Genetic Operators on Floating Point Parameters

J.R. Parker
Laboratory for Computer Vision
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada

Abstract

The single bit mutation and one point crossover operations are most commonly implemented on a chromosome that is encoded as a bit string. If the actual arguments are real numbers this implies a fixed point encoding and decoding each time an argument is updated. A method is presented here for applying these operators to floating point numbers directly, eliminating the need for bit strings The result accurately models the equivalent bit string operations, and is faster overall.

1. Introduction

The two operations that could be called the basis of genetic algorithms are the single bit mutation and the one point crossover. Together they provide the bulk of the variability in the parameter sets that is needed for the selection process to do its work. Both operations are applied to bit strings, which are the encoded form of the actual parameter. In the case where the parameters are symbolic, integer, or even fixed point this representation makes a great deal of sense, and is evocative of the original biological analogy.

For floating point (real) parameters the bit string representation requires a good deal of time spent converting to and from bit string form. Each time the objective function is evaluated the parameter must be decoded (converted into real form). Initially, each parameter must be encoded (converted to bit string form) so that the operators may be applied. In addition, some flexibility is lost - some work has been done with dynamic changing of the ranges and size of the bit string as the optimization proceeds, and this also requires encoding and decoding.

The idea of using real numbers directly has been discussed, but generally in the context of hybrid algorithms, and the operators used are not the same - they are statistical analogies. For example, Davis [DAVI91] suggests *real number mutation*, which is the replacing of a real number in a chromosome with a randomly selected real number. This is not a single bit mutation; indeed, it could change all of the bits in that number. *Real number creep* is also not a traditional operator, but is an additional one that can prove useful in some circumstances but which many purists eschew as being not general or robust.

What will be proposed here are two operators that are precise analogs of the bit string operations, but which apply to real numbers. Some of the advantages are obvious for continuous optimization problems; there are also some interesting by-products.

2. Single Bit Mutation

Without loss of generality the single bit mutation operator will be defined as one which reverses a specified bit in the encoded representation. Consider a 4-bit representation of the range 0.0 - 1.0; the values it is possible to represent are:

0.0 = 00 0.25 = 01 0.50 = 10 0.75 = 11

A bit in the low bit position has a value V=0.25 associated with it; more generally,

$$V = \frac{Dmax - Dmin}{2^n}$$

where *Dmax* is the largest value in the range, *Dmin* is the smallest, and *n* is the number of bits in the representation. In the example, $V=(1.0-0.0)/2^2$.

Each bit in the encoded bit string representation corresponds to a power of two multiplied by V. The low bit is $2^0 * V$, the next is $2^1 * V$ and so on. It seems apparent that to do a single bit reversal on a parameter *X* at bit position *n* is to subtract $2^n * V$ from *X* if bit *n* is set (1), or to add $2^n * V$ to *X* if bit *n* is clear (0).

Bit n of the fixed point bit string representation of X is set if the bit string is shifted right n bits has the low bit set; the corresponding floating point expression is to consider the expression $(X\text{-}Dmin)/(v*2^n)$. If this result has the low bit set then the bit corresponding to $v*2^n$ is set in the encoding.

This argument leads to the single bit mutation function seen in Figure 1. Note that the calculation of V would normally be done globally only once, and all values of $V*2^n$ could also be pre-computed. This leaves the operation count at one floating divide, one floating multiply, two floating subtractions and an integer division.

```
/* Implement a single bit mutation */

void fmut (float *b, int n, float xmax, float xmin, int size)
{
    double m, v, d3;

    v = (double)(xmax-xmin)/(double)(1<<size);
    m = (double)(1<<(size-n-1));
    if ( (int)((*b-xmin)/(v*m)) % 2 )
        *b = *b - v*m;
    else
        *b = *b + v*m;
}
```

Figure 1 - C code to perform a single bit mutation on a real argument **b**.

