Stochastic optimization algorithms, problem 2

Adam Tonderski
tadam@student.chalmers.se
930524-1037

Problem 2.1, Traveling Salesman Problem

TSP, the traveling salesman problem, concerns finding the shortest path through N cities, including the return back to the first city. It's a difficult problem to solve with brute force, since the number of possible solutions is very large, $\frac{(N-1)!}{2}$ to be exact.

Arriving at this number is straightforward. First of all there are $N!$ paths in total. However, because of rotational symmetry, each solution corresponds to N paths. For example, $(1, 2, 3)$, $(2, 3, 1)$ and $(3, 1, 2)$ represent the same solution. So we need to divide the total number of paths by N. Finally, for each of the remaining paths, there is a mirrored, or backwards, version that also needs to be removed. For example, $(1, 2, 3)$ and $(3, 2, 1)$ represent the same solution. So putting it all together we have:

$$\frac{N!}{2N} = \frac{(N-1)!}{2} \tag{1}$$

Note that in the case of $N = 2$, this formula doesn't hold since the number of paths is 1, not 0.5. This is because the argument about rotational symmetry and the argument about backwards/forwards (or clockwise/counterclockwise) become the same thing, and is therefore compensated for twice. Another way to view it is that there is that the concept of clockwise vs counterclockwise doesn't exist for a straight line.

Nearest Neighbour approach

A naive and very fast solution to the problem is the Nearest Neighbour path. It is created by picking a random starting city and then keep picking the nearest city from the remaining cities. A path created this way will almost never be the

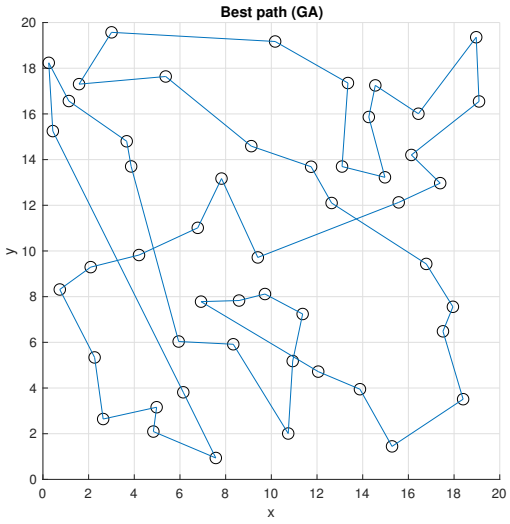

Figure 1: Best path that was found with the GA algorithm. Reaching this solution took 50 000 generations with 100 individuals in the population. It's still pretty bad though, with a total length of 161.2678 and a lot of crisscrossing.

Table 1: Parameters for the GA that was used to solve the TSP.

population size	100
mutation probability	0.04
tournament selection parameter	0.7
tournament size	2
elitism	1

best solution, but it turns out it's not a horrible solution either. A series of runs resulted in path lengths from 138.3583 to 158.9935. You can reproduce this result by running the 'NNPathLengthCalculator.m' script in '2.1d/'.

GA approach

The first approach to solving this problem was to use a GA, Genetic Algorithm. The only major change from previous GA implementations were swap mutations and no crossover. This was in order to prevent invalid paths. The first generation was initialized to completely random paths.

The best path after 50 000 generation was of length 161.2678 and can be seen in Figure 1. The chosen parameters for the GA are presented in Table 1. The algorithm can be tested by running 'GA21b.m' in '2.1b/'.

