salesperson Record, Quanta Magazine. Available at: https://www.quantamagazine.org/computer-scientists-break-traveling-salesperson-record-20201008/ (Accessed: 22 October 2020).

[5] Christofides, N. (1976) Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem. CARNEGIE-MELLON UNIV PITTSBURGH PA MANAGEMENT SCIENCES RESEARCH GROUP. Available at: https://apps.dtic.mil/sti/citations/ADA025602 (Accessed: 22 October 2020).

[6] Sitters, R. (no date) Chapter 2: Greedy Algorithms and Local Search. Available at: https://personal.vu.nl/r.a.sitters/AdvancedAlgorithms/2016/SlidesChapter2-2016.pdf (Accessed: 27 October 2020).

[7] Christofides algorithm (no date). Available at: https://xlinux.nist.gov/dads/HTML/christofides.html (Accessed: 27 October 2020).

[8] Huang, F., Gao, P. and Wang, Y. (2009) Comparison of Prim and Kruskal on Shanghai and Shenzhen 300 Index Hierarchical Structure Tree, in 2009 International Conference on Web Information Systems and Mining. 2009 International Conference on Web Information Systems and Mining, pp. 237241. doi: 10.1109/WISM.2009.56.

[9] Kolmogorov, V. (2009) Blossom V: a new implementation of a minimum cost perfect matching algorithm, Mathematical Programming Computation, 1(1), pp. 4367. doi: 10.1007/s12532-009-0002-8. Available at: http://mpc.zib.de/archive/2009/1Kolmogorov2009_Article_BlossomVANewImplementationOfAM.pdf (Accessed: 29 October 2020)

[10] Melanie, M. (no date) An Introduction to Genetic Algorithms, p. 162. Available at: http://www.boente.eti.br/fuzzy/ebook-fuzzy-mitchell.pdf (Accessed: 3 November 2020)

[11] Penev, M.K.V.S.S., 2005. Genetic operators crossover and mutation in solving the TSP problem. In International Conference on Computer Systems and Technologies. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.83.7264&rep=rep1&type=pdf (Accessed: 5 November 2020)

[12] Abdoun, O., Abouchabaka, J. and Tajani, C. (no date) Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem, p. 18. Available at: https://arxiv.org/pdf/1203.3099.pdf (Accessed: 5 November 2020)

[13] Hussain, A. et al. (2017) Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator, Computational Intelligence and Neuroscience, 2017, pp.

17. doi: 10.1155/2017/7430125. Available at: http://downloads.hindawi.com/journals/cin/2017/7430125.pdf (Accessed: 10 November 2020)

[14] Otman, A. (no date) A Comparative Study of Adaptive Crossover Operators for Genetic Algorithms to Resolve the Traveling Salesman Problem, International Journal of Computer Applications, 31, p. 9. Available at: https://arxiv.org/pdf/1203.3097.pdf (Accessed: 12 November 2020)

[15] TSP Test Data (no date). Available at: http://www.math.uwaterloo.ca/tsp/data/index.html (Accessed: 12 November 2020).

[16] TSPLIB (no date). Available at: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ (Accessed: 12 November 2020).

[17] Baidoo, E. and O., S. (2016) Solving the TSP using Traditional Computing Approach, International Journal of Computer Applications, 152(8), pp. 1319. doi: 10.5120/ijca2016911906. Available at: https://www.researchgate.net/profile/Evans_Baidoo2/publication/309224559_Solving_the_TSP_using_Traditional_Computing_Approach/links/58066b6408ae5ad188166189.pdf (Accessed: 12 November 2020)

[18] org.jgrapht.alg.matching.blossom.v5 (JGraphT: a free Java graph library) (no date). Available at: https://jgrapht.org/javadoc-1.3.0/org/jgrapht/alg/matching/blossom/v5/package-summary.html (Accessed: 29 April 2021).

[19] HierholzerEulerianCycle (JGraphT: a free Java graph library) (no date). Available at: https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/alg/cycle/HierholzerEulerianCycle.html (Accessed: 29 April 2021).