2. One Point Crossover

The crossover operation is a little more complex, but is based on the same ideas. Consider the situation where an argument has a size of 10 bits and a range from 0.0 to 1.0. Two instances of this argument, X and Y, are to be crossed over at bit 4; this situation is:

Argument	Bit Value	Real Value	Crossed Bit Value	Resulting Real
X	0110101010	.416016	0110010011	.393555
Y	0010010011	.143555	0010101010	.166016

This is accomplished by taking the high 4 bits of X and adding to it the low 6 bits of y (and vice versa). In real terms, the low 6 bits can represent values up to $(2^7\text{-}1)$*V, and the high 4 bits represent values from 2^7*V upwards. Thus, the high 4 bits of the real value X is given by:

$$X_h = \left\lfloor \frac{X - Dmin}{V \times 2^n} \right\rfloor \times V \times 2^n$$

for *n=4*, which effectively masks out the low 6 bits by truncation and then shifts the value left by the proper amount. The low 6 bits are obtained by simply subtracting this from X:

$$X_l = X - X_h$$

```
/* Perform a 1 point crossover */
void fcross1 (float *s1, float *s2, int n, float xmin, float xmax, int size)
{
        int i, k, c1;
        double xl, xh, yl, yh;
        double v, j;

        v = (double)(xmax-xmin)/(double)(1<<size);
        j = (double)(1 << (size-n));
        xh = (int)((*s1-xmin)/(j*v)) * (j*v);
        xl = (*s1) - xh;
        yh = (int)((*s2-xmin)/(j*v)) * (j*v);
        yl = (*s2) - yh;
        xh = xh + yl ;
        yh = yh + xl ;
        *s1 = (float)xh;
        *s2 = (float)yh;
}
```

Figure 2 - C code to perform a one point crossover on real arguments s1 and s2.

Now do the same for Y, and finally let $X = X_h+Y_l$ and $Y = Y_h+X_l$, as seen in the code for **fcross** in Figure 2.

3. Evaluation of the Methods

The single bit mutation function has been tested on tens of thousands of mutations using various experimental protocols. For an example, a program has been written that generates random whole-number ranges, bit string sizes, and values and then creates a bit string from the generated values. All possible one bit mutations were carried out, both on the bit string and the real value, and the results were compared. The worst case error, the difference between the two values, was 0. For this same experiment the worst case error for the one point crossover was 5%; this occurred in one case out of 1000. Two more cases had an error of 2%, and all of the rest were under 1% with the average error being 0.0082%.

Speed of execution is another important issue, but a more difficult one to evaluate. There is a very good chance that some else could code a faster bit string module than the one used in the evaluation. Still, some evaluation must be done and, after all, all of the code was written by the same person. For the execution profile done on a SPARCstation 2, an example that performed 2000 crossovers and 30000 mutations had the following performance:

Method	Time for Mutations	Time for crossovers
Real numbers	0.25	0.02
Bit strings	0.365	0.09

This experiment is flattering to the bit string representation in that some of the global overhead is not included, and the Sun workstation does not have floating point acceleration.

4. Dynamic Sizes and Ranges

When using real arguments in the way described thus far the notion of chromosome length is a fiction. The size and range associated with any argument is only of importance when a mutation or crossover is being performed. It is therefore a simple task to change these, or possibly even make them parameters themselves. Increasing the size of the bit string has the effect of decreasing V, which means that the precision of the search is improved. There is no significant implementation overhead involved in using large sizes up to the maximum precision of a floating point number.

Changing the ranges while performing an optimization can be done if it can be determined with some reliability that the argument involved does not have an optimal value in the outlying regions of the range. Decreasing the range while maintaining a constant size has the effect of refining the step size (decreasing V again) and increasing the precision of the result while improving the speed with which the optimal value can be achieved. To modify the range in a bit string implementation requires a decoding of the argument, a modification of the range, and then an encoding at the new range. The real implementation does not require a change in the encoding, and achieves high accuracy simulation of the bit string results provided that the ranges have acceptable numerical properties. Integral ranges and powers of two are best, while repeating fractions and irrational numbers are worst.

5. Conclusions and Further Work

A methods for modelling the effects of single bit mutations and one point crossovers using real arguments has been described. It has a high similarity to the equivalent bit string operations, and is not a statistical approximation; moreover, it executes at an acceptably high speed. This method has been used in a stellar photometry system now under development, and will be used in future in applications to computer vision and image processing.

The method could be easily applied to multiple mutation and crossover, and to other operators used in genetic algorithms.

This work has been supported by the Natural Sciences and Engineering Research Council of Canada.

6. References

[DAVI91] Davis, L. (ed), Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York NY, 1991.

[GOLD89] Goldberg, D.E., Genetic Algorithms, Optimization, and Machine Learning, Addison-Wesley, Reading MA. 1989.

[HOLL75] Holland, J.H., Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, MI. 1975.

[PARK95] Parker, J.R., A Genetic Algorithm for Stellar Photometry, University of Calgary Department of Computer Science Research Report #xxx. 1995.

[SCHR90] Schraudolph, N.N. and Belew, R.K., Technical Report. #LAUR 90-2795, Los Alamos National Laboratory, 1990.