Home problems, set 2

Mats Högberg, 930615-1979, matsho@student.chalmers.se

October 17, 2018

Problem 2.1

a) We can think of the problem of generating a TSP path as first selecting a starting city, then selecting the next city, then the next, and so on until we have visited all cities. The algorithm for this can be written as:

```
1. Choose starting city (N choices)
2. Choose next city (N - 1 choices)
3. Choose next city (N - 2 choices)
...
N. Choose last city (1 choice)
```

Let P_{all} be the total number of paths (including reverse paths and paths that start in different cities but run through the cities in the same order). We calculate P_{all} by multiplying the number of choices we have in each step. We get that

$$P_{all} = N * (N-1) * (N-2) * ... * 1 = N!$$

Now, since paths that start in different cities but run through the cities in the same order are equivalent, it doesn't matter which city we start in. Thus, we can choose a fixed starting city and always start and end the path in that city. We can thus remove step 1 in the algorithm above, and the result of multiplying the number of choices in each step would then instead be

$$P'_{all} = (N-1) * (N-2) * ...1 = (N-1)!$$

For each path $p \in P'_{all}$ there will also be a path $p_{reverse} \in P'_{all}$ that goes through the cities in the reverse order. To exclude all reverse paths, we simply divide the number of paths by 2

$$P_{distinct} = \frac{P'_{all}}{2} = \frac{(N-1)!}{2}$$

Thus the total number of distinct TSP paths is $\frac{(N-1)!}{2}$, where N is the number of cities.

b) See `./Problem 2.1/GA21b.m`

c) See `./Problem 2.1/AntSystem.m`

d) See code in `./Problem 2.1/NNPathLengthCalculator.m`.

Starting in city 50, we get a nearest neighbour path of length ≈ 135 (with other starting cities it ranges from ≈ 130 to ≈ 160). This is actually shorter than the best path found by the GA (as shown in figure 1), which had a length of ≈ 139.1.

I can think of two reasons why the GA performs so badly.

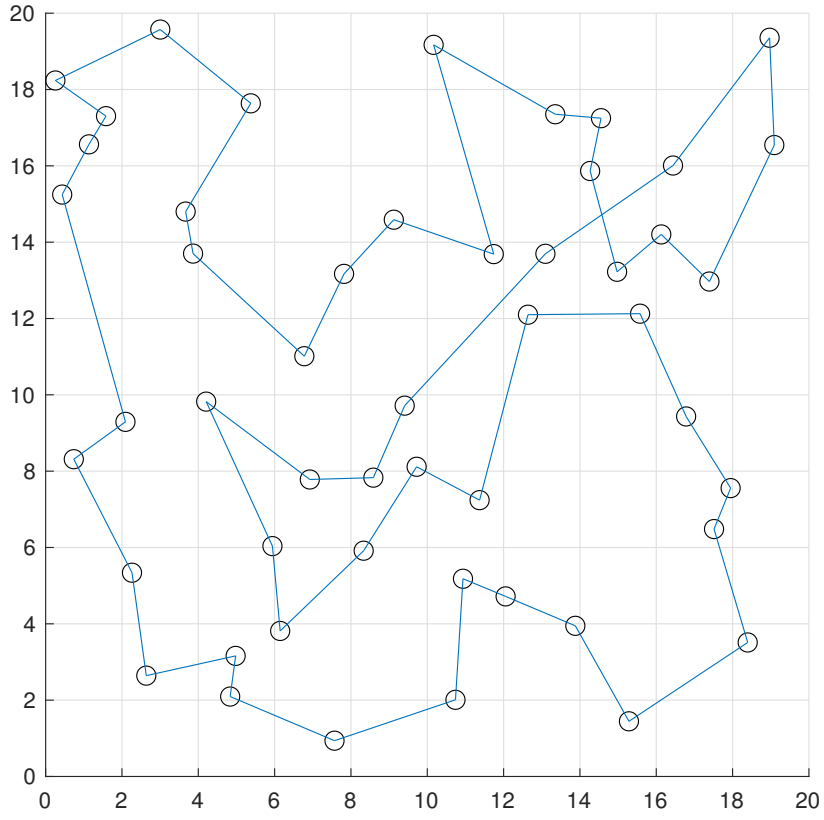

Figure 1: The best path found by the GA for the TSP. The length of this path is ≈ 139.1.

