

# GENETIC ALGORITHMS

## *Mutations and Pattern Matching in Genetic Algorithms*

Drew Alex Clinkenbeard

*Computer Science, CSU Channel Islands, One University Drive, Camarillo CA, USA*  
[drew.clinkenbeard@csuci.edu](mailto:drew.clinkenbeard@csuci.edu)

Keywords: Genetic Algorithms, Mutation, Pattern Matching.

Abstract: Genetic Algorithms are excellent tools for pattern matching. They closely resemble natural phenomena and take many queues from the same. In this paper we examine how a standard Genetic Algorithm operates as well as how a modified implementation functions. We pay special attention to the importance of mutations in achieving a timely result.

## 1 INTRODUCTION

The natural world has taught humanity a great many things. So it is no surprise that it is to the natural world that computer scientists look to find new techniques and procedures for problem solving. This paper hopes to examine one such technique called Genetic Algorithms.

Genetic Algorithms (GAs) belong in a set of heuristic optimization algorithms that call upon the Darwinian theory of evolution to solve problems. A GA will typically be coded to reduce a problem to a bit string that may then be manipulated using techniques similar to biological processes.

In a Genetic Algorithm the basic data type is treated like a chromosome or allele. These chromosomes are then manipulated until a goal state is reached. The fitness of each chromosome is determined according to the goal state of the problem being solved. Since the solution for an NP-Complete problem is easy to verify GAs are often used to solve them. However since 'GAs are well suited for optimizing combinatorial problems' (Gary, Hart, Panton, Philips, Trahan, Wagmer, 1997) they can be used for more than just the realm of NP-C problems.

## 2 TYPICAL GENETIC ALGORITHM

Data manipulation in a GA typically has three distinct steps: selection, reproduction, and evaluation. These steps are, in fact, what defines a GA. We will now examine the GA presented in the text: Fundamentals of Natural Computing (de Castro, 2006)

It goes without saying that in order for any of these steps to be performed there must be data present. The generation of the data is irrelevant to the procedure as long as the data can be manipulated. As mentioned earlier the data here will be reduced to a bit string to ease manipulation as well as evaluations. The algorithm in question uses a  $[n, m]$  matrix of bits which is in turn translated into a visual grid.

### 2.1 Selection

In the example from the book Roulette Wheel (RW) selection is performed in order to determine breeding pairs. Roulette Wheel selection is a weighted random selection that uses the fitness of a chromosome to determine how likely it is to be chosen. Chromosomes that are more fit will occupy a larger section of the wheel and thus will be more likely to be picked. Each time a new breeder is required a selection is made from the weighted 'wheel'.

Do to the random nature of RW selection it is not certain that the fittest element will be chosen. This introduces a level of randomness that allows for a more diverse selection. It is entirely possible that an unfit parent may produce very fit offspring. However an unfit parent may also introduce undesirable traits. Lengthening the refinement process.

Randomness is very important as GAs are essentially hill climbing algorithms and as such are susceptible to local minima and maxima. The randomness of RW selection combined with aberration introduced in breeding ensure that local maxima and minima are severely reduced if not eliminated altogether.

### 2.2 Breeding

There are two methods applied to the recombination of the breeding pairs to return varied offspring. One is called crossing-over, the other is point mutation. We will first examine crossing over.

#### 2.2.1 Crossing Over

Crossover has been observed as the reason why traits associated with one chromosome are sometimes not completely present. (Nelson, 2004) Further Crossover is the method by which alleles recombine to form a new pattern. (de Castro, 2006) This process is replicated in Genetic Algorithms by splitting the chromosomes to be mated and recombining them into a new form.

Generally speaking, in single point crossover, the two breeding strings,  $x$  and  $y$ , have a crossover point chosen at random. The random point,  $r$ , is then used to split both strings leaving four string behind:  $x$ ,  $x'$ ,  $y$ , and  $y'$ . These strings are then concatenated resulting in two strings that are fundamentally different than their parents. (de Castro, 2006) see table 1.

Table 1: An illustration of crossing over.

Before crossover	x x x x   x' x' x' x' y y y y   y' y' y' y'
After crossover	x x x x y' y' y' y' y y y y x' x' x' x'

### 2.2.2 Point Mutation

Point mutation is the process by which one nucleotide is substituted for a different nucleotide resulting in a change in the single base pair of the DNA. (NLM, 2009). This is a rare occurrence in nature but it is often deemed a necessary evolutionary step. If no mutation occurs then the only change an organism will experience is a reshuffling of existent traits.

