1 Genetic Algorithm

1.1 Implementation

1.1.1 Reading the file

Reading of the file is identical for both implementations, so I will only mention it here. We begin by simply read the file content to a string, and removing any occurences of $'\backslash n'$ in the string. Following from this I make the string a list where elements are the values in the string that are separated by commas. We then store the name string and remove it, and store the $size$ (by ignoring non-numeric characters in the second line). We proceed to take the first $size - 1$ elements from the list, remove any spaces or non-numeric characters from them and add each of them to a new one-dimensional list. We add this new list to another list and repeat the process for the next $size - i$ elements in the list (where i ranges from $1\, to\, (size - 1)$, at which point the new (and final list) has $\frac{1}{2} \cdot (size - 1) \cdot (size)$ elements and we have successfully acquired all useful information from the file.

1.1.2 Initial Generation

We make an initial generation of $population\ size = 100$ meaning in our initial generation we have 100 (random) tours of the system. I felt this was suitable as it is necessary to have a large population size in order to incorporate diversity of tours, whilst this was also small enough that it limited the amount of memory used and the extent to which the algorithm is slowed down. We simply store this generation (and future generations) as a list of lists, where each tour is a list within the generation list.

1.1.3 Fitness

From this we continue to the next step: calculating the 'fitness' of each tour in the generation, to do this I calculated the distance of each tour and simply set the value of fitness to $\frac{1}{distance^2}$ - I stored each value of fitness in a list (in the same order as the tours).

1.1.4 Selection (Choosing Parents), Crossover then Mutation

We repeat the following process *population* (100) times, as we wish to select new parents and 'cross them over' to produce each tour (child) in the next generation.

We need some method to decipher the parents that we crossover, when engaging in the 'crossover' process. It is important that the selection of parents is bias towards fitter tours, however, we do not wish to continually crossover the very fittest tours as it is likely we will quickly reach a local minimum due to the children produced lacking diversity. As a compromise, in order to select parents I chose to implement the roulette wheel approach, this essentially means we 'randomly' select parents with bias towards fitter tours. In order to do this I set total fitness as the sum of each tours fitness, $total\, fitness = \sum_{all\, tours}(tour\, fitness)$. I then equated the selection variable for each tour to the tours fitness multiplied by 360, and divided by the total fitness $selection\, variable = \frac{tour\, fitness \cdot 360}{total\, fitness}$ - (the selection variable corresponds to an angle on the wheel). We store each selection variable in a list, and for each tour the lower bound for it to be selected is simply the sum of the selection variables prior to its selection variable, the upper bound for it to be selected is simply the lower bound plus the tour in questions selection variable. We then generate a random decimal between 0 and 360, and select the selection variable for which $lower\, bound\, of\, selection < random\, decimal_{in\, range\, [0,360]} \leq upper\, bound\, of\, selection$ is true - the tour corresponding to this selection variable is set as a parent. I repeated this process to select a second parent - generating random decimals until I selected a parent that was different to the first parent.
In order to produce 'fitter' (shorter) tours we utilise a crossover process for the two distinct parents we selected. We set a random index between 0 and the $tour\, size - 1$, we set the elements of a child (call this $child1$) as the elements in the first parent up to this index and elements in the second parent after this index. Similarly, we set the elements of a child (call this $child2$) as the the elements in the second parent up to this index, and elements in the first parent after this index. We then fix each of the children so that they do not the same element twice, in order to do this we go through the element in the child and if the element appears twice in the child we replace that element with the smallest element that should appear in the tour and doesn't.

1.1.5 Mutation

In order to increase diversity in each generation and prevent children being too similar to their parents, for each child there is 15% chance that we mutate the child. Our method of mutating a child is simply by getting two random indexes (similarly to crossover these are in the range $[0, tour\ size - 1)$), and reversing the order of the elements in the child between these indexes. This alters the child substantially, however as there is some symmetry to the tours the fitness of the tour should not be altered massively.

1.1.6 Elitism

I realised that in order to consistently improve each generation, it makes sense that we keep some of the fittest elements from the previous generation and put them in the current generation, this process is called elitism. In order to do this we simply produce a list of tuples of the form $(tour, tour\ fitness)$ - we then order this list by fitness so that then when we take the first 20% of the list, we are taking the fittest 50% of tours. We then replace the first 20% of the current generation (essentially a random 20%) with the fittest 20% of tours from the previous generation.

We have now produced a new generation and we will continue to do so in the same fashion (going back to the fitness step recursively) until the minimum length of tour in a generation, has not improved in a number of generations. If the size of the tour is greater than 100; this number of generations is equal to $\frac{3}{2} \cdot size$, if it is between 40 and 100 the number of generations is $4 \cdot size$ otherwise the number of generations is $6 \cdot size$.

