Algorithm A : Genetic Algorithm

Results:

Tour length	Best Score	Overall Best
12	56	56 (GA/SA)
17	1444	1444 (GA/SA)
21	2549	2549 (GA/SA)
26	1584	1476 (SA)
42	1370	1254 (SA)
48	13010	13010 (GA)
58	26403	26403 (GA)
175	22543	22543 (GA)
180	1970	1970 (GA)
535	50957	50957 (GA)

Implementation:

The first implementation decision was the storage of distances between pairs of nodes. Due to the nature of each transition value being associated to a start and finish node, it was sensible to implement the values in Matrix form, which I implemented as a list of lists. I also included an empty column and row at 0 indexes, so that references to the matrix didn't need correcting and a reference M[i][j] is the distance between cities i and j. The matrix top diagonal is also mirrored to the bottom diagonal so that the inverse M[j][i] reference yields the same result.

A 'for' loop of a determined size is used for creating the initial population, which I could vary for optimum results. I used the 'random' import in python, which allowed me to use a random 'sample' of the list of cities, essentially reordering them randomly to create a tour. The tour cost is calculated by a 'for' loop through the list of nodes which adds to the tour score the cost of progressing from the current node to the next in the tour. The tour score and tour are added as a tuple to the genepool list. This was implemented as a list of tuples as opposed to a dictionary or two separate lists so that it could be reordered by size (referencing only tour cost) and indexed for the reproduction operator, which was not possible with other implementations. This allows access to the best X% of results easily.

A 'bestRoute' variable is used and is initially set to the minimum costing parent from the initial population. At the end of each generation, the lowest costing tour from the new generation is compared to the overall best route and if superior, replaces it as the stored cost and tour tuple. This is implemented because there is a chance that the best tour is never chosen by the reproduction operator, and therefore a tour equal or better may not progress to the next generation. Therefore, the variable ensures the best solution found is always the one output by the algorithm.

Random.random() is used in the reproduction operator to create a decimal between 0 and 1. It introduces the probability. The ordered genepool is split into 4 by index (helped by list of tuples implementation), and uses a comparator to select parents where it is more likely for better tours – if the random variable < 0.5, choose bottom quarter, then if < 0.75 etc. This means better parents are more likely selected, while keeping an element of chance in prospective children.

For the crossover operator, a new list is created containing all cities in numerical order. Cities already visited by the child while it is being created are removed from the list, and therefore it is a list of the vertices the child is yet to visit. I used this list because without it, children which are illegitimate could be created with repeated nodes – ie non-viable solutions.

My crossover operator is a method which creates children whose cost is at worst as good as its best parent, and otherwise better. It begins from a node, selected using 'random.choice', because though tours are circular, so have no start, starting always from one node would affect which order the other nodes are selected, so when the best route has been stagnant for multiple generations, this additional change increases the chance that an improvement will appear in a new child.

Once a child is created in the crossover operator (detailed in experimentation), the mutation operator introduces a chance that the child might randomly switch a pair of nodes. Again 'random' is used, and an if statement checks if it's below a set value, in which case; mutate. The set value is variable so that it can be altered when experimenting to find the best results.

After the mutation has or hasn't taken place, a new child is created as a tuple of cost and tour. I decided that if the mutated cost is less than or equal to the previous cost, it is preferred. Therefore,

if a child mutates to create a worse tour it is not accepted, because the mutation is seen as disadvantageous, and better children are a priority. This also allows for the use of a high probability of mutation, because it removes the risk of ruining tours, increasing the chance of finding a good mutation. Equal mutations are preferred to the child because they introduce difference without being worse, which is good for the genepool. The implementation issue which arose here was that the copy tour which was mutated and compared also altered the original, causing false results where the mutated tour continued with an incorrect cost regardless. The solution to this was to create a 'deepcopy' tour to mutate, which doesn't simply point to the original, but is a new object with identical values.

The final implementation decision relates to writing to a result text file. To avoid overwriting a good or even optimum result with a worse one, the stored result is read and compared to the result of the current execution of the programme, and the file is only written to if the result is superior.

