AEA – Homework 3

We continued the work done in Homework 2. To optimize our solutions, we have used various configurations for the parameters of the algorithms. The values of these parameters are expected to drastically modify the results given, as they basically increase exploration and reduce exploitation, or vice versa.

Genetic Algorithm

A genetic algorithm is very sensitive to small changes used in its implementation. We have analyzed the following changes in our parameters:

- **Mutation – probability**: the mutation probability is directly proportional to the exploration and inversely proportional to the exploitation of the algorithm, as a high mutation rate makes the solutions more and more random. We executed the algorithm using the following mutation probabilities: 0 (which basically means no mutation), 0.05 and 0.2;
- **Crossover – probability**: it is not as easy to see which is increased, the exploration or exploitation, when observing changes in the crossover probability. The crossover creates new individuals based on old ones, so one could say it increases exploration. But on the other hand, these old individuals are constantly selected in the new generations based on fitness function, and doesn’t add new information to the old individuals (for example, sequential crossover on the same 2 bitwise individuals, with the same cutting point, will always and periodically yield the same individuals). This means that the crossover operation can increase both exploration and exploitation, but because it can still create never before seen individuals, it increases exploration more than exploitation. We experimented using the following mutation probabilities: 0 (which basically means no mutation), 0.2 and 0.5;
- **Selection – method**: it is clear that this method is only used in exploitation, as it does not exploration – it only uses old individuals to appear in the new generation, some of them being repeated. We have experimented using Tournament selection, Roulette Wheel selection and Random selection (this ignores the fitness, which makes the GA very similar to a random search).

As in the previous homework, we ran the algorithm 10 times, and used some statistic measurements to analyze our solution. We have run the algorithm using the following configurations (specified as (“mutation probability”, “crossover probability”, “selection method”), where selection is abbreviated as TUR for tournament selection, RWS for roulette wheel selection and RND for random selection):

1. (0.05, 0.2, TUR) – the configuration used on the previous homework
2. (0.2, 0, TUR)
3. (0, 0.5, TUR)
4. (0.2, 0.5, TUR)
5. (0.05, 0.2, RWS)
6. (0.05, 0.2, RND)

We used these configurations on the small (number of cities = 17) and large (number of cities = 60) datasets from the previous homework and analyzed the final results (the time taken is very similar from a run to another so no analysis was done on it). The most important details are marked in the 2 tables below, one for each dataset:

Data (N=17)	Crossover %	Mutation %	Selection Type	Average	Minimum	Maximum	Standard deviation	Time
Minimum 200.00	0.5	0.2	TUR	242.9	200	284	22.95	21.13
Maximum 507.00	0.2	0.05	RWS	451.2	403	507	28.04	56.63
Minimum average 242.90	0.5	0.2	TUR	242.9	200	284	22.95	21.13
Minimum stddev 17.27	0.2	0.05	TUR	264.3	228	291	17.27	19.71
Maximum stddev 44.38	0.5	0	TUR	335.7	241	398	44.38	22.25

Data (N=60)	Crossover %	Mutation %	Selection Type	Average	Minimum	Maximum	Standard deviation	Time
Minimum 819.00	0.5	0.2	TUR	951	819	1079	67.03	40.25
Maximum 2160.00	0.2	0.05	RWS	2087	2004	2160	45.95	56.46
Minimum average 951.00	0.5	0.2	TUR	951	819	1079	67.03	40.25
Minimum stddev 31.33	0.2	0.05	RND	2022.1	1954	2061	31.33	30.13
Maximum stddev 109.51	0.5	0	TUR	1577	1441	1748	109.51	35.6

As we said, for each configuration we ran the algorithm 10 times. For these 10 runs, we computed the minimum fitness, maximum fitness, average fitness and the standard deviation of the fitness. Analyzing all these statistics for each of the configurations, we looked at the **lowest minimum** (it tells us which configuration has given us the best possible solution), the **highest maximum** (it tells us which config is the worst), **minimum average** (it tells us which config will most probably always give us a very good solution), **minimum standard deviation** (it tells us which config is the most robust – repeated runs will yield very appropriate results) and the **maximum standard deviation** (tells us which config is closest to random search). Even though the lowest minimum seems to be the only measurement of interest, it is not true: if we want to choose a specific configuration and we choose the one which gave the lowest minimum, it might be because of pure luck of getting a very good random solution; on subsequent runs, such a configuration is expected to yield mediocre results. This is why having the same configuration for the minimum, minimum average and minimum stddev is ideal, and having the configuration for the minimum different to that of the maximum and maximum stddev is also ideal.