1. The GA starts with a random path. The length of this path will, in almost all cases, be very long, and a lot of swaps will be required to make it into a good path.
2. It is easy for the GA to get stuck in a local minimum. If the GA finds a relatively good path (such as the one shown in figure 1), just swapping two cities in the path will most likely result in a longer path. Therefore we would need to hit a lucky streak of selections and mutations in order to find a shorter path and get out of the local minimum.

e) The best paths found after running both algorithms for a long time are plotted in figures 1 and 2. The length of the best path found by the GA is ≈ 139.1 and the length of the best path found by the ACO is ≈ 121.1. The length of the nearest-neighbour path (as found in d)), was ≈ 135 when starting in city 50.

Problem 2.2

The contour plot of $f(x, y)$, with the local minima marked with black dots, can be seen in figure 3. From looking at the plot, we can see that there are a total of four local minima over the range $x, y \in [-5, 5]$.

The local minima found by the PSO can be seen in table 1. The PSO finds all four local minima that were

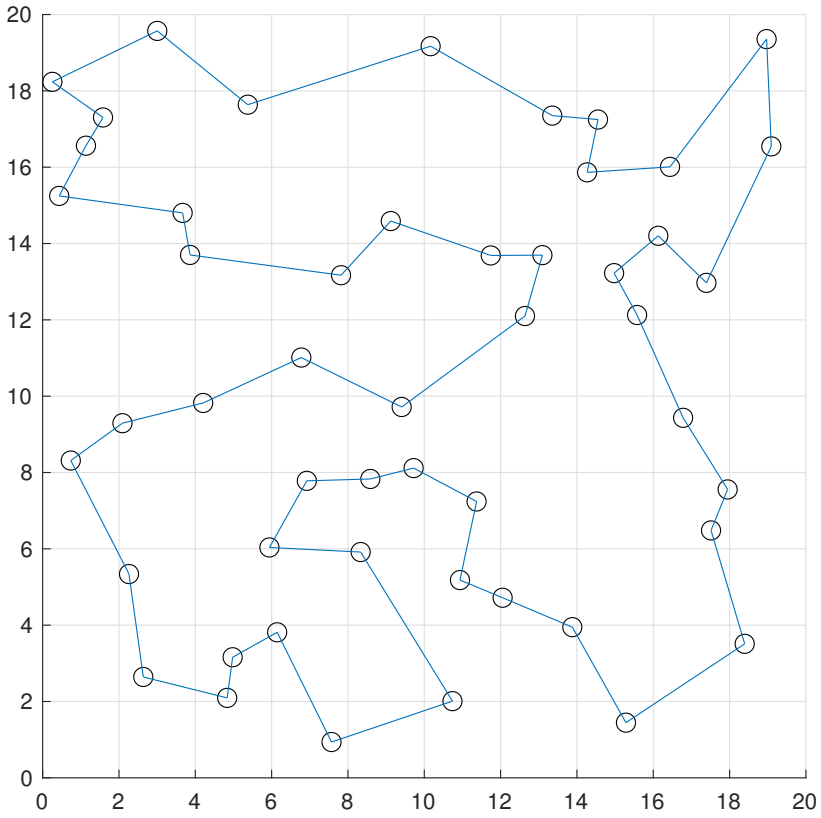

Figure 2: The best path found by the ACO for the TSP. The length of this path is ≈ 121.1.

x	y	$f(x, y)$
-2.805118	3.131313	0.000000
3.000000	2.000000	0.000000
3.584428	-1.848127	0.000000
-3.779310	-3.283186	0.000000

Table 1: The local minima found by the PSO in problem 2.2.

identified from the contour plot.

Problem 2.3

See code in `./Problem 2.3/NN23.m`.

