Technical Report

CT421 Assignment 1

Selection Mechanism (Tournament selection):

For my design I used tournament selection to choose the parents for the crossover operations. It works by selecting a random portion of individuals from the population and uses the fitness function to determine the fitness of these individuals. It chooses the best performing individuals in terms of fitness to be selected as parents. I chose this selection mechanism as it would be easy to implement and also easy to vary the k parameter which select the number of subsets of individuals from the population that are randomly selected. This allows for the selection pressure to be easily adjusted when using the algorithm.

Crossover operators:

For my design I chose ordered crossover and partially mixed crossover as my crossover operations. The main reason for me choosing ordered crossover was after reading "Geneticalgorithms_Lugeretal.pdf" in particular section **12.1.3** in the lecture notes. It was easy to visualize the variation in chromosomes from one generation to the next and how it would possibly improve the algorithms performance. It was also intuitive and easy to imagine how it would be implemented in code using array manipulation. The principal of the ordered crossover is that a segment of genes is selected from one parent and inserted into an offsprings keeping the sequence of genes in the same order. This segment size is defined in my 'genetic_algorithm' function as 25% the number of cities in the specific problem. I varied this during the experimental phase of the assignment. The same thing is then done again for the second parent and the other offspring. Then for each offspring we loop through the parents sequence in order and for any gene not already present (marked by a -1) it is filled in at the next available position. This gives two offspring representing a valid path.

My main reason for including partially mixed crossover (PMX) was largely down to the fact that it offers more of a chance that the parents features would be preserved than that of ordered crossover. If good parents were selected in the tournament selection this would be beneficial and my logic was that a combination of these operations together would provide perhaps some novel solutions.

Like ordered crossover in PMX two crossover points are randomly selected in each parent creating a segment to be swapped. The segment from one parent is placed directly into the corresponding positions of the offspring. To avoid duplicates a mapping is created between the swapped genes in the two parents. When filling in the remaining positions from the other parent if a duplicate is found the mapping is used to substitute the conflicting gene with its corresponding match. This process is repeated for both parents resulting in two offspring again representing valid paths.

It is possible to favour one of the crossover operations over the other or eliminate one completely by altering the set value that determines if the specific crossover operation occurs or not. An example of how this is done is explained in the setup instructions.

Mutation operations:

As I have never implemented a genetic algorithm before and although I understood the concept of mutation I didn't really know how much of an effect it would have on the performance of the algorithm exactly. I decided to keep things simple and go with the two mutation methods mentioned in "Geneticalgorithms_Lugeretal.pdf" and see for myself.

The two mutation operations I used were 'Shift' and 'Inversion' mutation. The principal of shift mutation is rather simple, it simply randomly selects a gene and randomly places it in a new position within the chromosome. Whilst testing that my implementation was working correctly (on a very small list) I realised that I had failed to account for when the gene being randomly selected and the position in which it is to be swapped to could be the same and no mutation would actually occur. To prevent this I set a while condition to continue shifting whilst the gene position and the new position were the same . Although simple it introduces diversity into the chromosome by creating variations not already present in the population and I found it was effective of escaping local optima when experimenting with different parameters such as on small populations. Inversion mutation works by inverting random sections of genes. Similar to shift mutation it preserves the overall gene set but has the possibility at introducing variations that prove helpful such as escaping local optima and also such as this case introducing new sub routes which helped introduce diversity within the population.

After spending sometime experimenting with the different mutation operators , I found the optimal mutation rate to be around 0.35. When set lower the algorithm was occasionally getting stuck in local minima and when set higher I would see the fitness converging towards worse solutions as the number of generations increased. This is likely due critical sequences genes being altered into un beneficial permutations.

Similar to the crossover operations the probability of either one of the mutation operators occurring can be varied. This is explained in the setup instructions

Elitism Strategy :

Initially my algorithm didn't implement any strategy other than tournament selection that promoted the survival of good solutions. Although I was able to improve my solutions by experimenting with parameters such as crossover , mutation , and tournament selection rates this varied significantly and I often had to use very high mutation rates , 0.6 – 0.8 and large populations which increased the computation time significantly in order not to get stuck in local optima and often they still did. To counter this I introduced an elitism strategy which ensures that a number which is defined as a percentage of individuals with the best fitness, the elite individuals are carried over in to the next generation. The strategy works by calculating the fitness of each individual in the population through the fitness function, the individuals are then compared with one another based on their fitness and sorted in ascending order into a list containing the tuples **,(individual,fitness)**. There was an immediate and significant improvement in the algorithms performance rate in faster convergence and improved solutions.