The first column specifies which measurement was considered and the value obtained for it. The next 3 columns specify the configuration of the parameters which gave us the value from the first column. The next 4 columns specify the statistics computed for the 10 runs of that configuration. Finally, the last column specifies the average running time of those 10 runs.

If we look closely, we can find the same configurations for both datasets, for the 5 statistic measurements analyzed, except for the minimum standard deviation, where the selection method used differs: on the small dataset we have the lowest stddev for Tournament selection, while for the large dataset the selection used is Random selection. Although it seems weird, the RND selection has this very low standard deviation because, in comparison with the TUR selection, RND does no exploitation which, for large datasets, means it has a very low chance of actually finding a good fitness; in this way, all candidate solutions can only have mediocre fitness values. The TUR selection has a very high exploitation, which means it can get blocked on local optimums, which explains why such a high standard deviation was obtained.

Further looking at the table, it can be seen that both tables show that configuration 4 was the best one (lowest fitness function) for both lowest minimum and minimum average. This is because it combined very well exploration (given by high rates for mutation and crossover) with exploitation (given by the tournament selection).

The worst solutions found are the ones were Roulette Wheel Selection was used. This selection, although fast, easy to implement and highly spread, does not behave well when the fitness function is very high because of the scaling it has to do. If all the fitness values are very high, the selection becomes a Random Selection, which at least is way faster (this can be seen in the Time column, in the rows corresponding to RWS and RND selections).

The very high standard deviation obtained for the TUR selection and no mutation is explained by having the TUR local optimum blockage explained above, combined with the fact that there is no mutation which could have gotten us out of the local optimum. This configuration converges very fast, although it can't be seen in the tables.

With configurations 1, 2, 3, 4 we tried to decide which values for mutation and crossover rates to use. The best (the one with the lowest minimum) one was configuration 4.

With configurations 1, 5, 6 we tried to decide which selection method is the best. The tournament selection has given the best results (configuration 1).

If we combine the mutation and crossover from 4 with the selection method from 1, we get exactly configuration 4, which means configuration 4 already uses the best mutation and crossover rates and the best selection method.

Ant Colony Optimization

Unlike the genetic algorithm, the ACO is not so sensitive to small changes to the parameters used. To actually see differences in our solutions, we have to greater increase or decrease the values of the parameters. We have analyzed the following changes in our parameters:

- **Number of ants –** the number of ants used increases the exploitation more than exploration, as all the ants try to find the shortest route. We have experimented using 2, 10 and 100 ants
- **Decay rate –** changes on this parameter have roughly the same impact on the algorithm as the changes on the mutation rate from the genetic algorithm. The decay is used for updating the pheromone matrix, for the arcs of the path taken by each ant. The pheromone matrix is updated 2 times: first it updates after each solution found, and it updates only the arcs of the path taken by that ant based on its fitness (local update), and at the end of the iteration it updates again all the arcs of every ant’s path, based on the best ant’s fitness (global update). If the decay is high, it means that the pheromone that already exists on an arc will be highly replaced by the current fitness, which furthermore means that past solutions are forgotten, which is similar to how high mutation rate operates. We used the following decay rates in our experiments: 0.01, 0.1, 05
- **Greedy rate –** this is operator is not specific to ant colony; we used this one to replace the roulette wheel like selection of the next arc for an ant with a greedy approach: normally, in finding the route an ant will take during the ACO algorithm, the ant has to decide at each step which will be the next node to visit; this next node is selected from all candidate nodes, in the same manner an individual is selected in the Roulette Wheel Selection in GAs, except it doesn’t compute its fitness, and uses the pheromone matrix instead. We used the greedy rate to replace this selection for, let’s say, 70% of the times, with a greedy approach, meaning that the next node to visit will be the one with the best pheromone from all candidate nodes. We experimented with greedy rates of 0.1, 0.7 and 0.99.

We ran the algorithm 10 times, and used the same statistic measurements to analyze our solution as before. We have run the algorithm using the following configurations (specified as (“number of ants”, “decay rate”, “greedy rate”) :

1. (10, 0.1, 0.7)
2. (100, 0.1, 0.7)
3. (2, 0.1, 0.7)
4. (10, 0.5, 0.7)
5. (10, 0.01, 0.7)
6. (10, 0.1, 0.99)
7. (10, 0.1, 0.1)

As before, we can find in the following 2 tables the measurements of our runs, each table corresponding to one of the datasets previously used:

