How genetic algorithms really work
I.Mutation and Hillclimbing

Heinz Mühlenbein
GMD Schloss Birlinghoven
D-5205 Sankt Augustin1
e-mail: muehlenbein@gmd.de

Abstract

In this paper mutation and hillclimbing are analyzed with the help of representative binary functions and a simple asexual evolutionary algorithm. An optimal mutation rate is computed and a good agreement to numerical results is shown. In most of the cases the optimal mutation rate is proportional to the length of the chromosome. For deceptive functions an evolutionary algorithm with a good hillclimbing strategy and a reasonable mutation rate performs best. The paper is a first step towards a statistical analysis of genetic algorithms. It shows that the power of mutation has been underestimated in traditional genetic algorithms.

1. Introduction

The parallel genetic algorithm PGA introduced in 1987 [13], [10], [14] uses two major enhancements compared to the genetic algorithm [9]. First, selection and mating is constrained by a population structure. Second, some or all individuals of the population may improve their fitness by hillclimbing. The importance of a population structure for genetic algorithms has been demonstrated in [12],[6] [11]. In this paper the effect of hillclimbing is investigated. Closely connected to hillclimbing is mutation.

The method of investigation differs from previous approaches to understand how genetic operators work. The investigation proceeds bottom-up. First mutation and hillclimbing is isolated analyzed within the framework of a very simple evolutionary algorithm. This will be done both empirically and analytically for a test suite of binary functions. Then the genetic operators will be combined in more complex genetic algorithms.

Previous studies on how the genetic algorithm works have proceeded along two different lines. In the first line of approach, the theoretical one, most arguments are based on the *schema theorem* [4]. I have already mentioned in [12] that this theorem cannot be used to explain the search strategy of the genetic algorithm. The second line of research was experimental and top-down. Here a full-blown genetic algorithm was run and the parameters of the GA were optimized for a suite of test functions. An example is the work of Grefenstette [7]. His research was extended by Schaffer et al [16]. Empirically Schaffer found that the following formula gave an almost optimal parameter setting for their test suite

$$\ln N + 0.93 \ln m + 0.45 \ln n = 0.56 \qquad (1)$$

where

N := size of the population
m := mutation rate
n := length of the chromosome

This equation can be approximated by

$$N * m * \sqrt{n} = 1.7 \tag{2}$$

The formula indicates that the mutation rate should decrease with the size of the population. This paper will show that the formula is incorrect in general. A theoretical explanation of the above formula was tried by Hesser et al. [8]. The proof, however, was based on heuristic arguments in favour of the formula to be proven.

Fogarty [3] investigated variable mutation rates. He found that the mutation rate should decrease with the number of generations. This conclusion will be confirmed in this paper. But the main emphasis of the analysis is a fixed mutation rate.

The bottom-up approach isolates the effects of the different parameters more clearly. The goal of this research is to end up with a robust PGA which needs only a few parameters to be input by the user. In fact, the current PGA uses only three parameters - the size of the population, the mutation rate and a criterion determining when to do hillclimbing.

This paper will derive first guidelines on how to set the mutation rate. The guidelines cannot be specified by a simple formula like (2). The interdependence of the parameters is much more complex. Furthermore, they strongly depend on the fitness function.

The outline of the paper is as follows. In section two a synthetic test suite is introduced. Then the basic evolutionary algorithm $(1+1, m, hc)$ is defined. This algorithm models evolution with a population of size 1. In the next two sections the algorithm is analyzed. Then the effect of hillclimbing is investigated.