Experimentation:

Both the size of population and number of generations are key variables which need experimenting to get the best results for different AISearch files. With the smallest problem, the genetic algorithm efficiently and consistently achieved good results with smaller populations for less generations. (In as few as 5 generations, beginning with a population of 200). For larger tour files, greater sizes of genepool and number of generations are required. I experimented with the number of generations and discovered that often the algorithm plateaued after a number of generations, meaning that often 50 was more than enough to reach a good result.

However, a trend emerged whereby a few of the files, presumably due to the way the graphs were laid out, didn't follow the usual pattern of improving rapidly to a good result, then showing very little and rare improvement beyond that value. In a couple of cases the best score decreased consistently, without an initial large decrease, and required a larger number of generations to get the best results.

I experimented with the reproduction operator, initially prioritising quality of offspring above diversity. Therefore, I tried a method whereby the best and second best tours were selected as parents, bred, and the child added to the next generation, while the parents were removed and the next top 2 tours were bred. The results from this method seemed initially promising for the 12 city tour, quickly reaching a result matching the best the algorithm had achieved, however this didn't continue for the other tours. The algorithm found decent results faster than other methods, because while the better tour was different it produced strong offspring, however without an element of chance or introduction of new genetic material, the result was all children become copies of the best result from before, and the algorithm doesn't progress.

I switched to a method I had come up with, which like many reproduction operators uses probability distributed so that better tours are more likely to be selected. Having done the experimentation with breeding the best tours only, I felt it was better to make a section of the parents more likely to be selected, as opposed to the best tour x% likely and so on (Roulette wheel format). Instead my algorithm divides the genepool into quarters based on quality (top 25%, middle etc). I then experimented with choosing how to distribute the probability. I found that an efficient option was to select the best 25% with p=0.5, then then next 25% with p=0.25, the next with p=0.15, and the bottom quarter with p=0.1. Then, a tour parent is randomly selected from the quartile. With this method, better tours are preferred but there is a chance of worse tours being used, and there is not as much stagnation because the current best tour is not necessarily chosen, nor is it more likely than any other in the best quarter.

The crossover operator was the area with which I experimented most. I began by creating the algorithm from the slides; selecting the first half of parent A and the second half of B, and combining. The correction I used for duplicates was to create a list of missing nodes from the tour, then find all second instances of nodes and replace them with the numerically lowest unvisited node. This method created children which held little similarity to the parents and more closely resembled a new random

tour. I felt that this wasn't an efficient crossover, and looked for one which uses the parents more effectively and is less likely to produce worse children than the parents.

I expanded on the splitting method, splitting into quarters and calculating their cost, selecting the best first, second, third, and fourth quarters to create better offspring. This method avoided much worse children, because in some cases it can simply replicate the better parent, but it didn't work well because lots of nodes were repeated, and so replaced with available nodes which defeats the attempt to find the better parts.

Clearly using large sections of each parent causes problems because of large numbers of repeated nodes, so going for a node-by-node approach seemed like the best direction to experiment in. I decided to attempt a crossover algorithm which starts on a node, which I selected at random, and selects the next node from either parent A or B based on the shorter route: starting on 5, if parent B includes the step 3,7 of cost 4 and parent B includes the step 3,2 of cost 7, parent A's DNA would be selected, and the new child would now be 3,7. This continues until the child is complete; that is the list of available nodes is empty. The list of available nodes was important because sometimes a parent's progression from a node will go to one which already exists, and would create an invalid route. Therefore, the algorithm checks if the next node has been visited, and if it has that parent's 'choice' is replaced by the numerically lowest available unvisited node, to then be compared to the step in parent B. While it is not the cheapest crossover function in terms of computation time, the results of this crossover are very good, and give a high chance of producing superior offspring, especially in terms of passing on to children the best attributes of the parents.

I also experimented with mutation probability, and was able to raise it very high because if a poor tour was created it wasn't accepted, although this slowed the algorithm.

Algorithm B : Simulated Annealing

Results:

Tour length	Best Score	Overall Best
12	56	56 (GA/SA)
17	1444	1444 (GA/SA)
21	2549	2549 (GA/SA)
26	1476	1476 (SA)
42	1254	1254 (SA)
48	14010	13010 (GA)
58	26503	26403 (GA)
175	25413	22543 (GA)
180	4310	1970 (GA)
535	76189	50957 (GA)

Implementation:

The matrix storage was the same as for the genetic algorithm for the same reasons.

