Abstract  Evolutionary algorithm is a probabilistic counterpart to a deterministic search method that impersonates the representation of natural biological evolution. Evolutionary algorithm (EA) operates on a population of potential solutions applying the principle of survival of the fittest to produce better and better estimates to a solution. At each generation, a new set of guesses is created by the process of selecting individuals according to their level of fitness in the problem domain and upbringing them together using operators copied from natural genetics. Evolutionary programming is similar to genetic programming, but the structure of the program is secure and its numerical parameters are allowed to change. The concept leads to the evolution of populations of individuals that are better suited to their environment than the individuals that they were created from, just as in natural adaptation. Mutability means for objects which can be changed and a mutant function mutates the object. In this paper we target a string and an array of random characters chosen from the set of upper-case alphabets together with the space, and of the same length as the target string. A fitness function computes the ‘closeness’ of its argument to the target string. A mutant function with a string and a mutation rate returns a copy of the string, with some characters mutated. Finally after several iteration it “mutates ” to target string successfully.

Keywords  Evolutionary Algorithm, Mutation and Mutant Function, Fitness Function

1. Introduction

An Evolutionary algorithm comes under Artificial Intelligence which is a subclass of evolutionary computation, a generic population-based meta-experimental optimization algorithm. The algorithm uses techniques inspired by biological evolution: reproduction, mutation, recombination, and selection. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function determines the environment within which the solutions live of the population then takes place after the repeated application of the above operators [1] & [8].

EA often perform well approximating solutions to all types of problems because they ideally do not make any assumption about the underlying fitness setting [4].

---

**Figure 1.** Structure of a single population evolutionary algorithm

These algorithms exhibit natural processes, such as
Better Implementation of Evolutionary Algorithm through Mutant Function

Evolutionary algorithms work on populations of individuals in place of single solutions. In this way the search is performed in a parallel manner.

Such a single population evolutionary algorithm is influential and performs well on a wide variety of problems. Nevertheless, better results can be achieved by introducing multiple subpopulations [2]. Every subpopulation evolves over a few generations quarantined (like the single population evolutionary algorithm) before one or more individuals are replaced between the subpopulation. The multi-population evolutionary algorithm prototypes the evolution of a species in a way more similar to nature than the single population evolutionary algorithm. Figure 2 shows the structure of such an extended multi-population evolutionary algorithm.

In most genuine applications of EAs, computational complexity is due to fitness function evaluation. Fitness estimate is one of the solutions to weaken this difficulty. Though, seemingly simple EA can solve often complex problems; so, there may be no direct link between algorithm complexity and problem complexity.

At the start of the computation a number of individuals (the population) are randomly initialized. The initial generation is produced.

If the optimization conditions are not met the creation of a new generation starts. Individuals are selected fitting to their fitness for the production of offspring. Parents are recombined to produce offspring. All offspring will be mutated with a certain probability. The fitness of the offspring is then evaluated/computed. The offspring are inserted into the population replacing the parents, producing a new generation. This iteration is performed until the optimization criteria are reached.

2. Analysis

The possibility of mutating a variable is inversely proportional to the number of variables (dimensions). The more dimensions one individual has, the smaller is the mutation probability. Different papers reported results for the optimal mutation rate. [5] writes, that a mutation rate of 1/n (n: number of variables of an individual) produced good results for a wide variety of test functions. It means that per mutation only one variable per individual is mutated. Hence, the mutation rate is independent of the size of the population.

Similar results are reported in [7] and [8] for a binary valued representation. For unimodal functions a mutation rate of 1/n was the best choice. An increase in the mutation rate at the beginning connected with a decrease in the mutation rate to 1/n at the end gave only an insignificant acceleration of the search.

The given recommendations from references [5], [7] and [8] for the mutation rate are only correct for separable functions. Yet, most real world functions are not fully separable. For these functions no recommendations for the mutation rate can be given. As long as nothing else is known, a mutation rate of 1/n is suggested as well.

The magnitude of the mutation steps is usually difficult to select. The best step-size depends on the problem studied and may even differ during the optimization process. It is known, that small mutation steps are often successful, mainly when the individual is already well adapted. However, greater changes (large mutation steps) can, when effective, produce good results much quicker. Therefore, a good mutation operator should often produce small step-sizes with a high probability and large step-sizes with a low probability [5 & 9].

This mutation procedure is able to generate most points in the hyper-cube defined by the variables of the individual and range of the mutation [3]. The range of mutation is given by the value of the parameter r and the area of the variables.
Most mutated individuals will be generated near the individual before mutation. Only some mutated individuals will be far away from the not mutated individual. That gives the probability of small step-sizes is greater than that of bigger steps. Figure 3 tries to give an influence of the mutation results of this mutation operator.

![Figure 3. Effect of mutation of real variables in two dimensions](image)

2.1. Binary Mutation

For binary values (0 and 1) individuals mutation processes compliments of variable values, because every variable has only two states. Thus, the size of the mutation step is always 1. For every individual the variable value to change is chosen (mostly uniform at random). Table 1 shows an example of a binary mutation for an individual with 11 variables, where variable 4 is mutated.

![Table 1. Individual before and after binary mutation](image)

III. Implementation

Process logic (for code listing 1):

- Let the target string be "PARAS NATH SINGH IS A MUTANT".
- An array of random characters chosen from the set of upper-case letters together with the space, “ABCDEFGHIJKLMNOPQRSTUVWXYZ “ (we will call it parent) and of the same length as the target string.
- A fitness function that computes the ‘closeness’ of its argument to the target string.
- A mutate function that given a string and a mutation rate returns a copy of the string, with some characters probably mutated.
- While the parent is not yet the target:
  - copy the parent C times, each time allowing some random probability that another character might be substituted using mutate.
  - Assess the fitness of the parent and all the copies to the target and make the most fit string the new parent, discarding the others.
  - repeat until the parent converges, (hopefully), to the target.