2. The synthetic test suite

Our test suite consists of three representative binary functions

- ONEMAX function
- (k,l)-deceptive function
- EQUAL function

ONEMAX just gives the number of 1's in the given string. A $(k,l) - deceptive$ function consists of l subfunctions, each of order k. The subfunctions have two local maxima. The global maximum is located at $(111\ldots1)$. It is isolated. The search is attracted by the local maximum located at $(000\ldots0)$. The idea of using deceptive functions for performance analysis of genetic algorithms has been emphasized by Goldberg [5]. The following deceptive function of order $k = 3$ has been used by many researchers.

bit	value	bit	value
111	30	100	14
101	0	010	22
110	0	001	26
011	0	000	28

The fitness of a $(k,l)-deceptive$ function is defined as the sum of the fitness of the l subfunctions. General order k deceptive functions can be found in [18].

EQUAL is defined as follows

$$EQUAL(b) = n^2 - (\#1 - \#0)^2 \tag{3}$$

EQUAL has many global maxima, which are given by an equal number of 1's and 0's (for n even). EQUAL is used in population genetics for the analysis of quantitative (or metric) traits [2].

3. The $(1+1,m,hc)$-algorithm

In the bottom-up approach mutation and local hillclimbing will be analyzed within the framework of a very simple evolutionary algorithm. Variants of this algorithm were used by many researchers (e.g.Bremermann [1], Rechenberg [15]).

I called the algorithm *Iterated Hillclimbing* in an earlier paper [12]. But the name $(1+1,m,hc)$-algorithm seems to be more appropriate. First, the interpretation of the algorithm in evolutionary terms is straightforward. Second, the name emphasizes the importance of the mutation rate m. Third, the algorithm has been analyzed in the problem domain of continuous functions by Bremermann [1] Rechenberg [15] and Schwefel [17]. $(1+1)$ denotes that the algorithm uses one parent and one offspring. The better of them will be parent of the next generation.

$(1+1,m,hc)$-algorithm

STEP1: Generate a random initial string s

STEP2: If $hc \neq \emptyset$, do the hillclimbing strategy hc, giving a modified s'.
If $f(s') > f(s)$ then $s := s'$

STEP3: If not TERMINATED, change each bit of s with probability m, giving s'
go to STEP2

This simple algorithm performs surprisingly well. Therefore it should be used as a benchmark for any other new search method.

4. Performance analysis of the $(1+1,m)$-algorithm

This section is an extension of the work presented in [12]. First the $(1+1,m)$-algorithm will be analyzed for the ONEMAX function. Despite the simplicity of the algorithm and

of the ONEMAX function, its statistical analysis is of surprising complexity. The analysis starts with the following question:

Given an initial string with i bits wrong. What is the expected number of trials T(k,n,m) required to reach the optimum?

This problem can be investigated with the theory of Markov chains. The Markkov process can be described by $(n+1)$ states. Each state is defined by the number of 1's in the string. The transition probabilities are given as follows

$$p(M_j \rightarrow M_l) = 0 \;\; if \;\; l < j$$

$$p(M_j \rightarrow M_l) = \sum_{r=0}^{j} K m^{r+s} * (1-m)^{n-(r+s)} \;\; s = l - j + r \; ; \; r \leq n - l \; ; \; l > j$$

$$p(M_j \rightarrow M_j) = 1 - \sum_{l>j} p(M_j \rightarrow M_l)$$

K is a product of binomial coefficients. The number of trials T can be obtained by inverting the matrix of the above transition probabilities. The exact computation is outside the scope of this paper. Instead, I will make an approximate analysis, trying to neglect higher order terms in m. This analysis has been inspired by the work of Bremermann [1]. The approximation will be justified by comparisons with numerical experiments.

Let us start with a simple example. The probability of a transition from state $j = 1$ to state $j = n$ is given by

$$p(M_1 \rightarrow M_n) = (1-m) * m^{n-1}$$

It is the product of mutating $n-1$ bits and not mutating the correct bit. This probability is the largest for

$$m = 1 - 1/n$$

If this mutation rate is applied, the mutated string will have two wrong bits on the average. It is very unlikely that the correct bit is not flipped, because the mutation rate is very high. Furthermore one of the n bits will not be flipped. But this bit is wrong with a high probability. So the algorithm is left with the problem to get the last two bits correct. It is easily seen that the algorithm has most difficulty getting the last bits correct. In this case the probability is high mutating the correct bits and low mutating the incorrect bits.