Experiments:

I initially began my experimentation on the mutation rate with the other parameters given the following fixed values. I ran 10 trials for each mutation rate and calculated the average best fitness found. The probability of either mutation occurring is set equally at 0.5. This was done for Berlin.tsp

pop_size = 100

max_generations = 500

crossover_prob = 0.3

tournament_size = 3

elite_rate = 0.01

Mutation Rate	Average Best Fitness found
0.01	11366
0.02	1012
0.03	9593
0.04	9358
0.05	8957
0.06	8856
0.07	8766
0.08	8807
0.09	8761

As the mutation rate was clearly increasing for small increments I began using larger increments

Mutation Rate	Average Best Fitness found
0.12	8513
0.14	8270
0.18	8303
0.22	8435
0.26	8284
0.3	8143
0.5	8093
0.7	10672
1.2	9220
1.4	8938
2.0	13963
2.2	14122

After a mutation rate of 0.5 there no longer appeared to be any improvement. Exploring in around this range I found the optimal mutation rate to be 0.35.

I then performed the same experiments for the crossover rate with the updated mutation rate set to 0.35. The probability of either crossover operation occurring is set at 0.5 so both are equally likely.

pop_size = 100

max_generations = 500

mutation_prob = 0.35

tournament_size = 3

elite_rate = 0.01

Number of trials for each rate = 10:

Crossover rate	Average Best Fitness found:
0.2	8219
0.25	8134
0.29	8039
0.33	8150
0.35	8070
0.4	8145
0.45	8014
0.5	8157
0.6	8322
0.7	8329

The best solutions appeared to lie in the region of 0.29 and 0.33. After trying different values in this area I found that the best solution was 0.3. I also began varying the probability of selection for the two crossover operators at this point. The table below shows the probability of ordered crossover occurring with the probability of partially mixed cross over being , 1-probability of ordered crossover.

Ordered crossover(Rate=0.3) probability	Average Best Fitness found
0.1	8313
0.2	8236
0.3	8183
0.4	8176
0.5	8210
0.6	8213
0.7	8195
0.75	8154
0.8	7973
0.9	8065

When the ordered crossover rate was 0.7, although my average fitness was 8195 the best individual fitness was found so far with a fitness of 7757. After exploring in region a little more I found that setting the probability of ordered crossover occurring to 0.75 to be the optimal value. Interestingly there were many individuals with a fitness in the region 7757 and 7900. Ordered crossover was clearly the better performer out of the two crossover operations.

I then performed the same experiments again this time with the elitism rate. The probability of ordered crossover was updated to 0.75 at this point

pop_size = 100

max_generations = 500

mutation_prob = 0.35

tournament_size = 3

crossover_prob = 0.3

Number of trials for each rate = 10:

Elitism rate	Average Best Fitness found:
0.01	8111
0.05	8104
0.15	8119
0.1	8044
0.2	8213
0.3	8302
0.35	8070
0.4	8145
0.45	8014
0.5	8157
0.6	8322
0.7	8329

I found that the optimal elitism rate at this point was around 0.05. This choice wasn't selected based on the average fitness found alone but rather on the fact that as I increased its value it was clear that convergence was happening at a faster rate and diversity was being reduced in the population.

Tournament size	Average Best Fitness found:	Computational Time-minutes
1	14381	1.5
2	8223	1.6
3	8072	1.11
4	8046	1.19

With a tournament size of 4 I coincidently achieved the optimal solution for the Berlin problem with an individual fitness of 7542 and lowest overall average fitness. Though this was optimal there was a significant increase in the computational time as I increased the tournament time. I will need to experiment with tournament sizes ranging from 2 – 4 for the larger problem sets as computational time will be a constraint as the search space increases with the population size increase.

I now began to experiment with the population parameter to observe its effect on the average fitness and computational time. I also reduced the number of trials as the computational time was growing significantly.

max_generations = 500

mutation_prob = 0.35

tournament_size = 4

crossover_prob = 0.3

Number of trials for each rate = 3

Population size	Average Best Fitness found:	Computational Time-Seconds
150	7935	91.80
200	8034	134.4
250	7912	154.8
300	7970	198.6

For a population size of 200 I again found the optimal solution of 7542 but the average best fitness found was 8034 and again there was a significant increase in the computational time. The optimal solution was also found for a population size 250 and an average best fitness of 7912. With population sizes from 200-300 the optimal solution was found every time but the computational time was increasing significantly.

