International Journal of Modern Physics C
Vol. 16, No. 11 (2005) 1811–1816

World Scientific
www.worldscientific.com

GENETIC ALGORITHMS, FLOATING POINT NUMBERS AND APPLICATIONS

YORICK HARDY and WILLI-HANS STEEB*
International School for Scientific Computing
University of Johannesburg, P. O. Box 524
Auckland Park 2006, South Africa
**whs@na.rau.ac.za*

RUEDI STOOP
Institut für Neuroinformatics, ETHZ/UNIZH
Winterthurerstr. 190, 8057 Zürich, Switzerland

Received 27 May 2005
Revised 8 June 2005

The core in most genetic algorithms is the bitwise manipulations of bit strings. We show that one can directly manipulate the bits in floating point numbers. This means the main bitwise operations in genetic algorithm mutations and crossings are directly done inside the floating point number. Thus the interval under consideration does not need to be known in advance. For applications, we consider the roots of polynomials and finding solutions of linear equations.

Keywords: Genetic algorithms; crossing; mutation; floating point numbers.

Genetic algorithms are a family of computational models inspired by evolution.[1–3] These algorithms encode a potential solution to a specific problem on a simple chromosome-like data structure and apply recombination operators (crossing, mutation) to these structures so as to preserve critical information. Genetic algorithms are often used to find the optimum of function. Here we consider partial and total functions over the real numbers. In the case of one-dimensional standard genetic algorithms, we start from a collection of bit strings "$s_{N-1}s_{N-2}\cdots s_0$", the so-called farm. We first calculate the integer number

$$m = \sum_{j=0}^{N-1} s_j 2^j$$

and then we map m into the floating point number x via

$$x = a + m\frac{b-a}{2^N - 1}\,,$$

where $[a, b]$ $(a < b)$ is the interval on which we are looking for the minimum or maximum of a given fitness function f. Finally we calculate the fitness $f(x)$ for the given fitness function f which we want to minimize or maximize. For the bitwise manipulations one would use a Bitset class. Both C++ (in the Standard Template Library) and Java provide a Bitset class. The bitwise manipulation can also be done on integers of data type `byte`, `short`, `int`, `long` and then mapped into the floating point numbers (see for example Ref. 4, which also provides a Fortran program). This is essentially a fixed point representation. Thus we should know the domain of the problem in advance.

Here we show that one can directly manipulate the bits in the floating point numbers. This means the main bitwise operations in genetic algorithm mutations and crossings are directly done inside the floating point number. Modern CPU's have very fast floating point processors and we could even use inline assembler in our C++ program to use the floating point instructions. If we use integers in our genetic algorithm and use floating point numbers for calculations, there is an additional cost in converting from integer to floating point representation. In some problems, such as the knapsack problem and four color problem,[3] integer representations have a clear advantage. However, when finding extrema of a function defined on the entire real axis it is generally necessary to restrict the genetic algorithms to a reasonable sub-interval when using an integer representation. If we work directly with double precision floating point numbers (64 bits), numbers as small as $2^{-1022} < 10^{-307}$ and large as $2^{1023} > 10^{308}$ can be represented. With extended double precision (80 bits) an even greater range is possible. Thus the genetic algorithm can evolve the population to specialize on appropriate sub-intervals within the bounds of double precision floating point numbers. This advantage does come at the cost of accuracy when large numbers are involved. Furthermore, the structure of the floating point number need not be know when applying the recombination operators since it is simply viewed as a bit string. We can also extend this method to apply it to an array of floating point numbers, for example to solve systems of linear equations.

As an application, we consider the roots of polynomials and the solutions of linear equations.

We consider the data type `double`. After the IEEE standard 754 for floating point numbers `double` (64 bits) is stored as

```
sign bit, 11 bit exponent, 52 bit mantissa
```

This means

```
  byte 1    byte 2     byte 3     byte 4   ...  byte 8
SXXX XXXX  XXXX MMMM  MMMM MMMM  MMMM MMMM ... MMMM MMMM
```

Bits are counted from right to left starting at 0. Thus the sign bit is at position 63. However, we do not need to know the structure of the floating point numbers in order to apply bitwise manipulations and the genetic algorithm. The two main

operations would be crossing and mutation. For crossing we select two random numbers between 0 and 63 and then we swap these parts of the two bitstrings. In mutation we select a bit in the bitstring at random and then swap it.

