Chuck Edwall
23265927
AI Genetic Algorithm Project Report

I implemented the genetic algorithm in Java, using arrays of bits for the candidate strings. In my code, I call them *kids*. I started with a readInput function that reads the input files and creates from them a double-dimensional array called *kb* for "knowledge base." Each row of the array is a clause, and each element in each row is a variable number in that clause.

After I had a reliable way of generating the knowledge base, I then wrote code that tests a given kid (string of bits) against a clause to see if the clause is satisfied. I then used this to write a method that tests a given kid against *all* of the clauses in the knowledge base in order to determine its *fitness*, which is the number of clauses the kid does not satisfy. Lower fitnesses are better.

After I had this reliable, easy way of determining kid fitness, I then implemented a method that creates the first generation, which always consists only of random strings of bits.

Once I had a first generation, I could then work with it: I implemented the method called "nextGen," which begins by calculating the fitnesses of all the children in the current generation and Quicksorting them, in order to possibly return a solution and terminate. If it does not indeed terminate, then it creates the next generation. To do this, it takes a settable percentage of the fittest kids in the current generation and mates them with one another, then takes the remaining, less lucky folks and mutates them, inserting them into the next generation. Because the mated children only generate one offspring for every two children, there will be empty spaces in the next generation, which are filled in with entirely new random children.

The mating of two parents to yield a child happens as follows. There is a chance that the offspring of the two children will be slightly mutated. If it turns out that a child is not mutated, then it consists entirely of bits chosen randomly from either of the two parents, one bit at a time. If the new child *is* to be mutated, then a settable percentage of its bits are flipped randomly.

The mutation of a single parent into a new child is much simpler, consisting of only a simple bit-by-bit test of a random number to possibly flip each bit with a settable chance.

I have tested the program with a few different text files, including the one from the Word file in the assignment. The others were generated by the file called problemgen.c, which I obtained from the link inside the Word file.

My default values are a generation size of 300, a mating rate of 30 percent (so that the best 60 percent of the current generation produce 30 percent of the next generation), a 20-percent per-bit mutation flip chance for non-mating parents, a 30 percent chance that the children of two mating parents will be mutated, and, if a such child is indeed mutated, a 30 percent chance for each of the child's bits to flip.

I noticed the number of required generations to solve given problems drastically increase when I reduced the population count. Also, I tried an optimization in which I avoided the creation of new random children by mating each mated pair twice instead of once, ensuring that for every pair of parents that died, there would be two new children to replace them. I thought that this would make the algorithm run faster, due to the idea that mating highly fit parents would cause more fit offspring, but, to my surprise, this actually increased the average number of required generations by nearly a factor of three! This is most likely because the variation in the population from generation to generation was greatly reduced, causing the program to proceed down its best available known path instead of exploring new paths to possible solutions. Creating new random children drastically improves efficiency.

The algorithm has been able to solve problems for the past five hours of development at the time of the writing of this sentence and paragraph, and I could have turned this all in on time, but I just am too engrossed in optimizing it, and decided not to turn it in until I had it as good as I can brainstorm how to get it to run. I can now solve the 2,000-5,000 constraint problems in under 200 steps every single time I run a test. I did it by adding a third type of mutation. This type of mutation is called a *tiny mutation*, and, when it occurs, only a constant number of bits are changed. In this case, that number is random: either one or two. By default, the rate of eligibility for this mutation is only two percent, which means that the top two percent of the population will be finely experimented on by experiencing one to two bit flips into the next generation. *Another* new addition is a complete pass-through of the top three percent of the population, which, makes sure that the good candidates are preserved as they are actively mutated to finely tune them generation after generation. This way, the fittest of the fit can only ever improve, because the parents themselves survive in case their children are less fit than they.

I had that idea – the idea to perform a tiny single-bit mutation on only the best candidates – from when I let my program run on *five.txt* for about ten minutes, only to realize that, when I had given up, the fittest bit string it had obtained differed by only *one bit* from the actual solution given in the file. When I saw this, I then implemented that behavior, and got the running time from ~1,000 generations down to about 20 to 40!

I completed yet another optimization, which further increases the problem size feasible inside of 200 generations to *eight.txt,* which contains 100 variables and 1000 constraints. The optimization consists of an added chunk of code that actively removes duplicate children, in order to allow better variation among the most fit children in the population. Duplicate children were being created because the tiny mutations, mating, and pass-through were all operating on the same children in the front of the pack, possibly sending duplicates into the next generation. Removing these duplicate children allows them to interact more with each other than with copies of themselves, thus making the probability of finding those last few bits significantly higher. This lowers the running time of the program on *five.txt* from 20-40 generations to 8 to 16.

I am aware that I may be writing an annoying amount of text in this report, but I would really appreciate it if you could run my program on *five.txt* to see how it finds the solution. The program gives very good visual feedback, with running progress indication if runtime exceeds one second. My 2.66GHz Intel Core 2 Duo processor solves *five.txt* in about four to eight seconds; your processor, if not a Core 2 Duo, should still be sufficiently fast to be able to solve five.txt without being boring. I'm really shocked to see how these optimizations have been working!

By far, the hardest 3-SAT problem I have is in *seven.txt*, which contains 100 variables and 2,000 constraints, for an R-value of 20. It's way slower than the *five.txt*'s R-value of 1.67, due to the fact that it is just ridiculously big. My program solves it in about 100 to 150 generations, taking about one minute to run on my 2.66GHz Core 2 Duo machine.

With a population size of 300, my algorithm is definitely tuned to larger problems. The shortest amount of time a problem can take on my machine is about 2 milliseconds, but, with smaller populations, smaller problems are faster and larger problems are a bit slower. I spend a lot more time waiting for large problems, so the population size is high.

Following is the output of my program when passed *seven.txt*, which is the hardest problem I have.

```
Now running a maximum of 2000 generations using seven.txt as input.
There are 100 variables and 2000 clauses of maximum size 3.
--------------------------------------
By the way, a solution was given in a comment in the input:
0010110010000001000000001100011001100011010110011100110000011111101011111010101101100011001100010111
As a sanity check, I tested it, and it solution given does indeed satisfy the constraints.

--------------------------------------
    Processing now.
....................
    Progress: At Generation: 19.
              Top Fitnesses: 142 142 143 148 150 152 152 152 153 154
....................
    Progress: At Generation: 39.
              Top Fitnesses: 105 109 109 110 110 111 111 111 111 111
....................
    Progress: At Generation: 59.
              Top Fitnesses: 89 90 90 92 93 93 93 93 94 94
....................
    Progress: At Generation: 79.
              Top Fitnesses: 62 65 70 73 73 73 74 74 74 76
....................
    Progress: At Generation: 99.
              Top Fitnesses: 15 22 26 30 32 33 34 34 37 39
....................
    Progress: At Generation: 119.
              Top Fitnesses: 8 8 11 15 15 17 17 17 18 18
....................
    Progress: At Generation: 139.
              Top Fitnesses: 8 8 11 11 11 13 13 14 14 16
....................
    Progress: At Generation: 159.
              Top Fitnesses: 8 11 11 13 13 13 13 13 13 14
..........
--------------------------------------

Yeah. A solution is FOUND.
Clause Satisfaction:  C =
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111
Fitness:              F = 0. A BIG, BEAUTIFUL, ZERO!
Generation Count:     G = 168
Solution:             X = 0010110010000001000000001100011001100011010110011100110000011111101011111010101101100011001100010111
Compute Time was 60186ms.
```