The GA uses real number encoding with tournament selection, single-point crossover, elitism and both creep and regular mutations (see code for the values used for the different parameters). The fitness measure for slope i was defined as:

$$F_i = \begin{cases} \frac{d_i}{L_i} * 0.9 & , \ d_i < L_i \\ 0.9 + \frac{\overline{v_i}}{v_{max}} * 0.1 & , \text{ otherwise} \end{cases} \tag{1}$$

This function tries to encode the fact that making it down the slope is what's most important in two different ways. First, 90 % of the fitness is rewarded depending on the distance covered, and only 10 % depending on the average speed. Second, it doesn't give any reward to high speed until the entire slope has been cleared. It is therefore always better to make it down the slope, no matter how slow you go, than having a high average speed but crashing. The split of 90 %/10 % between the distance and the speed factors was chosen rather arbitrarily based on performance. Some other splits were tested, such as 50/50, 75/25 and 80/20, but the best results were had with 90/10.

The fitness measure over a set of slopes was defined as the average $\overline{F}$ over the slopes in the set. Using $F = \min F_i$ was also tried, but it generated really bad results. This was probably caused by having one training slope that was (seemingly) impossible to finish without crashing, which made the algorithm optimize only the distance travelled on that particular slope. Had the GA instead been trained only on slopes that could actually be descended without crashing this fitness measure might have performed better.

Other fitness measures for single slopes were also tried, for example using only distance covered, multiplying distance and average speed, taking the average of distance and speed after normalizing with slope length and max speed, but the fitness function described in equation 1 gave the best results.

When running the GA, every individual that was better than the previous best seen individual on the training data was also tested on the validation data. If this individual wasn't also better on the validation data, a counter was incremented. The algorithm was terminated when this count reached 25, i.e. if the GA had found 25 consecutive new best individuals (measured on the training data), that weren't also better on the validation data.

The fitnesses of the best individual on the training data and the validation data during a run of the GA can be seen in figure 4. One thing to note in this plot is that the fitnesses on the training data and the validation data stay fairly constant as the number of generations increases. One could expect that as the GA progresses, the fitness on the training data would increase and the fitness on the validation data would decrease due to overfitting on the training data, but as we can see this doesn't happen in this case. This might be due to having training and validation data that are too similar, or maybe it's a sign that the training data is very diverse and that it's hard to overfit to it.