The theoretical analysis will be restricted to the above case, i.e. small i and $m \ll 1$. With these assumptions we may neglect higher order terms in m. Then the probability of an improvement is approximately given by

$$P(i,m) = (1-m)^{n-i}(1-(1-m)^i) \tag{4}$$

P is just the product of the probabilities of **not** changing one of the $(n-i)$ correct bits and changing **at least one** of the i wrong bits. From the above equation an optimal mutation rate can easily be computed. The solution is given by

$$m(i) = 1 - (1 - \frac{i}{n})^{1/i} \tag{5}$$

For large n and $i \ll n$, the optimal mutation rate is approximate $m = 1/n$.

If the $(1+1,m)-$algorithm knew in each step how many bits were wrong, then a mutation rate which decreases with i would be optimal. This observation explains Fogarty's result [3]. But the above assumption is of course unrealistic. Therefore the analysis will be based on a fixed mutation rate. The amount of computation with a fixed mutation rate and a variable mutation rate is not very different because the major problem is to get the last few bits correct. This result was also experimentally found by Fogarty [3].

The expected number of trials T to obtain an improvement can be computed from the probabilities. We assume that the number of trials needed to get an improvement if i bits are not correct is given by

$$< T(i,m) >= 1/P(i,m)$$

If the initial string is randomly generated, we expect that about n/2 of the bits will be wrong. Then the total number of expected trials to reach the optimum can be approximated by

$$< T(m) > = \sum_{i=1}^{n/2} < T(i,m) >$$

Theorem 1 *For $m = k/n$ with $n \gg 0, k \ll n$, the expected number of trials to reach the optimum is approximately given by*

$$< T(m) > \approx e^k \frac{n}{k} \ln n/2 \tag{6}$$

Proof:

$$P(1,k/n) = (1-k/n)^{n-1} * k/n \longrightarrow e^{-k} * k/n$$

$$P(i,k/n) \approx (1-k/n)^{n-i} * i * k/n \longrightarrow e^{-k} * k/n * i$$

$$T(k/n) \approx e^k * n/k \sum_{i=1}^{n/2} 1/i$$

Because of

$$\sum_{i=1}^{n/2} 1/i \leq \ln n/2 + 1$$

we obtain the result.

Remark: *The theorem is valid for any unimodal binary function. The optimal mutation rate is $m = 1/n$. A mutation rate k-times larger than the optimal mutation rate is worse than a mutation rate k-times smaller.*

The last observation follows from the fact that

$$\frac{T(k/n)}{T(1/kn)} = \frac{e^k}{e^{1/k}} * k^2 \tag{7}$$

Theorem 1 gives an estimate of the number of trials. This estimate is an upper bound for the amount of computation. For small mutation rates like $m = 1/n$ the string maybe not changed at all after testing all n loci. The following expressions give the distribution of the number of changed bits

$$\begin{aligned}
&\text{prob(0 bits changed,m=k/n)} = (1-m)^n \longrightarrow e^{-k} \\
&\text{prob(1 bit changed,m=k/n)} = nm(1-m)^{n-1} \longrightarrow k * e^{-k} \\
&\text{prob(2 bits changed,m=k/n)} \longrightarrow \tfrac{k^2}{2!} e^{-k}
\end{aligned}$$

An obvious improvement of the $(1+1,m)$-algorithm is to introduce a little bit of "intelligence" and to check if there is no bit changed after mutation. Then the total amount of computation in number of function evaluations is given by $T(k/n)(1-e^{-k})$. The last factor is the probability that at least one bit is changed by mutation.

Table 1 gives numerical data. The initial configuration was the 0 string, the mutation rate was $m = 1/n$.