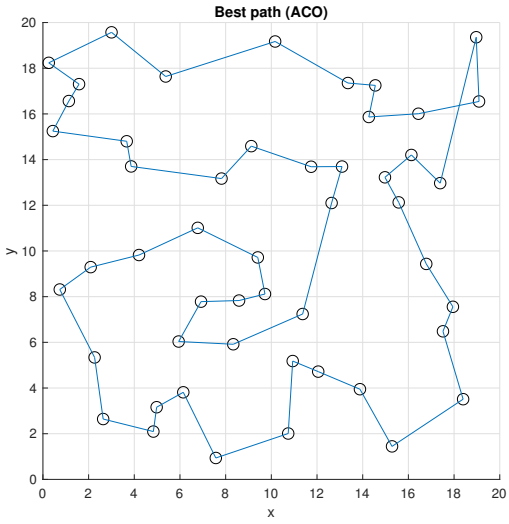

Figure 2: Best path that was found with the GA algorithm. Reaching this solution took 50 000 generations with 100 individuals in the population.

Ant Colony Optimization approach

The best approach that was tested in this problem was ACO, Ant Colony Optimization. This is not a surprise since graph searches is where this algorithm really shines. Following the recommendation in the assignment, the pheromone weights were initially set to the inverse of the nearest neighbour path length, see Section .

With 50 ants (to match the number of nodes), $\alpha = 1$, $\beta = 3$ and $\rho = 0.3$, a path of length 122.8058 was found. It took 46454 iterations and the path can be seen in Figure 2. This best path was stored in as 'bestPath' in the 'BestResultFound.m' script that can be found in '2.1e/'. The algorithm itself can be tested by running 'AntSystem.m' in '2.1c/'.

An interesting find was that the algorithm was a lot faster if the pheromone weights were updated 'backwards', from the destination node to the source node instead of the other way around. This was discovered by accident and points towards the possibility that the 'correct' approach gets stuck in a local optimum. While updating the weights backwards is hard to justify, and probably works this well by accident, it could be a good idea to update both directions, making the pheromone matrix symmetric. Since the problem graph isn't directed there isn't really any point in making the pheromone graph be directed. This is of course not applicable to any problem!

Table 2: Minima of $f(x, y)$ that were discovered using PSO

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

Conclusion

First of all, ACO was clearly the best algorithm for finding a really good solution. It found a path shorter than 130 after only a couple iterations. Getting that down to 123 was a lot harder and took thousands of iterations. Still, the other algorithms were unable to get there at all.

A more surprising find was that Nearest Neighbour outperformed (or performed similarly to) the GA even after almost 1 million total individuals. A big reason why ACO performed so much better is that it basically starts off with the Nearest Neighbour solution immediately and then improves on it, whereas the GA searches through the entire solution space. This shows that while GAs are a very general and powerful algorithm, they are not the best choice for all problems. While the lack of crossover doesn't make it a completely fair comparison, the basic conclusion about search spaces should still apply to a more refined GA implementation.

Problem 2.2, Particle Swarm Optimization

This problem is about minimizing the function

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2 \tag{2}$$

over the range $x, y \in [-5, 5]$ using PSO, Particle Swarm Optimization. The algorithm was implemented according to the instructions in the course book. The addition of a varying inertia weight proved especially useful since it improved the accuracy of the algorithm immensely. The program can be tested by running 'PSO22.m' in '2.2/'.

Figure 3 shows a contour plot of $f(x, y)$ (or a slightly modified version of it). The red dots correspond to the minima found by the PSO algorithm. Since each run only provides a single solution, plotting them on top of the contour was a useful way of making sure that all the solutions were found. Of course, it's also possible to analytically prove the number of solutions.

The exact minima locations are presented in Table 2. We can see that all the minima values are 0, which makes sense since the function is the sum of two squared terms and therefore cannot be negative.