It was not necessary to store the current tour as a cost; tour tuple, because there was only ever one current tour, however for efficiency and neat-ness the best score was stored as a tuple.

The best tour variable tuple was kept and updated to hold the best result achieved by the algorithm at any point in its running. This is because there is always a possibility that the tour could mutate, and create a worse tour that never exceeds the best from earlier during the algorithm. It is especially significant as the tour could very poorly mutate just before the algorithm finishes, and without a running best score, this would return a poor result which doesn't reflect the previous success of the algorithm.

For the iterative part of Simulated Annealing, I chose to implement a 'while' loop, which served as an escape for the algorithm if the temperature 'T' fell below a certain value, which I experimented with as well as the start temperature.

I used 'import random' so that, in order to switch two nodes in the current tour during the algorithm, two new variables can be set to random integers in the range of the length of the list and would give two indexes for nodes to swap. This gave a pseudo-random node swap which was required for the algorithm.

In order to keep the old route for comparison to the new one, a copy of the route is created. Because of the way python maps copies, with changes to the new copy also applying to the original, a copy is made as a new list with identical elements as new = old[:]. Next, the change is made to the

new tour, where the two indexes on the new tour exchanged with the opposite indexes on the old tour.

A 'for' loop is used to calculate the cost of a tour. For each node in the tour, the cost from node k to node k+1 is added to the cost variable, initially set to 0. The last node loops back to the first using an if statement checking that it is the last node.

For Simulated Annealing, the probability is defined by e^(change in cost/Temperature). Therefore, it is necessary to import 'math', so that math.e can be used to create the probability. 'If' and 'elif' loops are used in determining whether the old or new tour is accepted, as per the algorithm. The implementation here is the same used above, where old = new[:] if a random decimal rd.random() is less than P or if the new tour is better than the old one.

To implement the cooling schedule, I chose to decrease T marginally at each iteration of the while loop until T passed below the end value of 1.

The tour files are opened in the same way as for the genetic algorithm, both at the start of the programme, and for the tour file at the end to check that the new tour is superior to the previous best.

Experimentation:

I used a linear cooling function, where at the end of each iteration I used T = T – x. However, when experimenting with more effective functions, I found that using a function T = xT where 0<x<1 gave better results. This is because when the function reaches lower temperatures, it is beneficial to stay longer on them, so that the algorithm reaches the best 'local' minimum, whereas a linear function decreases too fast when the temperature is low and so doesn't reach the best scores.

Experimenting with different start values of temperature and cooling functions showed that small changes to the rate of cooling and start T had a large effect on the outcome of the algorithm.

Once I had settled on a cooling schedule where T decreases quadratically, I experimented with varying values for the rate of cooling. I began with 0.9, and found that the decrease was too abrupt; even with high starting temperatures, there wasn't enough time spent at lower temperatures to obtain good results. Therefore, I increased the value to find better results. I found that for the smaller tours, a steeper decrease such as 0.99 sufficed, and saved time. However, for the larger tours this failed to reach the optimum, as the score would decrease right up until the algorithm ended. Therefore, I increased the value to 0.999+. This dramatically increased the time taken for good results, but also increased the quality of results. I concluded that larger values obtained better results, but could also greatly increase the time taken; adding one '9' to the end of the value could exponentially increase the time taken.

As well as the cooling schedule, I experimented with the start temperature of the annealing process. Initially I started with the start temperature far too low, and as it was increased I found to a lesser extent a similar pattern as had occurred the cooling schedule; a higher value meant better results but more time taken. However, there is a point at which any further increase in T did not reflect in better results, but more time consuming ones. T=100000 proved to be a reasonably successful choice for many of the tour sizes.

I also experimented with the value at which the while loop ended – the minimum value of T. Many of the tour sizes would produce good results with an end value of 1, but decreasing the end value allowed the algorithm to reach better tours, because the change from 1 to 0.1 is very significant when the value is decreasing at a rate of 0.9999T. It allowed the algorithm to approach a local minimum with almost 0 chance of jumping to a worse state, and therefore fully exploring that minimum.

I also experimented with swapping multiple pairs of nodes, however this didn't improve results, perhaps because using individual swaps achieves the same results in 2 iterations, but if one is an improvement and the other makes the tour far worse, with single switches (ie. the minimum change), the good switches are preserved and used.