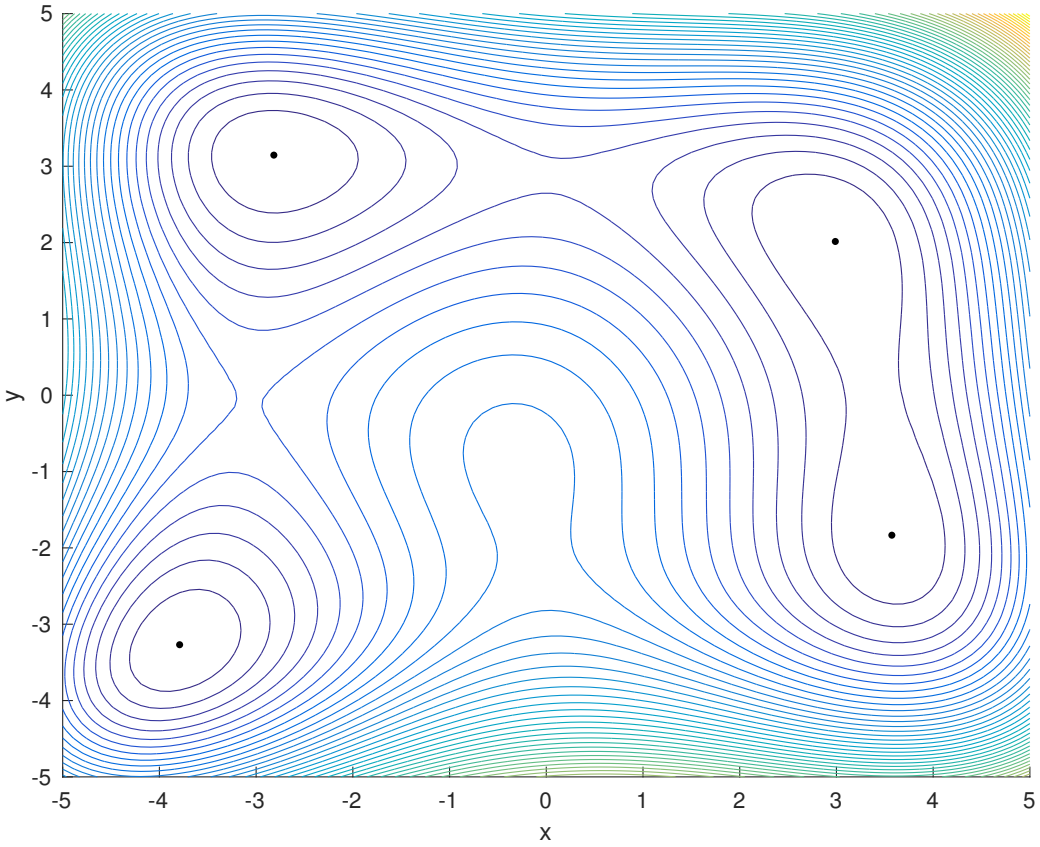

Figure 3: The contour plot of $f(x, y)$ for $x, y \in [-5, 5]$. The four local minima of the function in the given range are marked with black dots.

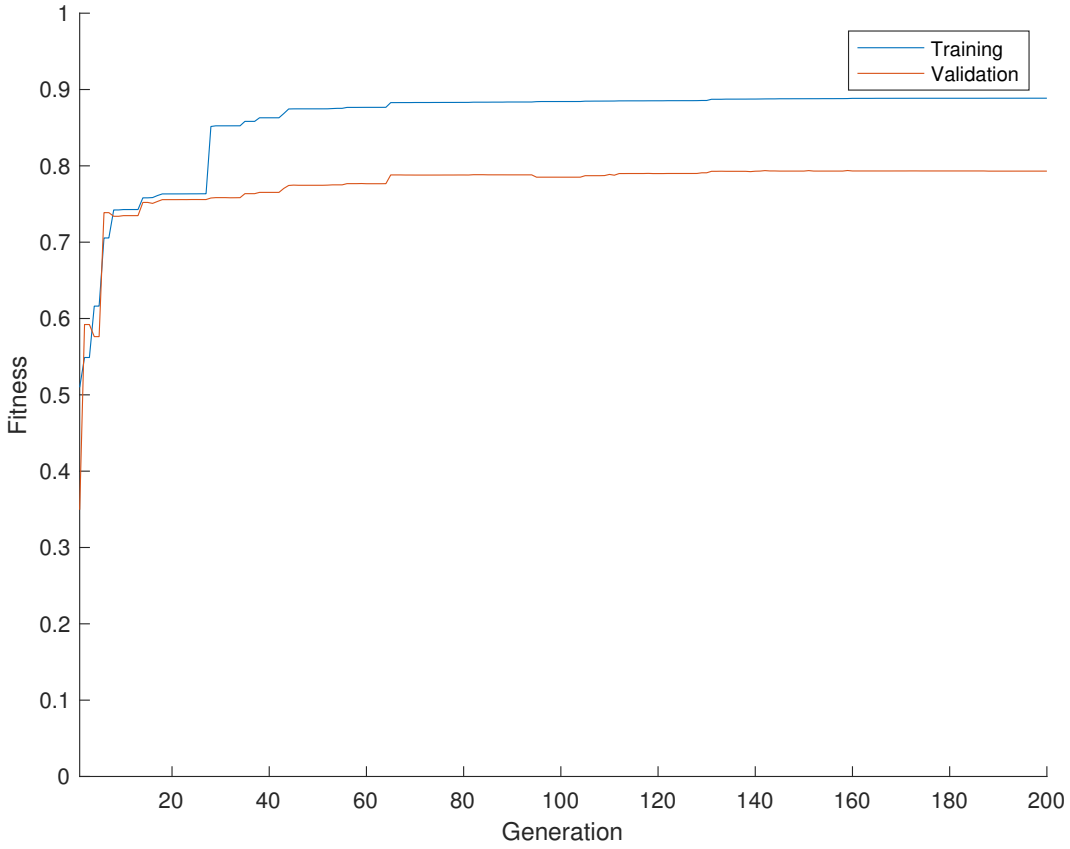

Figure 4: The fitnesses of the best individual on the training data and the validation data during a run of the GA.