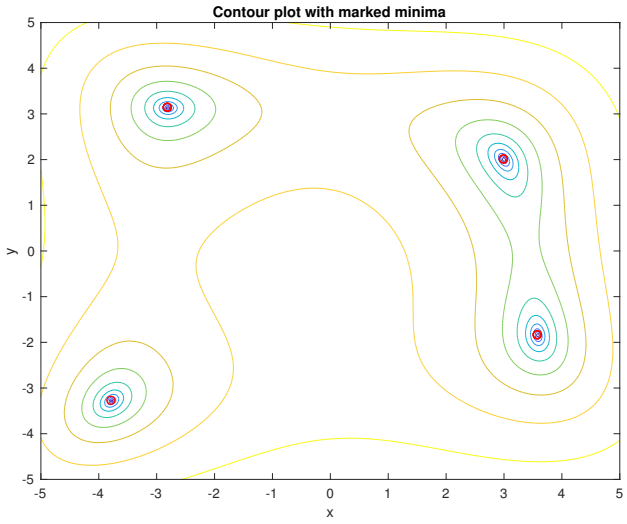

Figure 3: Contour plot of the function that has to be minimized. In order to make the visualization clearer, the function was actually modified to be $\log(0.01 + f(x, y))$. The red rings correspond to the solutions found by the PSO algorithm.

The swarm size was chosen to be 100, number of iterations to 2000, cognitive component to 2, social component to 2 and α to 1. The time step was set to 0.1, but in hindsight it would have been easier to hard code it to 1 and make the computations a little bit simpler. The precision didn't get affected by setting it to 1.

One of the most noticeable things about PSO was the speed. Reaching a solution took less than a second (with a couple of thousand iterations). Comparing this to a GA is not really apples to apples, but it still shows the potential for PSO to perform really well.

Problem 2.3, Truck braking optimization

The truck braking optimization problem is a simplified example of industry applications of neural networks together with GAs. The task was to create a simulation of a truck driving down a slope under the control of a neural network. Then the weights of this network were optimized by the GA.

An individual was evaluated by running simulations on multiple slopes and then averaging the fitnesses. The fitness, f, of a single slope was calculated according to:

$$f = L + \bar{v} \tag{3}$$

where L is the traveled length and $\bar{v}$ is the average velocity over the slope.

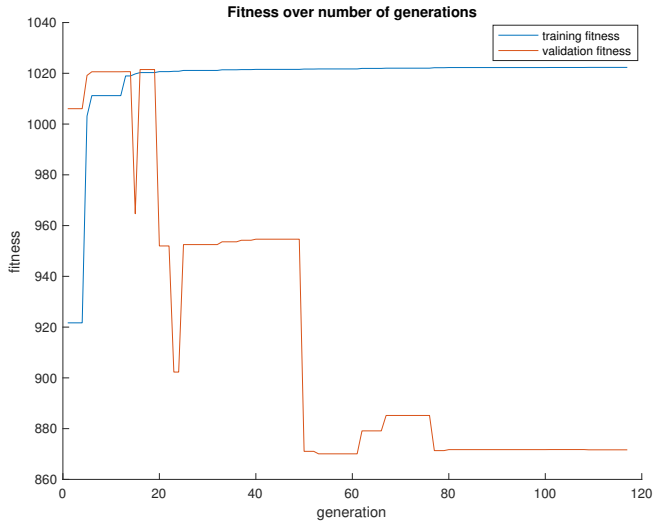

Figure 4: Fitness over time for the run that produced the best network. Overfitting starts to happen after around 20 generations, when the validation fitness (blue) starts to decrease even though the training fitness (red) increases.

Table 3: Fitnesses of the best network on the different slope sets

slope set	average	min	max
training	1020.28	1018.69	1022.47
validation	1021.48	1018.60	1022.99
test	1019.86	1017.54	1022.99

Figure 4 shows how the best training and validation fitnesses changed after each generation. It clearly shows how the network initially improves for both data sets but quickly start overfitting (after only 20 generations). Without the separation of data into multiple independent sets, it would be very hard to find this threshold and terminate the optimization in time. The fitness results of best network are presented in Table 3.

