# Genetic Algorithm Report

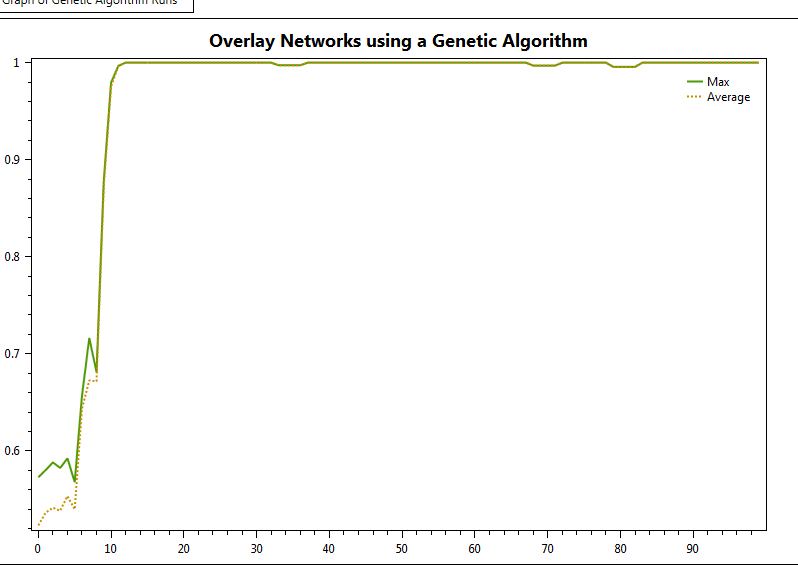
Santipab Tipparch

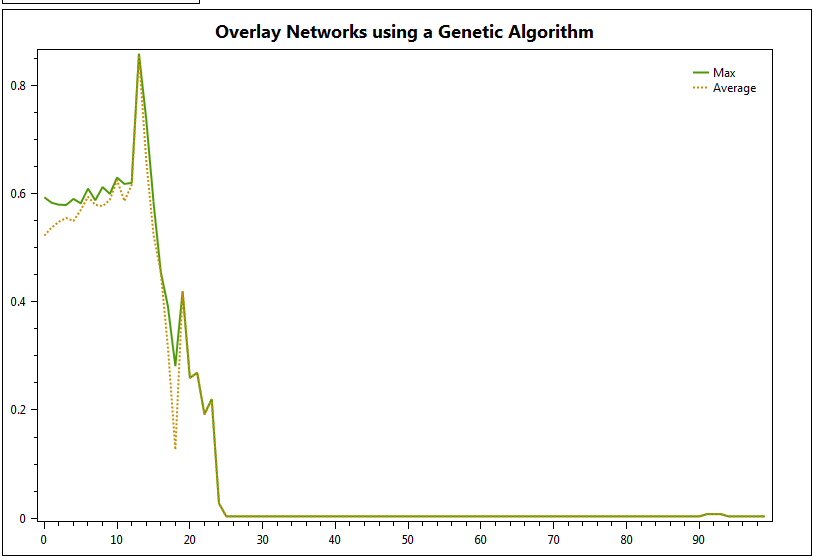
This is the Genetic Algorithm program to study overlay network optimization. Straightforward to implement, however the bulk of the time was spent finding a suitable graphing library, and another

large portion of the time was spent debugging reference pointer issues.

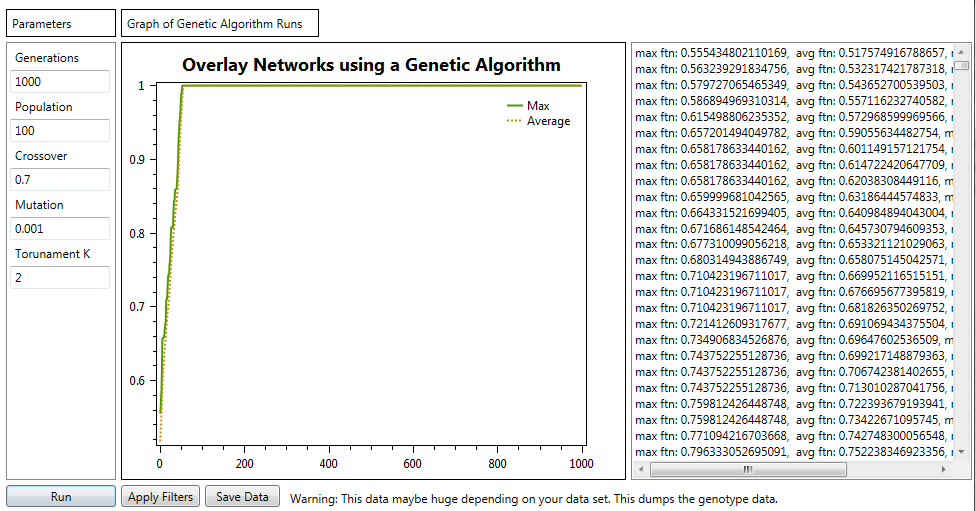
This is what a normal run looks like. We can see that in this version, the max is the local max of he population, and average is the average of the population. They are matching extremely closely. What I had first suspected was the random number generator. As it turned out, that was only a small factor.