The algorithm introduced in de Castro considers each element in an array for mutation. Though the odds are extremely low each element has a chance of being changed or not. This leads to a low rate of mutation, much like in nature, however every possible bit is susceptible to mutation.

### 2.3 Evaluation

Evaluation is often the most difficult and time consuming portion of the entire operation. It is this stage where the fitness of the results is gauged. The fitness is what determines the success of the current run as well as how likely the chromosomes will be chosen in the next iteration.

The GA in de Castro utilizes a hamming distance between the individual rows of the results and the desired goal. This relates directly to the difference between the individual rows of generated data and the goal data. When the hamming distance is zero the goal has been achieved.

Hamming distance is an excellent metric to determine the distance one string must travel to reach an intended state. The only issue with the hamming distance is the number of calculations that must be performed to determine the distance.

## 3 MODIFIED GENETIC ALGORITHM

The success of a GA largely hinges on the accuracy of its fitness algorithm. In nature the most fit organism is the one that is best suited to its environment and therefore best able to breed. In Genetic Algorithms this is almost exactly the case. The fitness of a GA is determined not by how well it camouflages itself, how strong it is, nor the

complexity of the nest it builds but by how well it matches the intended goal state.

The data used by this algorithm is a simple 24 bit hex color. The decision was made for two reasons. The first reason was the ease of manipulation. A string of 24 bits proved easy to manipulate and easy to evaluate.

The second reason for choosing a 24 bit string was determined by evaluation. A String of 24 bits is easy to evaluate using a variety of methods. Certainly the hamming distance used by Castro would be effective here. In the end though I settled on using the Levenshtein distance. This will be covered in greater depth in section << 3. >>

### 3.1 Selection

As previously discussed Roulette Wheel Selection is often used in a Genetic Algorithms. Roulette Wheel selection is a weighted random selection algorithm that uses the fitness of a chromosome to determine the weight applied to said chromosome.

Whereas RW selection more closely matches what occurs in nature, it is never guaranteed that the fittest will be one of the chosen mating pairs. In genetic breeding varied genes are desired as they increase diversity and help prevent disease. In the realm of GAs diversity is to be avoided. The selection method chosen for this algorithm only uses the top two most fit elements. Where as the method chosen here does not match what occurs in nature it does match the genetic manipulation that has led to larger tomatoes, persian cats, and miniature horses.

Once the two most fit specimens are determined by means of the Levenshtein distance, which will be discussed further in section 3.3, they are then sent to the breeder method where the next generation is created.

### 3.2 Breeding

Since a GA is a hill climbing algorithm it is susceptible to local maxima or minima. The previous algorithm used the RW process as a means of mitigating local minima and maxima. Even though it is probabilistically unlikely, it is still possible that using the RW method could introduce aberrations that not only slow the refinement process but disrupt it entirely. The algorithm used in this instance forgoes the use of Roulette Wheel selection in favor of picking the two most fit specimens. This could of course lead to a situation where the most fit does not match the goal state but does appear to be more fit than the rest. In order to prevent such an occurrence the algorithm presented here relies

heavily on mutation and crossover to inject randomness and avoid local minima or maxima.

### 3.2.1 Crossing over

The algorithm from de Castro uses a single crossover point. This method works well when one wishes to retain a resemblance to the progenitor. The algorithm created for this paper utilizes four crossing over points per string.

At this point it is important to note that the entire algorithm was tailored to find the goal state in question. Where as this limits the general application of this algorithm, the general application of a GA was never in question. Indeed the ultimate goal here was to see if a GA could be used to quickly find a specific string. Something I think it does quite well.

As mentioned this algorithm was tailored to solve this particular problem as such the crossing over points were also tailored to fit the problem. Essentially instead of choosing a single crossing over point at random, four crossing over points were predetermined to maximize the similarities between the offspring and the goal state. Table 2 illustrates the crossing over points.

Table 2: The following table shows the result of the crossover technique implemented. The letter indicates which string and the number indicates the array index.

x0 y3 y6 x9 x12 y15 x18 y21
-----------------------------

In essence the idea here was to minimize the change in the first six sections and maximize the change in the remaining string. To simplify matters each substring is three characters in length and each string is always cut in the same way. Then half of each string is recombined as shown in table 2. This process is repeated n times for the number of children desired. It is important to note that each time this process is called the order of the parent strings is determined at random. This can be seen in the included code: breeder.php. Changing the order of the parents is important as it results in a different combination of characters after crossover. This introduces a wider variety of offspring as well as ensuring more random results.

