Software Methodologies

AI Search Coursework

(A) Simulated Annealing

Best Results

Size	12	17	21	26	42	48	58	175	180	535
Tour length	56	1444	2549	1473	1055	12224	25395	21441	1950	48768
Time taken (s)	15	13	13	13	13	13	13	15	16	23

Implementation

The implementation of simulated annealing (SA) code was done using a variety of different data structures and representations. A separate class for the code related to the actual SA was created, in order to differentiate between the code required by the method and the code that was static.

There is a certain starting temperature, which was decided to be the square-root of the sum of all the distances in the current city file. The cooling rate was left at 0.99999 as this seemed to give the best values. The algorithm continues to run until a certain number of iterations have been completed, in this case 2 million was generally used.

For storing the distances between cities a 2-dimensional array was created where position [i][j] was the distance between city i and city j, while tours were stored in a list. Other variables were generally stored as integers and strings, to make manipulations easier.

The initial solution is found by selecting a random starting city, and then randomly choosing the next city from a list of the remaining cities. Two random cities within this solution are then chosen and the order of all these cities, including the 2 chosen, is reversed. This new solution is then checked and compared with the previous solution. If it has a greater fitness, then the new solution becomes the current solution and the process is repeated. Otherwise, the new solution is only accepted using the probability function $e^{\frac{\Delta E}{\tau}}$, where ΔE is the difference between the fitness of both solutions, and τ is the current temperature.

The entire process is looped using a for loop outside of the SA code, in order to allow for multiple attempts at each city file, as a different initial tour may result in better results, so only the best of these multiple results is output to the file.

Experimentation & Results

Originally the code was extremely inefficient, with a much slower algorithm being used, and far less iterations being carried out. I had originally implemented a method which looped through the program N times, where N was the number of cities, and each time created an initial tour by selecting a greedy tour starting from every city. While this did get better results than using random tours at first, it was much slower as had to run everything an additional N times. This meant that the time taken was exponential, with 535 cities taken around $3\frac{1}{2}$ hours to complete.

Size	12	17	21	26	42	48	58	175	180	535
Tour length	56	1444	2549	1473	1056	12330	25395	21441	1950	49269

This used an initial temperature of the square root of the number of cities, and a cooling rate of 0.999, and continued to iterate until the temperature was equal to or less than 0.000001.

In an attempt to improve, and avoid entering into local minimums, the code was altered to start at a random city and select a random tour rather than a greedy tour. This allowed for fewer loops of the entire code, by reducing the number of times the code had to be run from N down to a set value which could be altered, which meant that it was able to run much faster, drastically cutting down the run time from the previous version. However, because the random tours were often extremely long and the number of iterations was still relatively low, this got worse results.

Size	12	17	21	26	42	48	58	175	180	535
Tour length	56	1555	2769	1503	1084	13403	26235	21739	2000	50238

Upon creation I had already implemented reversing the entire order between two randomly selected cities, so I then altered this to instead simply select 2 cities and swap them, but this resulted in even worse tours, so I went back to reversing instead.

Size	12	17	21	26	42	48	58	175	180	535
Tour length	56	1753	3084	1720	1386	16601	27666	22327	5450	50538

The next attempt involved changing the initial temperature of the system. Two different versions were implemented, one where it was constant at 100, and one where it was determined by square-root of the sum of all the distances in the current city file. Both of these temperatures were found to be much more effective, and a cooling rate of 0.99999 was also implemented. To accommodate for this increase in temperature I put in an iterations counter, and rather then stopping at a certain temperature I stopped after a certain number of iterations. This was still extremely slow, however, so a large number of iterations was unreasonable and as such no good tours were able to be found.

Finally, because of this, I began trying to cut down on my run time rather than trying to improve my results in a different way. I noticed that whilst running, every iteration had to calculate a new distance once the cities had been reversed, and I did this by going through the entire tour, and checking the distance between each city using the array of all distances, and adding this all up to get the tour length. This meant that every iteration there was an additional loop of N values. Upon closer inspection it came to light that by reversing everything between the 2 cities only 2 distances would change, the city i-1 would now go from itself to city j (instead of i), and the city j+1 would now be travelled to from i (instead of j). As such, instead of completely recomputing the distance, I was able to simply remove, from the total length, the two distances that had been changed, and add the new distances to the total. This required some alterations in order to function properly, but it drastically cut down run time, to the point where it is now, which is that the code is able to run 2 million iterations for all city files and get what appear to be optimal or close to optimal results, in 10-20 seconds, as seen in the best results table above.

Because the code was so much faster I was able to loop it 10 times and still only take around 2 minutes for every city file, in order to get average results to show that it generally gets good results every time, rather than bad results with random lucky short tours.