To understand the basic idea to access a bit in a floating point number consider the following C++ code excerpt:

```
double x = 3.14159;
int* p = (int*) &x;
*(p+1) = *(p+1)^(1<<31); // short cut *(p+1) ^= (1<<31)
cout << "x = " << x << endl;
```

Note that the data type `double` occupies 64 bits and the sign bit is at bit position 63. The data type `int` occupies 32 bits, $\ll$ is the shift left operation and `^` is the XOR operation. The type `double` is 64 bits and type `int` is 32 bits. Thus two consecutive `int`s make up a double. The line `int* p =(int*) &x`; declares an integer pointer `p` which points to the variable `x` of type `double`. Thus the pointer `p` allows us to interpret the memory location of `x` as an integer, and consequently we can perform operations defined on integers on this memory location. Thus we can directly access the first 32 bits via `*p`. To access the second 32 bits of the `double` we use the pointer `p+1` which advances the pointer by one `int`, i.e., 32 bits. Thus the line $*(\mathtt{p}+1) = *(\mathtt{p}+1)\hat{}(1 << 31)$; takes the second 32 bits of `x` (bit positions 32 to 63 numbered from zero) interpreted as an integer, and does bitwise manipulations on the bits. The XOR operation `^` is used to swap the 31st bit of `*(p+1)` or equivalently the 63th bit of `x`. Thus `x` contains `-3.14159`, i.e., we swap the sign bit.

We must keep in mind that IEEE reserves exponent field values of all 0 s and all 1 s to denote special values in the floating point scheme. If the exponent is all 0 s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a denormalized number, which does not have an assumed leading 1 before the binary point. For `double` precision, denormalized numbers are of the form $(-1)^s \cdot 0.f \cdot 2^{-1022}$. From this we can interpret zero as a special type of denormalized number. The values +infinity and −infinity are denoted with an exponent of all 1 s and a fraction of all 0 s. The sign bit distinguishes between negative infinity and positive infinity. The value NaN (Not a Number) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1 s and a non-zero fraction. There are two categories of NaN: QNaN (Quiet NaN) and SNaN (Signaling NaN). A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values are generated by an operation when the result is not mathematically defined. An SNaN is a NaN with the most significant fraction bit clear. It generates an exception when used in operations.

We write the code using template functions so that it also be used for the data type `float`. The function `mutate()` implements the mutation and the function `cross()` implements the crossing operation. Both use the function `locate()`.

```
template <class T>
unsigned int &locate(int bit,T &x)
{
 unsigned int *p = (unsigned int *)&x;
 static const int bitsperT = sizeof(T)*8;
 assert(bit>=0 && bit<bitsperT);
 return *(p + (bit/bitsperint));
}

template <class T>
void mutate(int bit,T &x)
{ locate(bit,x) ^= (1 << (bit % bitsperint)); }

template <class T>
void cross(int bit1,int bit2,T &x1,T &x2)
{
 if(bit1>bit2) { bit2 ^= bit1; bit1 ^= bit2; bit2 ^= bit1; }
 unsigned int *p1 = &locate(bit1,x1);
 unsigned int *p2 = &locate(bit1,x2);
 if(bit1 != 0)
 {
 int a = (*p1) & (numeric_limits<unsigned int>::max()^((1<<bit1)-1));
 int b = (*p2) & (numeric_limits<unsigned int>::max()^((1<<bit1)-1));
 a ^= b; b ^= a; a ^= b;
 *(p1++) &= a; *(p2++) &= b;
 bit1 += bitsperint-(bit1%bitsperint);
 }
 for(;bit2-bit1 > bitsperint;bit1 += bitsperint,p1++,p2++)
 { (*p1) ^= (*p2); (*p2) ^= (*p1); (*p1) ^= (*p2); }
 if(bit2-bit1 != 0)
 {
 int a = (*p1) & ((1 << (bit2-bit1))-1);
 int b = (*p2) & ((1 << (bit2-bit1))-1);
 a ^= b; b ^= a; a ^= b;
 *p1 &= a; *p2 &= b;
 }
}
```

If we know that the values for the optimum lie only in the range $x \geq 0$ or $x \leq 0$, then we can leave the sign bit untouched.