Table 1
Numerical results for ONEMAX; q=0

n	T	sd	min	max	T est.
32	260	83	124	469	301
64	667	180	302	1238	723
128	1635	450	860	2816	1688
256	3632	912	2373	7262	3858

The agreement between formula (6) and the numerical results is surprisingly good. The numerical values clearly show the predicted asymptotic behavior of $n \ln n$.

The EQUAL function is very easy to optimize for the $(1+1,m)$-algorithm. Therefore we omit the statistical analysis and just give the numerical results.

Table 2
Numerical results for EQUAL; q=0

n	T	sd	min	max
32	29	8.2	17	48
64	57	10.9	35	82
128	113	14.0	82	150
256	222	18.5	186	258

The table shows that the number of trials to reach the optimum increases linearly with the problem size n. So EQUAL seems to be not a good candidate to be included in a test suite of optimization functions. But I will show in a forthcomimg book that the EQUAL function poses more difficulties to the plain genetic algorithm than the ONEMAX function.

5. Performance analysis of (k,l)-deceptive problems

The analysis proceeds as in the previous section. Let i be the number of subfunctions which are wrong. For simplicity assume that all wrong subfunctions are at the local maximum (00..0). This assumption is fulfilled with high probability after the early stages of the $(1+1,m)$-algorithm because the search is attracted by the local minimum.

The $(1+1,m)$-strategy will get nearer to the optimum, if at least one of the wrong subfunctions is changed to (11..1) and not a single one of the $k*(l-i)$ correct bits is flipped. For $m^2 \ll m$, the probability of success is given by

$$P(i,m) = (1-m)^{n-k*i} * (1-(1-m^k)^i)$$

The most difficult task for the algorithm is to get the very last subfunction correct. Therefore we consider P(1,m) only.

In this case the optimal mutation rate m can easily be computed. We obtain

$$m = k/n$$

The following formulas give the probabilities of success for the optimal mutation rate and the standard mutation rate $m = 1/n$.

$$P(1,k/n) \approx e^{-k} * l^{-k}$$

$$P(1,1/n) \approx e^{-1} * k^{-k} * l^{-k}$$

Theorem 2: *For $n \gg 0, k \ll n$, the expected number of trials T to reach the optimum for a (k,l)-deceptive function is given by*

$$< T(1/l) > \approx e^k * l^k * \ln l \tag{8}$$

$$< T(1/n) > \approx e * k^k * l^k * lnl$$

Proof: See Theorem 1

The following table gives some numerical results for the case $k = 3$ and $m = k/n$.

n	T	sd	min	max	T est.
30	47727	20362	17721	120417	46235
60	520422	192495	198713	1030237	481234

The table shows a high variance of $< T >$. Our crude estimate captures the overall tendency. The $(1+1,m)-$algorithm has no difficulties with deceptive functions. The amount of computation increases exponential in k, the order of the deceptive subfunction.

6. Analysis of hillclimbing

I have shown in a number of papers ([10],[12], that the efficiency of the PGA improves dramaticallly if hillclimbing is used. In this section two popular hillclimbing methods will be analyzed - *next ascent na* and *steepest ascent sa* hillclimbing. In next ascent hillclimbing, the bits are flipped in a predefined sequence. A flip is accepted, if the new string has a higher fitness value. In steepest ascent hillclimbing, all of the remaining bits of the sequence are flipped. Then the bit which gives the largest fitness improvement is actually flipped.

The following result is trivial.

Next ascent needs n trials to reach the optimum for ONEMAX, steepest ascent needs n(n-1)/2 trials.*

In this application, the more sophisticated hillclimbing methods performs worse than even the $(1+1,m)$-algorithm!

Let us now turn to the analysis of the (k,l)-deceptive function. For simplicity the analysis is restricted to the case k=3. The extension to arbitrary k is straightforward.