The above implementation produced the following tours in around 2 minutes for the first time progressively getting longer and the last tour took several hours to acquire:

	012	017	021	026	042	048	058	175	180	535
Genetic Algorithm	56	1444	2549	1473	1197	12909	26415	21961	3590	59994

1.2 Experimentation of Implementation

In order to arrive at the above tours a lot of alteration and fine-tuning of our implementation was necessary, as many of the tours were huge in length, here we will discuss the major alterations made with respect to individual methods or constant values.
Initially I used the following equation for equation fitness $fitness = \frac{1}{distance}$ however as I progressed I altered this to $\frac{1}{distance^5}$ and found that for many of the smaller city files the tour distance converged to a unanimous minimum much more quickly than with $fitness = \frac{1}{distance}$. However, when progressing to the larger city files, this method was far too aggressive and the tour distance converged to a local minimum relatively quickly. However, altering this to $fitness = \frac{1}{distance^2}$ offered a good compromise in the fact it made it far less likely to select some of the very large tours and converged to a low tour distance more quickly than $fitness = \frac{1}{distance}$ did. Whilst at the same time it was not too aggressive and did not eliminate the possibility of larger tours being used for crossover, it only reduced it significantly.
When performing the selection of parents, initially I did not prevent the same parent from being selected twice, this was something I implemented later, and it definitely sped up the rate at which the tour values were found for smaller city files, however I could not notice whether it had much affect on the larger files, as they often took a long time (upwards of an hour) to converge regardless.
When altering the crossover process, I tried a variety of methods - some of which greatly improved the tour distances acquired. My initial crossover method began by taking two random indexes and taking elements between these indexes from the first parent and placing these elements in the corresponding positions in the child. I would then loop through the elements in second parent and for each element if the corresponding element in the child was empty and the child did not contain that element I would place the element in that position child. I then proceeded to fill any empty elements in the child with the smallest element that the child should contain but didn't. I first realised that this method should also consider the case where we first fill the child with elements from the second parent, and then fill the child with elements from the first parent - and return the fitter of the two cases (by storing each case separately). However, I further improved this method to the method described in my implementation section as this differentiates from the parents more producing more diversity in the children. I then attempted to perform this process with 3 parents, (by taking 3 random indexes) and doing a very similar process to that described above, whilst the tours were of a similar length, to the ones produced in my current algorithm there were no noticeable improvements in distance and the time taken to converge was substantially longer.

I altered my mutation method greatly throughout the development of my algorithm, initially my mutation process was simply selecting two random indexes and swapping the two elements at those indexes to mutate the child. This method of mutation was too weak, so we improved this to swap multiple pairs of indexes, this was essentially just producing a random child and didn't account for the 'symmetry' of each tour. We improved the mutation method to the method described in the implementation section, this greatly improved the tours acquired as well as reducing the time it took to acquire them. My population in the above implementation is set as 100, initially I was using a small population of 10 tours, as I increased this I noticed more diversity in both the initial and future generations thus better tour lengths were acquired I continued to increase this however as my population got larger (using populations in the region of 300) I noticed the algorithm was taking a huge amount of time to converge without much improving the distances acquired significantly thus I decided 100 was a suitable value to set my population size to.
Elitism (described above) was not a concept that I implemented initially, it was only when I noticed that minimum distance was often not decreasing from generation to generation that I thought to involve this. It aided in not having to re-compute prior tours and also meant that generations were typically fitter than previous generations prior to Elitism been. This improved both the time it took to converge and the distances of tours acquired, I also initially took the fittest 50% from the previous generation when performing elitism but after trialling different values settled on taking the fittest 20%, as this allowed for more tours of interest but still meant that each generation was significantly different to the prior generation.
The stopping condition for the algorithm was initially to simply stop after 2000 iterations, as this gave all the city files opportunity to reach a relatively low tour of the cities in question, however it meant that each city required the same number of iterations to produce the minimum. I altered this so that the algorithm halted when minimum length of tour in a generation, had not improved in a number of generations. If the size of the tour is greater than 100; this number of generations is equal to $\frac{3}{2} \cdot size$, if it is between 40 and 100 the number of generations is $4 \cdot size$ otherwise the number of generations is $6 \cdot size$.

2 Simulated Annealing

2.1 Implementation

2.1.1 Starting

We start the algorithm similarly to the way we start the Genetic Algorithm, as we read the file using the exact same method. The annealing is process is then began by producing a random tour (random list of numbers from 1 to $size$) which we store in a one-dimensional list. We set this initial tour as the 'current tour'.

2.1.2 Adjacent Tour

We formulate an adjacent tour, the method of doing this is to take 2 random indexes for the current tour and reverse the order of elements between the elements associated with these indexes. This method should form an adjacent tour as it alters the current tour considerably but it still tends to be a tour of similar length to the current tour, this is because it makes use of the symmetry of tours.