Appendices

A Evolutionary Graph Data

This section shows the full tables of data used to generate the graphs, the only benefit this data gives over the graphs is being able to see the exact values at each generational landmark.

A.1 Qa194 Data Table

A.2 Zi929 Data Table

A.3 Mu1979 Data Table

Generation	(Left Axis) Best Individual: Just Genetic	(Right Axis) Best Individual: Genetic and Christofides
200	24183.8507855475	9658.95914978959
400	13631.8085464172	9553.40634861928
600	10997.0396460848	9548.91872392524
800	10570.0541109532	9548.91872392524
1000	10543.3664673219	9548.91872392524
1200	10543.3664673219	9548.91872392524
1400	10543.3664673219	9548.91872392524
1600	10543.3664673219	9548.91872392524
1800	10543.3664673219	9548.91872392524
2000	10543.3664673219	9548.91872392524
2200	10543.3664673219	9548.91872392524
2400	10543.3664673219	9548.91872392524
2600	10543.3664673219	9548.91872392524
2800	10543.3664673219	9548.91872392524
3000	10543.3664673219	9548.91872392524
3200	10543.3664673219	9548.91872392524
3400	10543.3664673219	9548.91872392524
3600	10543.3664673219	9548.91872392524
3800	10543.3664673219	9548.91872392524
4000	10543.3664673219	9548.91872392524
4200	10543.3664673219	9548.91872392524
4400	10543.3664673219	9548.91872392524
4600	10543.3664673219	9548.91872392524
4800	10543.3664673219	9548.91872392524
5000	10543.3664673219	9548.91872392524

Generation	(Left Axis) Best Individual: Just Genetic	(Right Axis) Best Individual: Genetic and Christofides
200	1131570.0643259	105660.91994179
400	854116.407093739	104386.767562323
600	698706.409970391	103305.590022213
800	595736.336490042	102609.782380465
1000	525003.112325443	102325.881053197
1200	458410.840154102	102094.324246916
1400	406324.28775855	101409.061195255
1600	353301.127969427	101353.264565514
1800	320067.448655719	101092.805425369
2000	293756.283451985	100738.501953806
2200	264162.792565944	100391.311255688
2400	240589.541010182	99965.4251940181
2600	221537.289922361	99817.3887479824
2800	205494.687094468	99629.7108958463
3000	192707.148822444	99605.1385057671
3200	180582.107020891	99585.2934443023
3400	167171.245373602	99558.0952060187
3600	158186.44296937	99523.3275427734
3800	150699.350806179	99523.3275427734
4000	144289.067914497	99501.6214379091
4200	138889.966849986	99483.8872785511
4400	133112.913432246	99468.4697227787
4600	130374.368178466	99468.4697227787
4800	127324.232053125	99468.4697020796
5000	124442.958821215	99467.6109972511

Generation	(Right Axis) Best Individual: Just Genetic	(Left Axis) Best Individual: Genetic and Christofides
200	2927351.44270941	95979.0698219643
400	2052251.58080071	95936.622322572
600	1676470.19152472	95805.5353500463
800	1447100.06082211	95489.3737495364
1000	1307421.50352176	95056.2718163248
1200	1184522.39370075	94925.8699915002
1400	1086455.79744563	94888.9690165811
1600	1016154.77155853	94841.4417546286
1800	932001.633986358	94774.6419337809
2000	863809.435046736	94624.3388929044
2200	796500.767322685	94438.4959880895
2400	740644.594314933	94369.142862523
2600	691812.233375989	94320.39891236
2800	643030.200774116	94093.9455421547
3000	602569.200104707	94068.7544781628
3200	571846.382069431	93992.4698822673
3400	537156.303350704	93770.3382554718
3600	505498.511486674	93616.335636831
3800	478107.361347279	93589.1330357189
4000	453560.500434714	93548.2499112858
4200	428559.793346816	93524.0301108748
4400	407493.937808355	93476.9508136421
4600	385159.107272679	93408.6827887513
4800	369668.543053302	93370.6474039174
5000	351729.696875395	93360.7690694728