*International Journal of Computer and Information Technology (ISSN: 2279 – 0764)*
*Volume 05 – Issue 06, November 2016*

# The Extension of Digit Inversion in Genetic Algorithm

Ixymbayeva S.Zhanyl
Computer Engineering and Software, Faculty of S.Seifullin Kazakh AgroTechnical University, Astana, Email: Kazakhstan zhanyl.x60 [AT] gmail.com

Nayzagarayeva A.Akkul
Infotmation Systems, Faculty of S.Seifullin Kazakh AgroTechnical University, Astana, Kazakhstan

Tabys Tannurbek
Computer Engineering and Software, Faculty of S.Seifullin Kazakh AgroTechnical University, Astana, Kazakhstan

*Abstract*--- **In this paper we proposed stated, a Genetic algorithms (GA) are machine learning search techniques inspired by Darwinian evolutionary models. The advantage of GA over factor analytic and other such statistical models is that GA models can address problems for which there is no human expertise or where the problem seeking a solution is too complicated for expertise based approaches.**

***Keywords***— a Genetic algorithm (GA), inversion, assembly language

## I. INTRODUCTION

Genetic algorithms (GA) are machine learning search techniques inspired by Darwinian evolutionary models. The advantage of GA over factor analytic and other such statistical models is that GA models can address problems for which there is no human expertise or where the problem seeking a solution is too complicated for expertise based approaches. GA can be applied to challenges which can be formulated as function optimization problems. This makes GA ideal for application to discrete combinatorial problems and mixed-integer problems [1].

Before a GA can be put to work on any problem, a method is needed to encode potential solutions to that problem in a form that a computer can process. One common approach is to encode solutions as binary strings: sequences of 1's and 0's, where the digit at each position represents the value of some aspect of the solution. Another, similar approach is to encode solutions as arrays of integers or decimal numbers, with each position again representing some particular aspect of the solution. This approach allows for greater precision and complexity than the comparatively restricted method of using binary numbers only and often "is intuitively closer to the problem space" [2]

This technique was used, for example, in the work of Steffen Schulze-Kremer, who wrote a genetic algorithm to predict the three-dimensional structure of a protein based on the sequence of amino acids that go into it [3]. Schulze-Kremer's GA used real-valued numbers to represent the so-called "torsion angles" between the peptide bonds that connect amino acids. (A protein is made up of a sequence of basic building blocks called amino acids, which are joined together like the links in a chain. Once all the amino acids are linked, the protein folds up into a complex three-dimensional shape based on which amino acids attract each other and which ones repel each other. The shape of a protein determines its function.)

Before you can use a GA to solve a problem, a way must be found of encoding any potential solution to the problem. This could be as a string of real numbers or, as is more typically the case, a binary bit string. Let's refer to this bit string from now on as the chromosome. A typical chromosome may look like this:

100101011101010010100111011011101111111101

At the beginning of a run of a GA a large population of random chromosomes is created. Each one, when decoded will represent a different solution to the problem at hand. Let's say there are N chromosomes in the initial population [4-6]. Then, the following steps are repeated until a solution is found

•Test each chromosome to see how good it is at solving the problem at hand and assign a fitness score accordingly. The fitness score is a measure of how good that chromosome is at solving the problem to hand.

•Select two members from the current population. The chance of being selected is proportional to the chromosomes fitness. Roulette wheel selection is a commonly used method.

•Dependent on the crossover rate crossover the bits from each chosen chromosome at a randomly chosen point.

•Step through the chosen chromosomes bits and flip dependent on the mutation rate.

•Repeat step 2, 3, 4 until a new population of N members has been created.

## II. ROULETTE WHEEL SELECTION

This is a way of choosing members from the population of chromosomes in a way that is proportional to their fitness. It does not guarantee that the fittest member goes through to the next generation, merely that it has a very good chance of doing so. It works like this:

Imagine that the population's total fitness score is represented by a pie chart, or roulette wheel. Now you assign a slice of the wheel to each member of the population. The size of the slice is proportional to that chromosomes fitness score. i.e. the fitter a member is the bigger the slice of pie it gets. Now, to choose a chromosome all you have to do is spin the ball and grab the chromosome at the point it stops.

This is simply the chance that two chromosomes will swap their bits. A good value for this is around 0.7. Crossover is performed by selecting a random gene along the length of the chromosomes and swapping all the genes after that point.

e.g. Given two chromosomes

**10001001110010010**
**01010001001000011**

Choose a random bit along the length, say at position 9, and swap all the bits after that point

So the above become:

10001001101000011
01010001010010010

## III. WHAT'S THE MUTATION RATE?

This is the chance that a bit within a chromosome will be flipped (0 becomes 1, 1 becomes 0 - this operation is called digit inversion. We give an example of inversion realization on low level programming language assembler - appendix 1[7]). This is usually a very low value for binary encoded genes, say 0.001

So whenever chromosomes are chosen from the population the algorithm first checks to see if crossover should be applied and then the algorithm iterates down the length of each chromosome mutating the bits if applicable.

## IV. PRACTIC IMPLEMENTATION

To hammer home the theory you've just learnt let's look at a simple problem:

Given the digits 0 through 9 and the operators +, -, * and /, find a sequence that will represent a given target number. The operators will be applied sequentially from left to right as you read.

So, given the target number 23, the sequence 6+5*4/2+1 would be one possible solution.

If 75.5 is the chosen number then 5/2+9*7-5 would be a possible solution.

Please make sure you understand the problem before moving on. I know it's a little contrived but I've used it because it's very simple.