2.1.3 Probabilistically Swapping Tours

If the length of the adjacent tour is less than the length of the current tour than we will make the adjacent tour the current tour, as we are interested in the shortest tour possible. If not then we will generate a random number between 0 and 1, and if this random number is less than or equal to $e^{\frac{current\,tour\,distance - adjacent\,tour\,distance}{\log(temperature)}}$ more clearly represented as $e^{-\frac{(adjacent\,tour\,distance - current\,tour\,distance)}{\log(temperature)}}$ then we will swap the tours anyway, as this prevents the tour from getting stuck at a local minimum in terms of distance. Its clear that in order to acquire the smallest length possible we should not expect the length to decrease with each iteration of the algorithm. You may be wondering what temperature is set as, initially we will set it as a relatively high value to essentially allow free swapping this will equate to $temperature = 10 \cdot size$ where size is the size of a tour.

2.1.4 Cooling Schedule

If the current tour is swapped with an adjacent tour that is greater in length then we wish to cool the temperature (as this makes it less likely that the tour will current tour will swap with a tour of greater length in the future). The cooling

schedule is an important part of the process as I wish to make it so the temperature cools at a very slow rate so that we present plenty of opportunity for 'negative' swaps to occur and for us to avoid local minimum, with this in mind should the scenario above occur (current tour is swapped with an adjacent tour that is greater in length) then we set the temperature to $temperature = \frac{initial\,temperature}{1+\log(1+k)}$. k is simply a constant and initially 0, however each time we cool the temperature we increment k by 1.

We repeat the above process (starting from the adjacent tour method) using our new current tour rather than an initial random tour, until the temperature has cooled to an appropriate value, I set this value to $temperature = limit + 1$. The limit varies on the size of the tour, should the size of the tour be greater than 300 the limit is size + 100, if $100 < size < 300$ then the limit is size + 10 otherwise the limit is simply size. I also terminate the algorithm if over 1000 iterations have occurred as it is important to make sure termination occurs.

Using this method I was able to acquire the following tour lengths. Each of the tours took a matter of minutes to acquire.

	012	017	021	026	042	048	058	175	180	535
Simulated Annealing	56	1444	2549	1473	1187	12487	25395	21430	1950	48997

2.2 Experimentation of Implementation

The initial temperature plays a huge role in whether the current tour swaps with a 'worse' adjacent tour. Consequently, it also plays a key role in preventing the tour length from converging to a local minimum early in the algorithm - at least if it does reach a local minimum this should prevent it from staying there. Because of its importance I trialled multiple different initial temperatures, initially I tried 100, which worked well for smaller city files, and then I trialled 300 for some of the larger tours and saw that this also worked well however increasing the temperature even more so worked a little better for larger city files. Thus to me it was clear at this point that the initial temperature should differ dependent on the size of the city file, thus I set it to a simple linear function of size $initial\,temperature = 10 \cdot size$ as this seemed to fit the needs of the city files as a whole and allowed almost free swapping between adjacent and current tours initially which helps to avoid local minimum.

For the same reasons as the above, we also want to make sure that as the tour length gets smaller and progresses, we adjust the temperature appropriately in order to make sure that if we are to swap to a longer 'worse' tour then it is beneficial in terms of the overall minimum we will reach. This is decided both by the initial temperature (which we discussed above) and the cooling schedule which we will discuss now. Thus I trialled a number of different cooling schedules, the first of which was $temperature = \frac{temperature}{1+0.1 \cdot k}$ again here k is a constant which is initially 1, and as we progress with each iteration k is incremented by 1. This cool schedule produced some very good tours relatively quickly however I found that it cooled far too quickly and for the larger city files it would often converge to a local minimum more quickly than it should. From here I proceeded to use the cooling schedule $temperature = temperature \cdot (0.9)^k$ (and even altered 0.9 to other values between 0.8 and 0.9), again here k is a constant which is initially 1 and after each swap k is incremented by 1. This was a important change to the algorithm as I noticed the minimum tour lengths produced were considerably smaller in some cases and never larger, at this point my algorithm was beginning to be more refined as the changes in tour length were smaller. On the off chance it improved I altered the cooling schedule to $temperature = \frac{temperature}{1+\log(1+k)}$, here k is a constant that is initially 0 and is incremented by 1 each time cooling occurs. Whilst using this cooling schedule this produced new minimums for some of the larger city files. When I outputted temperature values I noticed that it cooled at a similar rate to the previous cooling schedule (perhaps slightly slower) thus I was unsure why performance would better but decided to implement the cooling schedule regardless, as it definitely did not effect the algorithm negatively.

The cooling of temperature initially occurred after each iteration, however after contemplating this further I decided that I would try only cooling the temperature if the current tour has swapped with a 'worse' longer tour as we should make it less likely for such a swap to occur after such a swap does occur however if such a swap has not occurred we are not necessarily concerned with altering the temperature. After implementing this I noticed it did somewhat reduce the distance of tours acquired and hence incorporated it in my final implementation.

When producing an adjacent tour initially we simply reversed the order of any two successive cities in order to find an adjacent tour, I altered this to the method described above (which reverses the order of a random amount of successive cities). This is a more aggressive method but implemented alongside our cooling schedule helped to massively speed up the time it took to converge to a low distance, admittedly I did believe this would perhaps incorporate too much diversity in the adjacent tour from the current tour but the results certainly disagreed.