Size	12	17	21	26	42	48	58	175	180	535
Best	56	1444	2549	1473	1055	12224	25395	21441	1950	48768
Average (From 10)	56.0	1444.0	2551.5	1473.1	1057.1	12461.2	25455.8	21545.9	1957.0	48909.1

(B) Genetic

Best Results

Size	12	17	21	26	42	48	58	175	180	535
Best	56	1444	2549	1497	1229	17842	33748	33623	232050	114461
Time Taken	9	28	12	12	14	14	17	39	36	223

Implementation

The Genetic code was primarily implemented using a number of 2-dimensional lists. A starting population was created, which was a list, in our final implementation, of size 1000. Each entry in this list was another list of size N, where N was the number of cities, and each of these was randomly generated upon initialising the Genetic class. The tour with the longest length is also stored as the worst tour in the population, for use later.

All static functions, such as reading and writing files and creating the array of distances between each city, were the same as those used in our Simulated Annealing code, as these did not need to change. Other common variables were also the same, such as storing of the best tour and it's length.

The function for calculating the initial distance for every member of the population simply ran through the population list, and for each one calculated its' tour length from the first city to the last, using the distance matrix that was created. This method was also used for calculating the distance after crossover took place, which will be elaborated upon below. When mutating the new tours, the method used in Simulated Annealing, which involved removing and adding back only the two edges that had changed.

In order to create new generations of tours, two random parent tours are selected from the population. This is done by applying a normalised fitness to every member of the population, such that the array containing all of the normalised fitnesses adds up to 1. A random value between 0 and 1 is then generated, and the value of each member of the fitness array is subtracted from this random value until it is less than or equal to 0. This method allows for shorter tours, which have a greater normalised fitness as the fitness was calculated as $\frac{1}{Tour\ Length}$, to be more likely to be chosen. The crossover mentioned above was implemented as selecting a random starting and ending point from the first parent, and putting it into the new child in the same position. The rest of the empty spots were then filled in from the second parent in the order that they appear, provided the city does not already appear in the child.

After crossover is completed, the new child is then mutated a set number of time, 100 in our case. The best mutation length found is then returned back. At this point the code checks if the final tour is better than the worst tour in the current population. If it is then it replaces this tour, the next worst tour is found, and the normalised fitness array is generated anew with the new changes. This will loop P times, where P is the population size, and the population is constantly updated as the program runs due to it not separating parent and child generations, but instead allowing them to create new generations together. This entire process would then loop a set number of times, with the idea of moving through generations, which allowed for a smaller population size.

Experimentation & Results

Our code began extremely inefficient, and got very lengthy tours. This was because a population of 3000, and 10 generations was used, and the population that was able to create children was only updated after 3000 children had been created using the previous population, which is a more

generational approach. Not using the faster method for calculating distance also meant that our code was very slow, as seen below. We also had a random chance that mutations could happen, and was dependant on a given mutation rate.

Size	12	17	21	26	42	48	58	175	180	535
Tour length	67	2219	4314	1998	2008	32850	84020	44032	668510	147672
Time Taken	228	247	277	230	261	274	252	386	392	1498

In order to speed things up the more efficient method of calculating distances was implemented, which dramatically improved run time, therefore allowing us to increase the population size to 4000 and have 20 generations. This, in turn, meant that we were able to get much shorter tours than before, and in less time. At this point we also implemented a more steady-state approach, where parents and children were allowed to generate new children together, by implementing the removal of the worst member of the population whilst the code is still generating new children, as this seemed to give better results.

Size	12	17	21	26	42	48	58	175	180	535
Tour length	56	1678	3181	1677	1755	25571	57850	41145	500890	139582
Time Taken	189	184	185	189	195	197	217	294	292	1026

At this point we made an adjustment to the mutation code, such that while mutating, it checked each mutation and stored the length of the best one, so as to only return the best mutation of all of those that were found for the current child. We did this with a population of 5000 and 20 generations as well as with the original 4000 population, and got better, albeit slower, results with the larger population.

Size	12	17	21	26	42	48	58	175	180	535
Tour length	56	1592	3254	1635	1612	24318	52607	40034	476880	137541
Time Taken	296	302	310	314	331	335	357	463	463	1403

At this point we realised that because of what we had previously implemented the need for a mutation rate was essentially obsolete, as if only the best of the random mutations was returned there was no reason to not have a random number of said mutations, so this was removed. We also attempted a similar experiment as before, but in the opposite direction. This was in the sense of a much lower population at 1000, but a larger number of generations at 40. This not only ran faster due to a smaller population, but also appeared to give much shorter tour lengths.

Size	12	17	21	26	42	48	58	175	180	535
Tour length	56	1461	2966	1631	1533	20873	46345	38677	406160	132024
Time Taken	75	83	83	84	92	92	90	174	200	922

As a final experiment we ran the code with a population size of 10,000 and only 1 generation. The lack of generations should not have had much effect due to the population being constantly updated during the running of the program. This, however, gave us much longer results than anticipated, and had quite a slow run time. In light of this and the results from the previous experiment we also tried running with various small populations and large generation numbers, and got the best results that were seen in the Best Results table.

Size	12	17	21	26	42	48	58	175	180	535
Best	56	1444	2549	1497	1229	17842	33748	33623	232050	114461
Time Taken	9	28	12	12	14	14	17	39	36	223