To run the best network found on a specific slope, edit the `iSlope` and `iDataSet` variables in `TestProgram.m` and run the file. This file contains two chromosomes, one that works really well when `GEAR_CHANGE_COOLDOWN = 1` and another one that works pretty well when `GEAR_CHANGE_COOLDOWN = 2`. The reason for supplying two different chromosomes, for two different values of gear change cooldown, is that I made the mistake of setting this constant to 1 instead of 2 when I implemented the truck simulation, therefore allowing the gear to be changed every second instead of every two seconds. I didn't discover this error until shortly before the submission deadline, so I didn't have that much time to find an optimal network for the correct simulation. Therefore I'm including the best chromosome for the incorrect simulation as well, since I spent a lot more time on finding it. If you want to test that chromosome in addition to the one for the correct simulation, you would have to manually set the `GEAR_CHANGE_COOLDOWN` constant to 1 in the `RunSimulation.m` file.

The hardest thing about this problem was not to find optimal parameters for the GA (even though it took a while), but rather to define a suitable fitness measure and to generate good training and validation data. The ability for the neural network to generalize to previously unseen data depended a lot on which data it had been trained on. For example, if none of the slopes in the training data required the truck to make heavy use of the foundation breaks, the neural network would fail to handle steeper test slopes where just downshifting wasn't enough to keep the speed of the truck down.

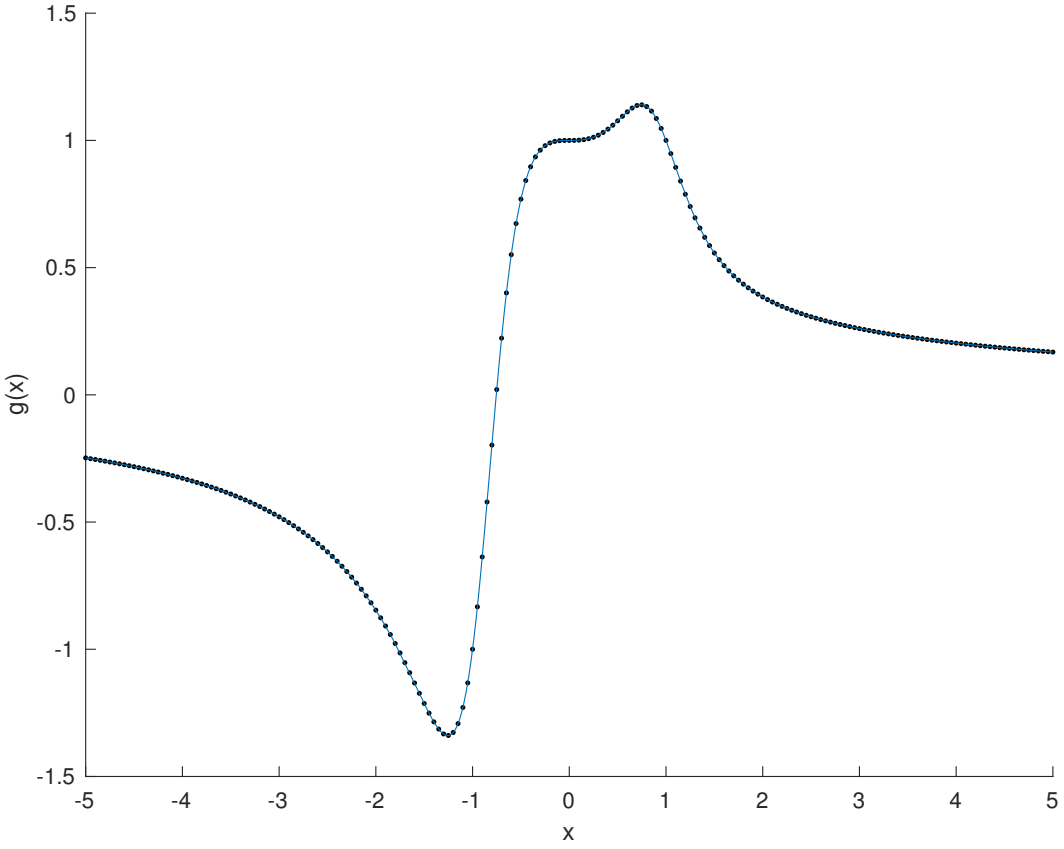

Figure 5: A plot of the given function data and the best estimation of $g(x)$ found by the LGP program. Black dots = data, blue line = $\hat{g}(x)$.