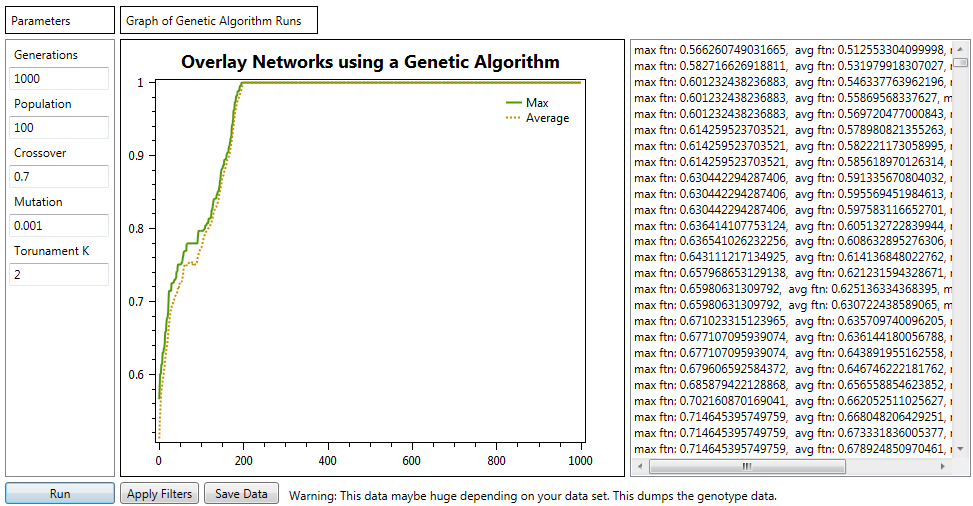
The following two examples are the algorithm running for 100 generations, this was quicker and easier to debug. The encoded genetic string was also reduced to 9 bits.



The occasional drop off would occur, about 1 out of 4 times I ran it. This didn’t seem right so I continued debugging and changed the Max to a global max instead.

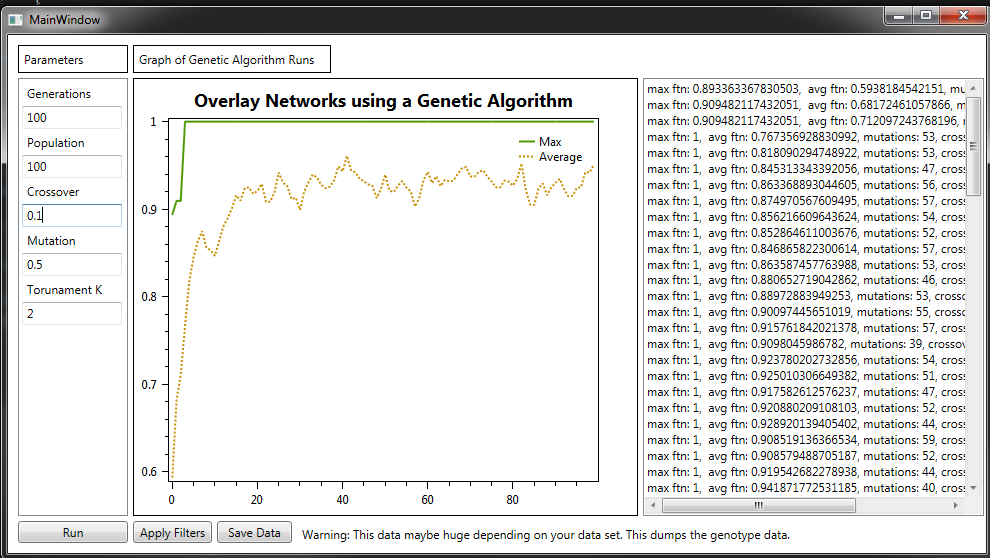
After about an hour of searching I had figured it out, it seems that a lot of my references were pointing back to the previous generation. A mutation occurring in one generation would mutate all the children of the same parent gene – this meant that in the selection round, the genes weren’t being copied, just brought over, and only the memory address was being copied. To fix this, I ensured that all the links were structs, in C# this means that it is not referenced, but is allocated dynamically in memory and stored on the stack and not memory heap. Also I changed the ‘chromosome’ class, this was essentially a wrapper class for the genes, which exclusively copied in the selection method.

This seems to fix the problem as we can see below.