Using the parameter values determined in the previous experiments for the 'Berlin.tsp' , I began to try and find good solutions on for next largest tsp problem, kroA100.tsp: I began first by varying the number of generations with the following parameters until I reached a convergence.

pop_size = 100

mutation_prob = 0.35

tournament_size = 4

crossover_prob = 0.3

Number of trials for each rate = 3

Number of generations	Average Best Fitness found:	Computational Time-Seconds
500	29281	41.54
600	28909	51.15
700	27048	59.87

800	24750	67.86
900	24704	76.60
1000	24254	86.79
1100	23824	93.23

As the number of generations was increasing the average best fitness was improving. I then began to vary the population size with the following parameters:

max_generations = 1100

mutation_prob = 0.35

tournament_size = 4

crossover_prob = 0.3

Number of trials for each rate = 3

Population size	Average Best Fitness found:	Computational Time-Seconds
150	23925	
200	23253	159.95
250	22521	201.00
300	23407	243.43
350	22927	276.88
400	22390	311.59
600	22157	462

There was no real improvement for population sizes over 250. I experimented with larger generation numbers but algorithm was converging on local optima. In attempt to over some this I reduced the population size to 250 and increased the rates of crossover and mutation and also varied the tournament size and elitism rate but I was still unable to find a better solution.

The best solution I found for this problem was **21970** with the following parameters. But the computational time was 4 minutes. I kept falling into local optima and the closer I got to the optimal solution

pop_size = 900

max_generations = 1000

mutation_prob = 0.35

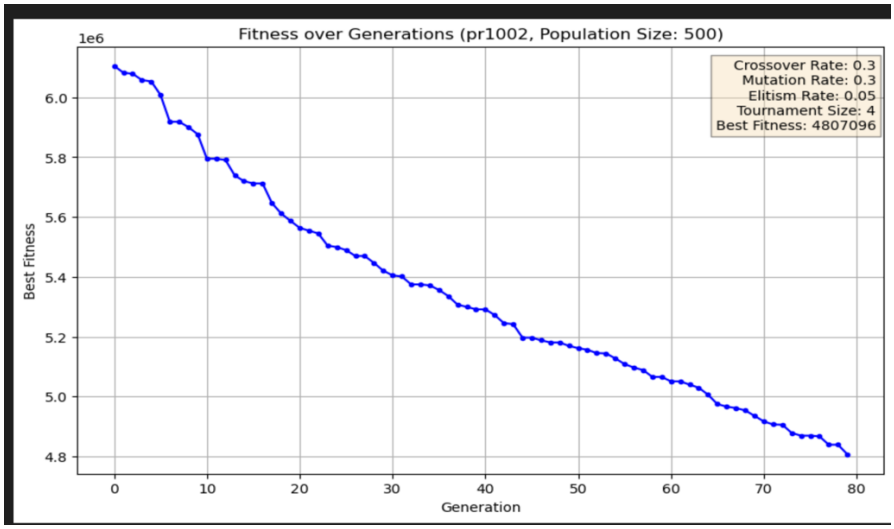
tournament_size = 4

crossover_prob = 0.3

Number of trials for each rate = 3

For the ‘prl002.tsp’ problem the search space is significantly larger. I experimented with small populations sizes in order to reduce the computational time and set generations = 1000. When the population size was 100 the algorithm was still converging when it reached the maximum number of generations. I then increased the number of generations to 1200 to see if it would improve further. It reached a best fitness of 2506976. Again it appeared to be still converging so I increased the number of generations again to 1200 and reached a fitness of 2340524. Again it was still converging but very far away from the optimal solution.

Here's a graph that shows the fitness improving for a population size of 500



Results:

	Berlin	**kroA100**	**Pr100**
Optimal fitness	**7542**	**21294**	**259045**
My best result	**7542**	**21970**	**2506976**

Overall the algorithm performed well in converging towards the optimal solutions although the draw back here was the computational time it took to run especially in more complex problems and it struggled with local optima. I believe increasing mutation or adjusting crossover probabilities might help avoid local optima but these adjustments can also raise computational time and the balance of these parameters proved sensitive . I think experimenting further with some the tournament size based on that of the population could also help improve the algorithms performance. Also implementing a hybrid method that integrates a local search step to refine offspring would likely improve its performance. Introducing early algorithm termination criteria would also be a good idea to reduce computational time when stuck in local optima.