Regardless of the amount of crossing over unless some aberration is introduced the only data attained will be a restructuring of the existing data. Even with crossing over the sets are still limited to the data that is presented. This is a serious flaw for if the initial random data is not worthwhile then a poor solution will be reached. This is why point mutation is also used.

### 3.2.2 Point Mutation

In addition to crossover the other method by which randomness is typically introduced is mutation. The standard practice in a GA is to select with some small probability a substring independent of the rest of the string and replace it with a modified bit string. In this case since we are dealing solely with bit strings a 1 becomes a 0 and vice versa. This process is designed to mimic point mutation found in nature.

Our algorithm relies on mutation as the primary source of randomization. In a typical GA, or in nature, a breeding pool is comprised of multiple parents and mutation is rare. In this modified algorithm the breeding pool is limited and mutation is very common. Indeed at least one mutation is guaranteed per iteration. The algorithm is randomized enough that there is no promise of which offspring will be mutated nor how many will be mutated.

One of the most interesting discoveries that arose from this algorithm was the importance of point mutation. This will be discussed further in the conclusion.

## 3.3 Evaluation

As previously mentioned the goal state is, entirely dependent upon the type of problem being solved. In our case it is easy to measure the difference between one 24 bit string and the next.

Initially the idea was to XOR a test string, s, with the goal state, g. The resulting string would return a value that would determine its fitness. Essentially the closer s is to g the smaller the number and thus the more fit the result. This is very similar to the hamming distance used by de Castro. The issue with the hamming distance is the time it takes to compute and the fact that additional computations are necessary to determine the difference between the produced chromosome and the desired state.

Another method of determining the fitness of the algorithm is by calculating the Levenshtein distance of the two strings. A Levenshtein distance between two strings is a measure of the number of changes needed to convert string s to string g. (Sluzburger, 2007)

In essence the Levenshtien distance is a measure of how many insertions, deletions, or substitutions are needed to convert string s to string g. See table 3 (Sluzburger, 2007)

It is clear to see that the levenshtein distance is ideally suited to evaluating the fitness of our string. It has the added benefit of being a delivered function in PHP the language used to implement the algorithm.

Table 3: You can see that there are 3 changes needed to go from string s to sting g.

String s	1100
String g	1001
distance	3

## 4 ANALYSIS

The single most interesting piece of information gleaned from this experiment was the correlation between mutations and the number of generations needed to reach a goal state.

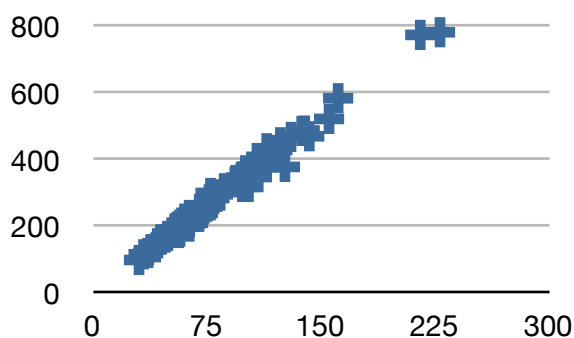


Figure 1: Mutations, shown on the Y axis, versus generations, shown on the X axis.

As figure 1 shows the number of total mutations versus the total number of generations increase in a linear fashion. This is to be expected as there is at least one mutation per generation. The interesting trend arises when the numbers are analyzed further.

When mutations are turned off completely no solution was found even when the maximum allowed number of iterations was increased from 500 to 10000. This led me to question what would happen if the number of mutations was maximized. Certainly using the maximum number of mutations would lower the number of generations needed to find a solution.

When the number of mutations was maximized again there was no solution found even when the number of iterations was increased to 10000. This makes sense because as the probability of mutation per string approaches 1 the likely hood of a string matching the goal state being mutated also approaches 1. Given more time and resources it would have been interesting to research how many iterations would be needed, if it was even possible, for a goal state to be reached with only crossover, only mutation, or with a better tailored crossover scheme.

## 5 CONCLUSIONS

Randomness is of great importance when it comes to reaching an overall goal. This is evident in nature as well since without mutation no significant change can occur. Genetic Algorithms, like all stochastic hill climbing algorithms, use randomness to avoid local maxima and minima.

