AI Search Assignment Report

Best Path Lengths

No. of cities.	12	17	21	26	42	48	58	175	180	535
SA	56	1,444	2,549	1,473	1,187	12,160	25,395	21,434	1,950	48,583
GA	56	1,444	2,549	1,473	1,187	12,253	25,395	21,730	2,240	48,957

Genetic Algorithm

Initial Implementation

- I store the 'population' in a dictionary, with the path (a list of city numbers) as the key and the path length (an int) as the value. This ensures that there are no duplicate tours, which helps to maintain diversity, as well as being an efficient way of mapping the path to its length.
- Initially, I would start each iteration by removing the longest x % of paths from the population (where $x = \textit{death rate}$). To sort the population by fitness, I convert the dictionary to a list of tuples; the list can then be sorted.
- Next, 'breeding' would take place, with parents being selected completely randomly.
  - My implementation of crossover was as follows: pick a random index i; copy the first i cities from parent1 to child1 and the first i cities from parent2 to child2; for child1, iterate through the cities in parent2, appending those that aren't already in child1's path, and vice versa for child2 and parent1.
  - Mutation is performed on each child with a probability equal to a given parameter (*mutation rate*). This was initially implemented by randomly selecting two cities in the path, and swapping them.
- The population is then converted back to a dictionary, before the children are added.

Experimentation

With a population size of 100 and 100 iterations, I experimented with some parameters to get the results in figure 1. Note that all figures apply to the shortest path found for the 535 city tour.

Mutation Rate	Death Rate	Shortest Path
0.01	0.70	139316
0.01	0.80	137783
0.02	0.80	137783
0.05	0.80	131229
0.05	0.90	131179
0.06	0.90	129467
0.07	0.90	129467
0.10	0.90	123323

Figure 1

Tournament Size	p	Shortest Path
0.5	0.60	123840
0.5	0.75	124305
0.5	0.80	121149
0.5	0.90	125505
0.2	0.80	125672
0.7	0.80	124984

Figure 2

Mutation Rate	Death Rate	Shortest Path
0.20	0.80	122064
0.20	0.70	120518
0.20	0.50	123542
0.10	0.70	127395
0.20	0.70	120518
0.40	0.70	117833

Figure 3

These results weren't getting much better by just changing parameters, so I decided to alter the algorithm. At this stage, the parent selection for breeding was completely random. I changed it here, so that the best members are more likely to breed, by implementing 'tournament selection'. I select a fraction of the population (*tournament size*) and let the 'fittest individual' have a probability p of winning the tournament. If this path is not selected, the second shortest path is selected with probability $p(1-p)$, then $p(1-p)^2$, $p(1-p)^3$, etc.. Two tournaments are held, with the winner from each going on to form a breeding pair. Experimenting with *mutation rate* $= 0.10$ and *death rate* $= 0.90$, I tweak the parameters used in the tournament selection to get figure 2. I wasn't able to get a great deal of improvement here, so I tried altering the first two parameters with *tournament size* $= 0.5$ and $p = 0.80$ to get figure 3.

At this point, the longest x % of the paths were being removed at the start of each iteration, with the left being rest to breed. Instead, I tried replacing the entire population with the offspring of breeding, thinking that worse members of the population now have a better chance to breed, increasing diversity which should help avoid local minima in path length. However, it seems that the cost of potentially losing the best path in a given generation was too great. After implementing this I tried tweaking parameters, but the best path length I could get was 162450, so I went back to the old, elitist, implementation.

When breeding a new member for the next population, I produce two 'siblings'. I tried selecting only the fittest of these siblings and discarding the other, which I thought should fill the population with shorter paths, thus leading to better results. However, I was again unable to get results as good as previously attained just by tweaking parameters; the best I got was 125469. When I went back to the old implementation and played with all of the parameters again I managed to find a path length of 114680, and when I increased the iterations and population size from 100 to 200, I found a path length of 101012.

Next, I tried to change the mutation method to increase diversity, which I thought should help prevent the GA converging too early. Instead of making one mutation (swapping one pair of cities) if the probability allows it, I tried decreasing the probability, but applying it to each city in the path, potentially swapping them with another random city. Again, this made things much worse, so I went back to the previous mutation method and turned my attention to a different area of the algorithm.

Instead of just removing the bottom x % of the population at the beginning of each iteration, I tried removing the less fit members in a stochastic manner. To do this, I sort all of the paths from shortest to longest. Each path now has a normalised ranking, given by their position in this list divided by the length of the list. Each path has a probability equal to the square root of its normalised ranking of being removed from the population. With this method, I believed the diversity should remain higher, allowing better path lengths as local minima can be avoided more effectively, but I still have some form of elitism, which should ensure that the best path does not get worse as the algorithm progresses. I thought that this would take a while to demonstrate, so I began testing with a larger population and more iterations. With both of these parameters set to 500, I got the results shown in figure 4. The 'power of ranking' parameter is what was initially the square root, i.e the power to which the normalised rank is raised to in order to obtain the probability of a path being removed.