Obviously the manipulations of the bits in the mantissa contribute to a smaller change in x then the manipulation of the bits in the exponent. This could be taken

into account at the beginning of the genetic algorithm by giving a bigger weight to the manipulations of the bits in the exponent part.

For the two examples, we consider the real roots of polynomials. The roots are given by the solution of the equation $p(x^*) = 0$. We can select $f(x) = -p^2(x)$ as the fitness function f, which has to be maximized, i.e., the zeros of the polynomial p are found where f takes a global maximum.[5] Obviously the global maximum of f is 0. Another possible fitness function would be $f(x) = |p(x)|$. This fitness function has to be minimized. For faster calculation of $f(x)$, one uses Horner's scheme.[5] Consider first

$$p(x) = x^4 - 7x^3 + 8x^2 + 2x - 1\,.$$

The roots are positive and negative, namely -0.405717, 0.271902, 1.65434 and 5.47947. With the farm large enough we obtain all four roots.

An important special case is the calculation of the eigenvalues of a density matrix. A density matrix ρ is a positive definite matrix (and thus hermitian) with trace 1. Since the trace of a square matrix is the sum of the eigenvalues we find that for all eigenvalues $0 \leq \lambda \leq 1$. Thus in this case we would only do the bitwise manipulation of the mantissa. As an example consider the density matrix

$$\rho = \begin{pmatrix} 3/8 & 0 & 0 & 1/4 \\ 0 & 1/8 & 0 & 0 \\ 0 & 0 & 1/8 & 0 \\ 1/4 & 0 & 0 & 3/8 \end{pmatrix}\,.$$

Then the characteristic equation is given by

$$\lambda^4 - \lambda^3 + 0.28125\lambda^2 - 0.03125\lambda + 0.001220703125 = 0\,.$$

The eigenvalues are given by 0.125 (three times) and 0.625. In a typical run, a bigger part of the farm tends to the three times degenerate eigenvalue 0.125 and only a smaller part to the single eigenvalue 0.625.

The method has also be successfully applied for solving linear equations $A\mathbf{x} = \mathbf{b}$, where A is an $n \times n$ matrix over $\mathbf{R}$. A possible fitness function is[5]

$$f(\mathbf{x}) = -\left(\frac{1}{n}\sum_{i=0}^{n-1}\left|\sum_{j=0}^{n-1} a_{ij}x_j - b_i\right|\right)\,,$$

where a_{ij} are the matrix elements of A. Thus for the vector $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$, we deal with a one-dimensional array of data type `double`.

References

1. W.-H. Steeb, *The Nonlinear Workbook*: *Chaos*, *Fractals*, *Cellular Automata*, *Neural Networks*, *Genetic Algorithms*, *Gene Expression Programming, Support Vector Machine*, *Wavelets*, *Hidden Markov Models*, *Fuzzy Logic with C++*, *Java and Symbolic C++ Programs*, 3rd edn. (World Scientific, 2005).

2. Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd edn. (Springer-Verlag, Berlin, 1996).
3. Y. Hardy and W.-H. Steeb, *Classical and Quantum Computing with C++ and Java Simulations* (Birckhäuser-Verlag, Basel, 2001).
4. S. M. de Oliveira, P. M. C. de Oliveira and D. Stauffer, *Evolution, Money, War and Computers* (Teubner-Verlag, Stuttgart and Leipzig, 1999).
5. W.-H. Steeb, Y. Hardy, A. Hardy and R. Stoop, *Problems and Solutions in Scientific Computing with C++ and Java Simulations* (World Scientific, Singapore, 2004).