Data (N=17)	No. Ants	Decay	Greedy Rate	Average	Minimum	Maximum	Standard deviation	Time
Minimum 168.00	100	0.1	0.7	186.3	168	192	7.27	30.13
Maximum 223.00	2	0.1	0.7	205	195	223	9.84	0.59
Minimum average 186.30	100	0.1	0.7	186.3	168	192	7.27	30.13
Minimum stddev 5.31	10	0.1	0.7	197.6	189	205	5.31	3.44
Maximum stddev 9.84	2	0.1	0.7	205	195	223	9.84	0.59

Data (N=60)	No. Ants	Decay	Greedy Rate	Average	Minimum	Maximum	Standard deviation	Time
Minimum 305.00	10	0.1	0.99	330.6	305	352	14.85	28.14
Maximum 588.00	10	0.1	0.1	549.7	502	588	25.16	30.66
Minimum average 330.60	10	0.1	0.99	330.6	305	352	14.85	28.14
Minimum stddev 14.11	100	0.1	0.7	371.7	350	398	14.11	280.76
Maximum stddev 43.60	2	0.1	0.7	415.9	351	490	43.6	5.76

The tables’ columns and rows have the same meaning as the previous ones.

This time, the statistic measurements differ greatly from one dataset to the other, by the configuration which gave the results. The only configuration which gives us on both datasets the same measurement is (2, 0.1, 0.7), on the maximum stddev obtained. This one was obtained because of the very low number of ants used, which made the algorithm block very soon in a local optimum.

An important thing to note is the 0.1 Decay rate present in all runs. This can be explained because 0.01 is a way too low decay rate, which means the first solutions found will be the only ones found, because of the very slow replacement of pheromone. A decay rate of 0.5 is too high, and makes the algorithm act like a GA with high mutation => random search with many possible solutions given. The middle decay rate of 0.1 serves us the most on all measurements (given the fact that there are other parameters used too - the same decay rate would have been highly unlikely to create both the lowest and the highest standard deviations).

On the small dataset, the greedy rate is the same for all measurements: 0.7. This is because of the same reason why a decay rate of 0.1 is present everywhere. The only difference is, then, on the number of ants used, where all results are as expected: the highest number of ants gives us the best results (minimum and minimum average); the lowest number gives the worst results (maximum) and the highest standard deviation. The interesting thing here is the 10 ants which gave the lowest standard deviation. Considering the 2 ants which gave the highest stddev, we would expect that the 100 ants would give the lowest standard deviation. But because high number of ants means high exploitation, this creates a high opportunity for the algorithm to get stuck in a local optimum. So the medium number of ants is expected to be the one to give the lowest stddev.

On the large dataset, the search space is a very big one. Heuristic approaches on very big search spaces are expected to give poor results. This makes room for a high greedy rate to give us very good results, which is what we can see in the minimum and minimum average rows, and also in the maximum row, where a low greedy rate was used. The highest standard deviation, as we already have seen, is given by the lowest number of ants. Then again, the lowest standard deviation surprises. Considering what we said for the small dataset, we would expect the lowest standard deviation to be obtained for the 10 ants. But because we have large dataset – very big search space, there are a lot of local optimums were the algorithm can get stuck, and because we have a big search space with uniformly generated distances between the cities, a big number of ants will all converge to a mediocre fitness.

Analyzing the parameters one by one (which value gave the best results considering the other ones fixed), we would expect the best value to be found for the configuration (100, 0.5, 0.99) for the small dataset and (100, 0.01, 0.99) for the large dataset. The different value for the decay rate is to be expected because, in both cases, we have high greedy rate, which solves the exploitation problem, while for the large dataset we have a big search space that makes the high exploration, related to the high decay rate, mostly useless. These 2 runs' results are noted below:

Data	Minimum	Maximum	Average	Stddev	Time
N=17	177.00	196.00	187.50	6.70	37.25
N=60	266.00	330.00	301.20	15.49	328.96

On the small dataset, individually selecting the best parameters has not given us an even better configuration. Moderate values need to be chosen for at least some of the parameters, if not for all of them, especially when you get a very small exploitation given by the high decay.

On the large dataset, as the decay rate was low, greedy rate high and number of ants high, we did indeed obtain better results. As we said earlier, having a high exploitation yields the best results, even if the result corresponds to a local optimum and not the global one.

Conclusion

The two algorithms used and optimized both give viable solutions, and changing the values of the parameters used can drastically change the outcome of the executions.

The best configuration found so far is using ACO with 100 ants, 0.1 decay rate and 0.7 greedy rate for the small dataset (gives us the solution with value 168, compared to 178 obtained with no optimizations), and ACO with 100 ants, 0.1 decay rate and 0.99 greedy rate for the large dataset (gives us the solution with value 266, compared to 376 obtained with no optimizations).