Some implementation details that should be noted are that the number of hidden neurons was chosen to 7 (after some experimentation). Weights were initialized between -7 and 7 so that the activation function, tanh, could take practically all values between -1 and 1. The brake pressure was decoded by uniformly projecting the output on the range [0,1] (by adding one and then dividing by 2). The gear shift was decoded using the following equation:

$$\Delta G = \begin{cases} 0 & \text{if } |o| < \frac{1}{3} \\ \text{sgn}(o) & \text{otherwise} \end{cases} \tag{4}$$

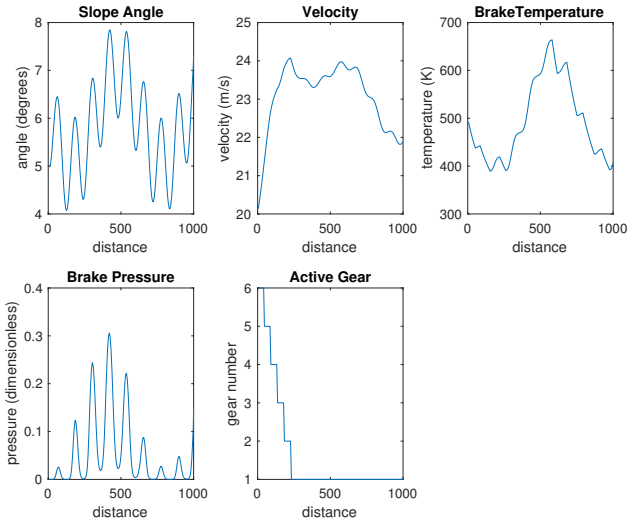

Figure 5: Performance of the best network on one of the slopes with a high angle variation. Notice how the brake pressure has a similar periodicity to the slope angle. Another interesting observation is that the brake temperature has a maximum around 650, which is exactly where the brake starts to lose efficiency.

where ΔG is the gear shift, and o is the network output.

The mutation probability was set to $\frac{2}{m}$ (where m is the chromosome length) and the creep rate was set to 1. The value of the other parameters and other implementation details can be found in the attached code (folder '2.3/'). The entire algorithm can be tested by running 'GA23.m' and the best network can be tested by running 'TestProgram.m'. It currently runs the second validation slope, but it can be changed by modifying the 'iSlope' and 'iDataSet' variables.

One of the main challenges in this assignment was constructing a good set of slopes. Some slopes where the angle was too high turned out to be impossible to solve under the constraints presented in the problem. It was also important to have a high diversity between the slopes, so that the network learned to handle a wide range of scenarios. Figure 5 and Figure 6 show how the best network manages to respond decently well to 2 completely different shapes.

Equation 3 was chosen instead of the suggested fitness of $f = L\bar{v}$, because it put a higher priority on actually finishing the entire slopes. The original formula promoted more aggressive driving, which usually lead to some slopes failing. However, with the modified formula the network managed to complete all the training slopes, all the validations slopes and all the test slopes. As seen in Figure 5, the velocity was not always hovering around maximum for all of the slopes. But, as every truck driver knows, safety first!

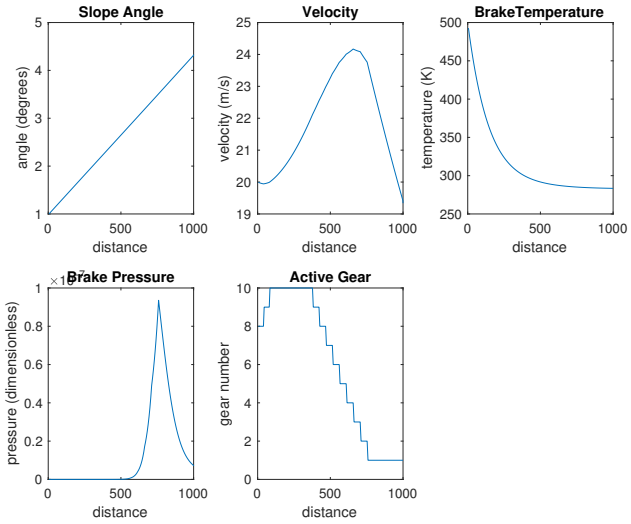

Figure 6: Performance of the best network a slope with constant angle increase. It's quite interesting to see how the gear is initially increased, in order to maximize the velocity, and then rapidly decreased when the velocity starts to come closer to the allowed maximum. Another noteworthy observation is that the brake pressure is very close to zero the entire time.