Let i be the number of subfunctions wrong. Let P(i,x,y) be the probability that a mutation generates a new configuration which will lead to an improvement after hillclimbing. x denotes the probability of jumping into the attractor region (111) from (000). y is the probability of jumping into the attractor region (000) from (111). Then the probability can be estimated like in the case of the ONEMAX-function.

$$P(i,x,y) = (1-y)^{l-i} * (1-(1-x)^i) \tag{9}$$

For next ascent one obtains

$$x = m^2 * (1-m) + m^3 \ = m^2$$

$$y = 2 * m - m^2 \ \approx 2 * m$$

These formulas can be derived as follows. Given (000) we have to generate (110) or (111) to get into the attractor region of (111). This gives x. On the other hand, six of the eight possible configurations lead to the attractor region (000) from (111).

In the same manner one gets for steepest ascent

$$x = 3m^2 - 2m^3 \approx 3m^2$$

$$y = 3m^2 - 2m^3 \approx 3m^2$$

The number of tested configurations to obtain an improvement is given by

$$C(i,m,hc) = 1/P(i,x,y) * \#hcop$$

Here $\#hcop$ denotes the number of hillclimbing operations per mutation. For next ascent we simply have $\#naop = n$. The analysis of steepest ascent is more difficult because $\#saop$ depends on the number of subfunctions $\#sc$ which have been changed by the mutation.

$$\#saop = n * (\#sc + 1)$$

$\#sc$ can be obtained from the probability distribution of how many subfunctions are changed by a mutation. This computation is omitted. For next ascent hillclimbing an optimal mutation rate can be computed as before by maximizing $P(1,x,y)$. The optimal mutation rate is given by

$$m = \frac{1}{l+1}$$

If we maximize $P(1,x,y)$ for steepest ascent we obtain

$$m = \sqrt{1/n}$$

This mutation rate minimizes the number of generations needed to reach the optimum, but not the number of tested configurations. The mutation rate is much larger than $m = 1/l$. Therefore it changes more subfunctions.

The following theorem just summarizes the results.

Theorem 3: *The probability Prob that the* $(1+1,m,hc)$- *algorithm gets the last subfunction of a* $(3,l)$*-deceptive function correct is given by*

$$Prob(1,1/l,na) \approx e^{-2} * l^{-2}$$

$$Prob(1,\sqrt{1/n},sa) \approx e^{-1} * l^{-1}$$

$$Prob(1,1/l,sa) \approx 3 * l^{-2}$$

$$Prob(1,1/n,sa) \approx 3^{-1} * l^{-2}$$

The number of tested configurations to reach the optimum is given by

$$C(1/l,na) \approx 3e^2 l^3 * \ln l \tag{10}$$

Proof: See Theorem1.

Remark: The probabilities are all of the same order in l, except steepest ascent with a mutation rate of $\sqrt{1/n}$. As mentioned above, this mutation rate need not to minimize the amount of computation. The $(1+1,1/l,na)$-algorithm performs only slightly better than the $(1+1,1/l)$-algorithm.

In table 3 numerical results for a range of mutation rates are given. The $(1+1,m,sa)$-algorithm performs much better than the other two algorithms. The amount of computation is fairly constant in a range of reasonable mutation rates. For $l = 20$ a mutation rate of $\sqrt{1/60}$ gives the lowest number of generations, but a mutation rate of $m = 1/10$ minimizes the amount of computation. It is an open question, whether the $(1+1,m,sa)-$algorithmus with an optimal mutation rate is asymptotically of less order in l than the other two algorithms.

The last two rows of table 3 show results for the case $q = 0$. Here the algorithm starts with the worst initial string, but the amount of computation is only slightly larger than starting with half of the bits correct. This demonstrates also numerically that the $(1+1,m,hc)$-algorithm spends most of its time getting the last subfunctions correct.

The results of this section can be summarized.

A good hillclimbing strategy reduces the amount of computation substantially.