In the end it is clear that a genetic algorithm is excellent at pattern matching and as such is well suited to such tasks as protein identification, cryptanalysis, and NP-C problem solving.

The drawbacks of GAs are, of course, the time it takes to solve a problem and poorly defined goal states.

Better hardware ensures that GAs can be run for increasingly difficult problems while still reaching a conclusion in reasonable time. Even still as the problems grow more complex the time to reach a solution will also grow, this is the central difficulty of NP problems. Better hardware can address this issue however any large NP problem will still take significant amounts of time to solve.

A proper fitting algorithm must be used to determine if an acceptable goal has been reached. The amount of time that the fitness algorithm takes will definitely impact the over all performance of the GA.

Finally specifically tailored breeding criteria can also significantly decrease the amount of time it takes to find a solution. This criteria can be tailored through several means. Both through experimentation and analysis of the data to be discovered.

## 6 QUO VADIS

I regret not having more time as I would have greatly enjoyed coding the exact algorithm from Castor and modifying the mutation rates to see how that affected the outcome. I would also like to see how my algorithm would have changed with the introduction of a true RW selection process.

It would also be interesting to see how a tailored algorithm handles scaling. If an algorithm could find a specific 24 bit string in as little as 30 generations, how long to find a 48 bit string? 128? 1024? also what specific modifications would be needed to tailor the breeding program to be more efficient.

Is it possible to create a GA that will itself be dynamic and adjust the breeding criteria to suit what it is searching for. In essence a GA that analysis the results of breeding and changes crossover and mutations to better match its fitness requirements.

Sadly most of the other questions raised by this project delve more into biology than computer science. The other questions are, of course, how is natural breeding affected by mutation? If mutation rates artificially increased how long will the organism maintain a familiar shape? With enough induced mutation would it be possible to cause a fish to grow legs, a snake to grow wings, or a human to develop increased natural healing or strength? Alas these questions are for another major.

## ACKNOWLEDGEMENTS

I must thank Dr. Greg Wood from CSU Channel Islands for introducing me early in my academic career to genetic algorithms. It was my initial work with his peptide simulator that got me interested in GAs to begin with,

Also I must thank Dr. AJ Bieszczad, also from CSUCI, for giving me the opportunity to experiment with GAs.

Also I have done a great deal of reading on this subject and have done my best to give credit where it is due. I fear that in my reading a particular phrase or idea may have stuck without realising where it originated. Regardless I have done my best to cite all sources that where not original.

Finally I must thank Jennifer Bonsangue who supplied much dinner and encouragement throughout the coding and the paper writing.

## REFERENCES

- de Castro, Leandro Nunes, 2006. *Fundamentals of Natural Computing*, Chapman & Hall/CRC. Boca Raton, FL.
- Gary, P., Hart, W., Painton, L., Philips, C., Trahan, M., Wagmer J., 1997. *A Survey of Global Optimization Methods*, Sandia National Labs. Albuquerque, NM. <http://www.cs.sandia.gov/opt/survey/>
- Nelson , Philip, 2004. *Biological Physics*, W.H. Freeman and Company. New York.
- Sluzberger, C., 2007. [www.levenshtein.net](http://www.levenshtein.net)
- MeSH Browser, 2009, *National Library of Medicine - Medical Subject Headings: Tree Number G05.365.590.675* U.S. National Library of Medicine. Bethesda, MD. [http://www.nlm.nih.gov/mesh/2009/mesh\\_browser/MBrowser.html](http://www.nlm.nih.gov/mesh/2009/mesh_browser/MBrowser.html)

## APPENDIX

The following section contains the code used to implement the algorithm.

### printer.php

```
<?php
function printer($gia, $gen, $mGen, $mTot){
//takes in an array and prints to an HTML file.
    $file = fopen("run.txt","a");

    $header = "<h2> GENERATION $gen
Mutation this Gen $mGen Total Mutations $mTot </h2><br>\n";

    fwrite ($file, $header);

    foreach ($gia as $printK => $printV) {

        $color =
        dechex(bindec($printV));

        $printV "\n" ; //file
        $foo = "<font color=\"\$color
\">Generation = $gen Color = $color
key = $printK value = $printV </font><br>\n" ; //file

        fwrite ($file, $foo);

    }//end of foreach

    $foo = "$gen , $mGen , $mTot \n";

    fwrite ($file, $foo); // used to
create a csv file

    fclose($file);
} //end of function
?>
```