### Stage 1: Encoding

First we need to encode a possible solution as a string of bits… a chromosome. So how do we do this? Well, first we need to represent all the different characters available to the solution... that is 0 through 9 and +, -, * and /. This will represent a gene. Each chromosome will be made up of several genes.

Four bits are required to represent the range of characters used:

| | |
|---|---|
| **0**: | **0000** |
| **1**: | **0001** |
| **2**: | **0010** |
| **3**: | **0011** |
| **4**: | **0100** |
| **5**: | **0101** |
| **6**: | **0110** |
| **7**: | **0111** |
| **8**: | **1000** |
| **9**: | **1001** |
| **+**: | **1010** |
| **-**: | **1011** |
| *****: | **1100** |
| **/**: | **1101** |

The above show all the different genes required to encode the problem as described. The possible genes **1110** & **1111** will remain unused and will be ignored by the algorithm if encountered.

So now you can see that the solution mentioned above for 23, ' 6+5*4/2+1' would be represented by nine genes like so:

**0110 1010 0101 1100 0100 1101 0010 1010 0001**
**6 + 5 * 4 / 2 + 1**

These genes are all strung together to form the chromosome:

**011010100101110001001101001010100001**

Because the algorithm deals with random arrangements of bits it is often going to come across a string of bits like this:

**0010001010101110101101110010**

Decoded, these bits represent:

**0010 0010 1010 1110 1011 0111 0010**
**2 2 + n/a - 7 2**

Which is meaningless in the context of this problem! Therefore, when decoding, the algorithm will just ignore any genes which don't conform to the expected pattern of: number -> operator -> number -> operator …and so on. With this in mind the above 'nonsense' chromosome is read (and tested) as:

**2 + 7**

### Stage 2: Deciding on a Fitness Function

This can be the most difficult part of the algorithm to figure out. It really depends on what problem you are trying to solve but the general idea is to give a higher fitness score the closer a chromosome comes to solving the problem. With regards to the simple project I'm describing here, a fitness score can be assigned that's inversely proportional to the difference between the solution and the value a decoded chromosome represents.

If we assume the target number for the remainder of the tutorial is 42, the chromosome mentioned above

**011010100101110001001101001010100001**

has a fitness score of 1/(42-23) or 1/19.

As it stands, if a solution is found, a divide by zero error would occur as the fitness would be 1/(42-42). This is not a problem however as we have found what we were looking for... a solution. Therefore a test can be made for this occurrence and the algorithm halted accordingly.

**Stage 3: Getting down to business**

Please tinker around with the mutation rate, crossover rate, size of chromosome etc to get a feel for how each parameter effects the algorithm. Hopefully the code should be documented well enough for you to follow what is going on!

Note: The code given will parse a chromosome bit string into the values we have discussed and it will attempt to find a solution which uses all the valid symbols it has found. Therefore if the target is 42, + 6 * 7 / 2 would not give a positive result even though the first four symbols("+ 6 * 7") do give a valid solution.

## V. CONCLUSIONS

There are different selection techniques to use, different crossover and mutation operators to try and more esoteric stuff like fitness sharing and speciation to fool around with. All or some of these techniques will improve the performance of your GA- s considerably.

## REFERENCES

1Sivanandam S, Deepa S: Introduction to Genetic Algorithms. 2008, Springer

2 Fleming and Purshouse 2002, Genetic algorithms in control systems engineering p. 1228.

3 Schulze-Kremer, Steffen. Molecular bioinformatics: algorithms and applications. - Berlin; New York; de Gruyter, 1995.

4 Painter TS (1933). "A new method for the study of chromosome rearrangements and the plotting of chromosome maps". *Science* 78 (2034): 585–586. doi:10.1126/science.78.2034.585. PMID 17801695.

5 Gardner R.J.M. and Sutherland G.R. 2004. Chromosome abnormalities and genetic counseling. Oxford.

6 Lehtonen S, Myllys L, Huttunen S (2009). "Phylogenetic analysis of non-coding plastid DNA in the presence of short inversions". Phytotaxa 1: 3–20.

7 Hyde, Randall. "Chapter 12 – Classes and Objects". The Art of Assembly Language, 2nd Edition. No Starch Press. © 2010

## Appendix 1 – Listing Example of Digit Inversion on Assembly language

```
.model small
.stack 256
.386
.data
   b dw 19
   d dw 45
   mode db 0
   endOfLine db 13, 10, '$'
   mes_bx db 'BX:', 13, 10, '$'
   mes_dx db 'DX:', 13, 10, '$'
   mes_pr db 'Direct: $'
   mes_ob db 'Inverse: $'
.code
main proc
.startup
   mov ah, 09h
      lea dx, mes_bx
   int 21h
      lea dx, mes_pr
   int 21h
   mov bx, b
   call OutputDirect
      lea dx, mes_ob
   int 21h
   mov mode, 1
   call OutputDirect
      lea dx, endOfLine
   int 21h

   lea dx, mes_dx
   int 21h
      mov bx, d
      lea dx, mes_pr
   int 21h
   mov mode, 0
   call OutputDirect
      lea dx, mes_ob
   int 21h
   mov mode, 1
   call OutputDirect
.exit
main endp;------------------------------------------------------------
OutputDirect proc near
;------------------------------------------------------------
   pusha
      cmp mode, 0
   je @skip
      not bx
      @skip:
      mov ah, 02h
   mov cx, 16
      @cycle:
      shl bx, 1
      jb @one
         mov dl, '0'
         jmp @output
      @one:
         mov dl, '1'
      @output:
         int 21h
               loop @cycle
      mov ah, 09h
   lea dx, endOfLine
   int 21h
      popa
   ret
OutputDirect endp
End
```