Mutation Rate	Tournament Size	p	Power of Ranking	Shortest Path
0.8	0.6	0.8	0.5	83442
0.8	0.6	0.7	0.5	83327
0.7	0.6	0.7	0.5	81306
0.7	0.6	0.7	0.4	82745
0.7	0.6	0.7	0.6	84431

Figure 4

The final breakthrough in improving the algorithm came from a tweak I made to my SA algorithm, which I thought I could incorporate into my GA. When a path is undergoing mutation, once I've selected two random cities, instead of just swapping them I will reverse the order of the sub-path that they contain. As predicted, this did indeed improve the algorithm. It seems that this more significant mutation increases the diversity of the population, and allows the algorithm to more effectively explore the problem space.

When running with 500 as the parameter for population size and number of iterations, I got a best path length of 64025. This was only running for the amount of time it took for me to make a cup of tea, and it returned a better result than when I allowed the previous implementation to run overnight. When I ran it with half the population and twice the iterations (250 and 1000 respectively), the GA returned a best length of 61063. From here, I was able to achieve the path lengths given at the beginning of this report just by increasing the population size and iteration number, and leaving the algorithm to run for longer.

Simulated Annealing

Initial Implementation

- The probability of accepting a neighbour as the new path is given by $e^{\frac{current\_length - new\_length}{temperature}}$.
- The temperature is decreased by multiplying it by a constant (1 - *cooling_rate*) after every iteration.
- To begin with, I defined a neighbour as a path which can be found by swapping the positions of two random cities in the current path.
- I halt the algorithm when the temperature is less than 1.

Experimentation

Initial experimentation with parameters gave the results in figure 5.

Start Temp	1000	5000	10000	10000	10000	10000	1000
Cooling Rate	0.0010	0.0010	0.0010	0.0010	0.0005	0.0001	0.0001
Best Path	99603	100148	99704	130948	92502	77764	79086

Figure 5

At this stage I was only evaluating one neighbour path at each temperature. I introduced a parameter called ‘tries’, which is the number of times (at each temperature) that I will pick a random neighbour state, and check if the probability function will allow it to replace the current

path. The algorithm will continue to consider new neighbours until either a successor is found, or this number of tries has been reached.
Experimenting with the optimum parameters from before (start temperature of 10000 and cooling rate of 0.0001), I tried some different 'tries' parameters, giving the results in figure 6. This addition significantly increases the runtime of the algorithm, but is clearly leading to better results.

Tries	1	5	10	20	100	200
Best Path	77764	68487	66111	63252	60511	59370

Figure 6

Next, I changed the algorithm so that instead of considering x random neighbours at each temperature, the algorithm goes through all of the possible neighbours until a successor is found. It was immediately clear that this leads to worse results. This could be because the same cities (those earlier in the path) are repeatedly being swapped back and forth, before swaps on the later cities can be considered. For that reason, I tried generating all of the possible swaps, shuffling them, then going through that list instead. This additional computation meant that the algorithm became extremely slow to run, so to compensate I tried changing the cooling schedule to a linear decrease, instead of an exponential decay, i.e. temp -= cooling rate instead of temp *= cooling rate.
Again, this was still not as good. I got a result of 147287 on the first run with a start temperature of 10000 and a linear cooling rate of 5. However, I wasn't sure if this was because of the different implementation, or the different cooling schedule, so I decided to let the algorithm run on the old cooling schedule to allow a more accurate comparison. After approximately 5 hours of run time, this method only achieved a path length of 62160, which was still longer than the best length achieved by the previous method, so I decided to go back to it.

In an attempt to improve the run time, I added a catch where if a new state hasn't been selected in 10000 tries, the program terminates. I thought that this should allow me to implement more computationally complex algorithms without having to wait as long to check the results, but it turns out that a value of 10000 doesn't terminate the program very quickly at all, so I tried 500 instead. While this did terminate the algorithm earlier, thus saving time, it gave a result of 67014. Terminating the algorithm early enough to notice an improved running time comes with too great a cost to the best path length, so I removed this feature.

The final improvement I made to the algorithm is the one that also led to an increase in the genetic algorithm's efficiency; trying a different neighbour function. Instead of picking two random cities and swapping their positions, I tried selecting two consecutive cities. This was unsuccessful. However I then tried picking two random cities, and reversing the order of all cities in between them. This drastically improved the algorithm, and from here I was able to obtain the final results simply by tweaking the parameters for each set of cities. Primarily, this just involved increasing / decreasing the end temperature and cooling rate, to alter the running time as desired. For example, to get my final result for the 535-city tour, I had to decrease the end temperature from 1 to 0.2.