### Breeder.php

```
<?php
function breeder($str1, $str2){
// takes in two strings and 'breeds'
them to produce one offspring

    $i = rand(0,1);

    if ($i == 1) {
        $x = $str1;
        $y = $str2;
    }else {
```

```

        $x = $str2;
        $y = $str1;
    }
    $x1 = substr($x,9,3);
    $x2 = substr($x,0,3);
    $x3 = substr($x,12,3);
    $x4 = substr($x,18,3);

    $y1 = substr($y,3,3);
    $y2 = substr($y,21,3);
    $y3 = substr($y,6,3);
    $y4 = substr($y,15,3);

    $foo = $x2 . $y1 . $y3 . $x1 .
    $x3 . $y4 . $x4 . $y2;
    return $foo;
}
?>

```

### functions.php

```

<?PHP

include('creator.php');
include ('breeder.php');
include ('printer.php');
include ('mutator.php');
include ('fitter.php');

?>

```

### creator.php

```

<?php
function creator($length){
//creates a $length length string of
binary
    $foo = (string) rand(0,1);
    for ($i = 1; $i <= $length -1; $i+
+) {
        $foo = $foo .
(string)rand(0,1);
    }
//    echo "in creator $foo \n";
    return $foo;
}
?>

```

### driver.php

```

<?php

// include functions
include ('functions.php');
// Define parameters

```

```

$continue = true;
$g = 0; //generation counter
$a = 15; //generation counter

$maxRuns = 500;

$initial = 8;

$goal = "1111" . "1111" . "0000" .
"0000" . "0000" . "0000";

$mMin = 1; // min number of mutants
$mMax = 6; // max number of mutants
$mTot = 0; // total number of
mutation
$mGen = 0;

$i=0;

$testFile = "testFile.txt";

//$fh = fopen($testFile, 'w') or
die("can't open file");

//step 00: generate entities
while ($i <= $initial ) {
    $key = (string)$i;

    $key = "sub" . $key;

    $subject[$key] = creator(24);
    $fitness[$key] = 99 ;

    $i++;
} //end of while

// use this section to print
headers and intial dataset
//=====
printer($subject,"start",$mgen,
$mTot);

for ($i = 1; $i <= $maxRuns; $i++)
{
    //step 01: select the top 2
fittest ones
ation = $a\n";
    foreach ($subject as $key =>
$value) {
        $fitness[$key] =
fitter($value, $goal);
        if ($fitness[$key] == 0) {
            $continue = 0;
        } //end if
    } //end for each

    $b1 =
current(array_keys($fitness));

```

```

        $b2 =
next(array_keys($fitness));
        $mCount = rand($mMin,
$mMax);
        $mTot = $mTot +
$mCount;
        $mGen = $mCount;
//step 02: breed more
//step 03: mutate
//[debug]echo "b1 = $b1 \n";
//[debug]echo "b2 = $b2 \n";

        $b1 = $subject[$b1];
        $b2 = $subject[$b2];

        foreach ($subject as $key =>
$value) {
            $subject[$key] = breeder($b1,
$b2);

            if ($mCount >= 0){
                $subject[$key] =
mutator($value, $mCount);
                $mCount--;
            } //end if

        } // end mutate for each

//step 04: print
printer($subject, $g, $mGen,
$mTot);

        $mGen = 0;
        $g++; //looping stuff
        if ($continue == 0){
            $i = 5000;
        }

    } // end main loop

//step 06: end

?>

```

### fitter.php

```

<?php

function fitter($str1, $goal){
//compares one string to the goal state
and returns the fitness.
//echo "in fitter \n";
    $lev = levenshtein($str1, $goal);
    return $lev;
}

?>

```

### mutator.php

```

<?php

function mutator($str, $int){
//takes in a string and mutates it $int
times.
    for ($i = 0; $i <= $int; $i++){

        $pos = rand(0, strlen($str));

        if (substr($str, $pos, 1)
=='0' ){
            $str =
substr_replace($str, '1', $pos, 1);
        }else{
            $str =
substr_replace($str, '0', $pos, 1);
        } //end of if
    } //end of for

    if (strlen($str) > 24){
        $str = substr($str, 0, 24);
    }
    $str = str_pad($str,
24, "0");
    return $str;
}

?>

```