Table 3
Numerical results for a (3,l)-deceptive function

n	hc	m	q	T	Con.
30	na	0.100	0.5	1300	37480
30	sa	0.100	0.5	98	11144
30	sa	0.180	0.5	47	7932
60	sa	0.300	0.5	634	518068
60	sa	0.200	0.5	162	103628
60	sa	0.160	0.5	136	74210
60	sa	0.129	0.5	128	60570
60	sa	0.100	0.5	148	57565
60	sa	0.050	0.5	434	104392
60	sa	0.017	0.5	3711	365995
60	na	0.040	0.5	7703	430340
60	sa	0.129	0.0	139	64969
60	sa	0.100	0.0	179	69244

7. Conclusion and outloook

In this paper search by mutation and hillclimbing was investigated in a bottom-up manner. Search by mutation appears to be fairly robust and can be analyzed with statistical methods. In a forthcoming paper search by crossingover will be investigated for the same set of test functions. It will be shown that both search methods are complementary. The probability that a mutation will give a better string decreases with the number of bits which are correct. In contrast, the probability that crossingover produces a better string increases with the number of correct bits.

The question now arises: *How to combine mutation and crossingover in a genetic algorithm so that there will be a synergy effect?* This question will be answered in a forthcoming book. The numerical experiments have been completed. The results will explain why many *nonstandard* genetic algorithms perform well and the *standard* genetic algorithm performs poorly.

Acknowledgement: The author thanks Andreas Reinholtz, who has done the numerical experiments.

1 H.J. Bremermann, J. Rogson, and S. Salaff. Global properties of evolution processes. In H.H. Pattee, editor, *Natural Automata and Useful Simulations*, pages 3–42, Washington, 1966. Spartan Books.

2 D.S. Falconer. *Introduction to Quantitative Genetics*. Longmann, London, 1981.

3 T.C. Fogarty. Varying the Probability of Mutation in the Genetic Algorithm. In J.D. Schaffer, editor, *Proc. of the Third International Conference on Genetic Algorithms*, pages 104–109, San Mateo, 1989. Morgan Kaufmann Publishers.

4 D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading MA, 1989.

5 D.E. Goldberg. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3:493–530, 1989.

6 M. Gorges-Schleuter. *Genetic algorithms and population structure - A massively parallel algorithm* . PhD thesis, University of Dortmund, 1990.
7 J.J. Grefenstette. Optimization of Control Parameters for Genetic Algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, 16:122–128, 1986.
8 J. Hesser and R. Männer. Towards an Optimal Mutation Probability for Genetic Algorithms. In H.P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature*, pages 23–32, New York, 1991. Springer LNCS 496.
9 J.H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.
10 H. Mühlenbein. Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 416–421, San Mateo, 1989. Morgan Kaufman.
11 H. Mühlenbein. Darwin's Continental Cycle and its Simulation by the Prisoner's Dilemma. *Complex Systems*, 5:459–478, 1991.
12 H. Mühlenbein. Evolution in Time and Space - the Parallel Genetic Algorithm. In G. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 316–338, San Mateo, 1991. Morgan Kaufman.
13 H. Mühlenbein, M. Gorges-Schleuter, and O. Krämer. Evolution algorithm in combinatorial optimization. *Parallel Computing*, 7:65–85, 1988.
14 H. Mühlenbein, M. Schomisch, and J. Born. The parallel genetic algorithm as function optimizer. *Parallel Computing*, 17:619–632, 1991.
15 I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann, Freiburg, 1973.
16 J.D. Schaffer, R.A. Caruana, L.J. Eshelman, and R. Das. A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization. In J.D. Schaffer, editor, *Proc. of the Third International Conference on Genetic Algorithms*, pages 51–61, San Mateo, 1989. Morgan Kaufmann Publishers.
17 H.P. Schwefel. *Numerical Optimization of Computer Models.* Wiley, Chichester, 1981.
18 L.D. Whitley. Fundamental Principles of Deception in Genetic Search. In G. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 221–241, San Mateo, 1991. Morgan Kaufman.