Problem 2.4, LGP programming

The task here was to use LGP, Linear Genetic Programming, to construct a function that matches a given set of data as closely as possible.

During initial testing it proved to be very important to include a penalty for chromosomes that were deemed too long. Otherwise the chromosomes kept growing indefinitely. The chosen fitness equation was:

$$f = \begin{cases} 1/e & \text{if } L < L_{\max} \\ 1/e * k^{L-L_{\max}} & \text{if } L > L_{\max} \end{cases}$$

where e is the total error (calculated according to the instructions in the assignment), L is the chromosome length, $L_{\max}$ is the maximum chromosome length and k is the penalty factor.

After a lot of parameter tweaking a good value for the constant registers was found to be $[1, 2, 3]$. The number of variable registers was set to 6, $L_{\max}$ was set to 200 and k to 0.995. After over 30 000 generations, with 100 individuals per generation, this configuration discovered a function with an error of 2.8×10^{-9}. This error is small enough that this is very possibly the original function that was

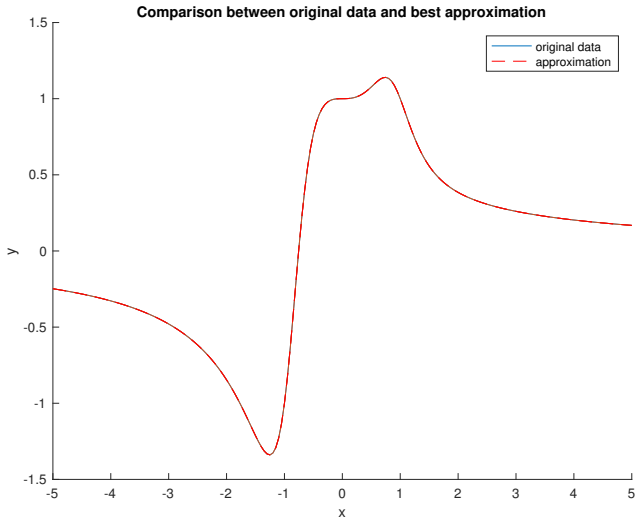

Figure 7: Comparison of original function data with the best LGP approximation. The error is 2.8×10^{-9}.

used to generate the data and any error is due to rounding. A comparison between the approximation, see $g(x)$ below, and the original data is shown in Figure 7.

$$g(x) = \frac{\frac{3}{x^2*(x-1)} + 3}{3 * x + \frac{3}{x^2*(x-1)} + 3} = \frac{x^3 - x^2 + 1}{x^4 - x^2 + 1} \tag{5}$$

Since this was a more difficult problem to solve, some experimentation was done with variable mutation rates. However, two problems were found that in the end resulted in skipping this altogether. First of all, computing the population diversity slowed down the algorithm to the point where even if the number of required generations was reduced, the total run time for a good solution was longer. Secondly, the population diversity didn't respond very well to the mutation rate, probably in big part due to the variable chromosome lengths.

In the end a much simpler approach was adopted when it came to variable mutation rates. The mutation rate was made to be inversely proportional to the chromosome length, not the population diversity. This was of course inspired by the rule of thumb that says that a good mutation rate is somewhere around $1/m$, where m is the chromosome length.

Another interesting tweak that was very successful (around 10x reduction in the number of iteration to find a good solution) was to set every variable register to the input value, instead of just the first one. This solved the problem of frequent divisions by 0, which caused the constants to explode in value. By forcing the

algorithm to explicitly create a 0 if it was needed, the frequency of these large numbers diminished significantly.

However, since instructions in the assignment clearly specify that all variable registers but the first one should be set to 0, the final implementation didn't include this change. The code can be tested by running 'LGP24.m' in '2.4/'. There is also a 'TestFit.m' in the same directory that compares the best found approximation (hard coded) with the original function data (like in Figure 7).