Problem 2.4

See code in `./Problem 2.4/LGP24.m`.

The best estimation of $g(x)$ found by the LGP program was

$$\hat{g}(x) = \frac{x^3 - x^2 + 1}{x^4 - x^2 + 1}$$

This estimation has the error $e = 0.00000$ for the given function data. A plot of $\hat{g}(x)$, alongside the given function data, can be seen in figure 5. We can see that the LGP program found an approximation of $g(x)$ that is very close, if not the exactly equal, to the sampled function, at least in the given range $x \in [-5, 5]$.

As for the LPG, the number of variable registers was set to 7, the number of constant registers was set to 3, and the values for the constant registers were 1, 3 and -1. Chromosomes with more than 100 instructions (i.e. 400 genes) were penalized by a factor of 10. A variable mutation rate depending on the diversity of the population was used, which seemed to help in preventing premature convergence. Plots for how the mutation rate and diversity of the population varied during a run of the GA can be seen in figure 6.

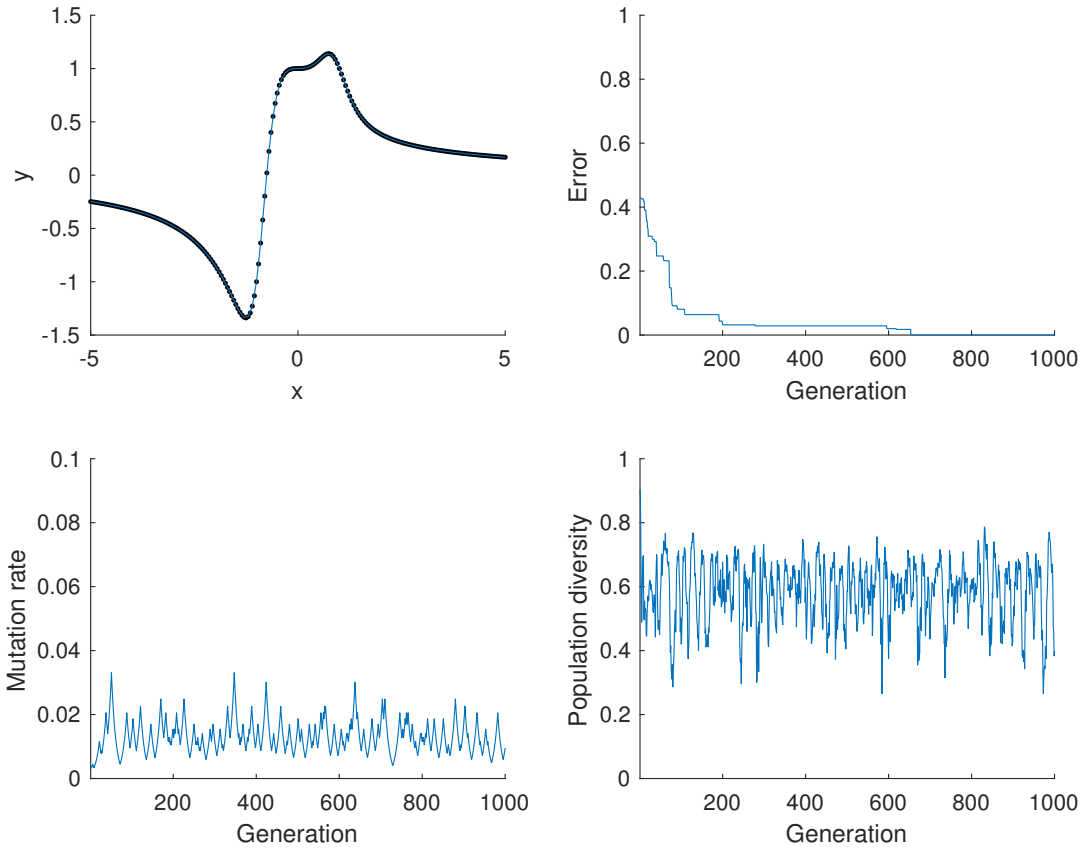

Figure 6: Plots showing the function data given in problem 2.4 and the estimated function found by the LGP program, as well as how the error, mutation rate and diversity of the population varied during a run of the GA.