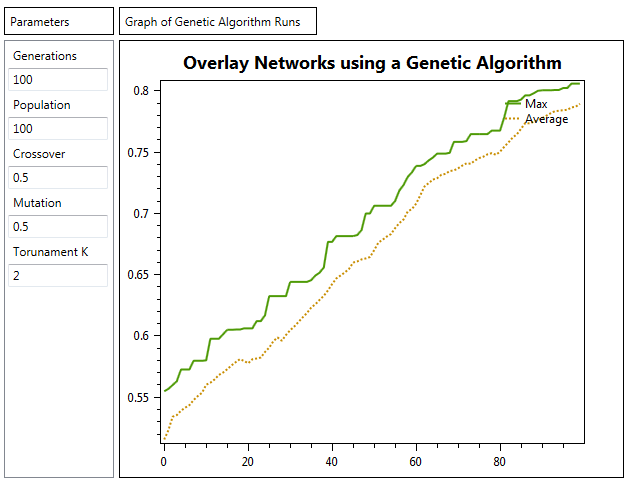


This is a quick test run with default parameters. Running this multiple times usually yields a similar result.

I liked the speed of 100 generations more since it showed me a more detailed graph, and I could easily match it to the side bar which told me in details the number of mutations and crossovers that occurred.

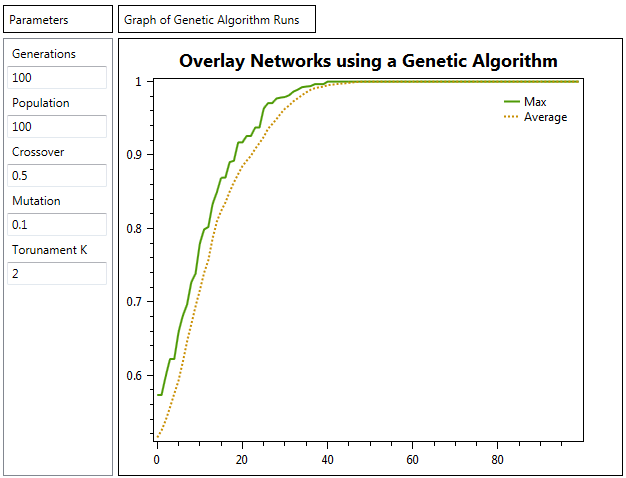


In these settings, the number of mutation was high, so we have a more unstable set, but it still found the maximum fairly quickly. So I tried increasing the crossover rate



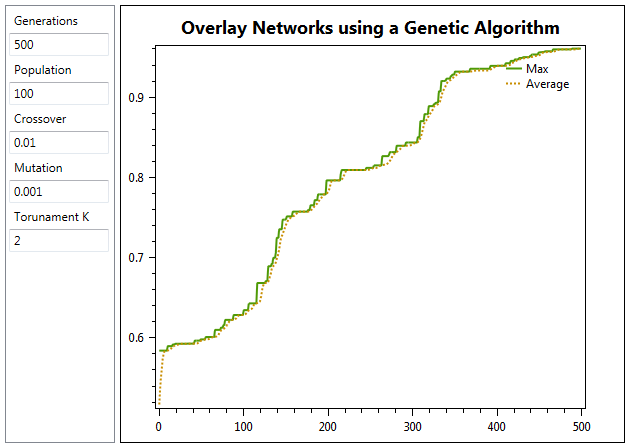
Some interesting results occurred here. We found the maximum value a lot slower this time. The average slowly climbed as mutations fought against crossovers changing the overall population giving it a variety of fitness values.

Decreasing the mutation gives us:



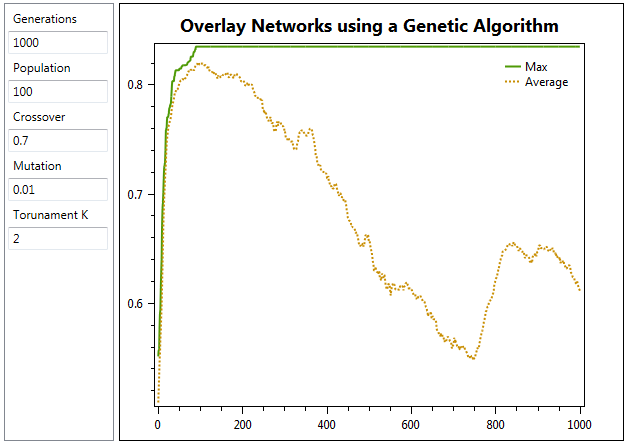
This result appeared to be similar to what the first hundred looked like on the default settings result.

Another thing I observed was that the lower the crossover was also would cause the algorithm to convers slower, and would take the algorithm significantly longer to find the max.



A lot of errors I found also occurred with the random function I was using, so I tried to randomize it by seeding an instance of another random function.

Here is a run where the overall average did not converge:



There’s a very high chance that it’s because of the seeded random function which is pseudo random.

**Conclusion:** Stability of the converging average plot line seems to rely much on the randomness of the algorithm, and the mutation rate. The rate at which we find the global max value with the highest fitness also seems to rely much on the crossover rate. The crossover allows more area between the global max and the average, a higher crossover can help find the max faster, but may cause the average to degrade over time. A well-tuned crossover and mutation rate can help the algorithm become more stable and find the max value much faster.