/*Implementation in C language */

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

cchar target[] = "PARAS NATH SINGH IS A MUTANT";
cchar str[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ ";
define CHOICE (sizeof(str) - 1)
define MUTATE 15
define COPIES 30

/* returns random integer from 0 to n - 1 */
int irand(int n)
{
    int r, rand_max = RAND_MAX - (RAND_MAX % n);
    while((r = rand()) >= rand_max);
    return r / (rand_max / n);
}

/* number of different chars between x and y */
int unfitness(char *x, char *y)
{
    int i, sum = 0;
    for (i = 0; x[i]; i++)
        sum += (x[i] != y[i]);
    return sum;
}

/* each char of y has 1/MUTATE chance of differing from x */
void mutate(char *x, char *y)
{
    int i;
    for (i = 0; x[i]; i++)
        y[i] = irand(MUTATE) ? x[i] : str[irand(CHOICE)];
y[i] = '\0';
}

int main()
{
    int i, best_i, unfit, best, iters = 0;
    char speci[COPIES][sizeof(target) / sizeof(char)];
    /* init rand string */
    for (i = 0; target[i]; i++)
        speci[0][i] = str[irand(CHOICE)];
```
spec[0][i] = 0;
    do {
        for (i = 1; i < COPIES; i++)
            mutate(spec[0], spec[i]);
        /* find best fitting string */ best = 0;
        for (best_i = i = 0; i < COPIES; i++) {
            unfit = unfitness(target, spec[i]);
            if (unfit < best || !i) {
                best = unfit;
                best_i = i;
            }
        }
        if (best_i) strcpy(spec[0], spec[best_i]);
        printf("iter %d, score %d: %s\n", iters++, best, spec[0]);
    } while (best);
    return 0;
} /*End of main */

4. Test & Results

Output of the above program comes with iteration number and score (for best variable) as following: iter 0, score 25: AAJAJFOFSZHMSCPBESWSUZFLZWWY iter 1, score 24: AAJAJFOASZHMSCPBESWSUZFLZWWY iter 2, score 23: AAJASF0AZHHMSCPBESWSUZNLZWWY iter 3, score 22: AAJASEAOAZHHMSCPBESWSUZNLZWWY iter 4, score 22: AAJASEAOAZHHMSCPBESWSUZNLZWWY iter 5, score 21: AAJASEAOAZZMSCPBESWSUZNLZWWY iter 6, score 20: AAJASEAOAZZMSCPBESWSUZNLZWWY iter 7, score 20: AAJASEAOAZZMSCPBESWSUZNLZWWY iter 8, score 20: AAJASEAOAZZMSCPBESWSUZNLZWWY iter 9, score 20: AAJASEAOAZZMSCPBESWSUZNLZWWY iter 10, score 19: AAJASEAOAZZMSCPBESWSUZNLZWWY iter 11, score 18: AAJASEAOAZZMSCPBESWSUZNLZWWY iter 12, score 18: AAJASEAOAZZMSCPBESWSUZNLZWWY iter 13, score 18: AAJASEAOAZZMSCPBESWSUZNLZWWY iter 14, score 18: AAJASEAOAZZMSCPBESWSUZNLZWWY iter 15, score 18: AAJASEAOAZZMSCPBESWSUZNLZWWY iter 16, score 17: AAJASEAOAZZMSCPBESWSUZNLZWWY iter 17, score 17: AAJASEAOAZZMSCPBESWSUZNLZWWY iter 18, score 17: AAJASEAOAZZMSCPBESWSUZNLZWWY iter 19, score 16: AAJASEHATFSISPCEISTWNUZWWY iter 20, score 15: AAJASEHATFSISPCEISTWNUZWWY iter 21, score 14: AAJASEHATFSISPCEISTWNUZWWY iter 22, score 14: AAJASEHATFSISPCEISTWNUZWWY iter 23, score 13: AAJASEHATFSISPCEISTWNUZWWY iter 24, score 12: AAJASEHATFSISPCEISTWNUZWWY iter 25, score 12: AAJASEHATFSISPCEISTWNUZWWY iter 26, score 11: AAJASEHATFSISPCEISTWNUZWWY iter 27, score 11: AAJASEHATFSISPCEISTWNUZWWY iter 28, score 11: AAJASEHATFSISPCEISTWNUZWWY iter 29, score 11: AAJASEHATFSISPCEISTWNUZWWY iter 30, score 11: AAJASEHATFSISPCEISTWNUZWWY iter 31, score 11: AAJASEHATFSISPCEISTWNUZWWY iter 32, score 11: AAJASEHATFSISPCEISTWNUZWWY iter 33, score 11: AAJASEHATFSISPCEISTWNUZWWY iter 34, score 11: AAJASEHATFSISPCEISTWNUZWWY iter 35, score 11: AAJASEHATFSISPCEISTWNUZWWY iter 36, score 10: AAJASEHATFSISPCEISTWNUZWWY iter 37, score 10: AAJASEHATFSISPCEISTWNUZWWY iter 38, score 10: AAJASEHATFSISPCEISTWNUZWWY

Better Implementation of Evolutionary Algorithm through Mutant Function

iter 39, score 10: FALASENATH SISPCEISTA NUTWHT iter 40, score 9: FALASENATH SISPCEISTA NUTWHT iter 41, score 9: FALASENATH SISPCEISTA NUTWHT iter 42, score 9: FALASENATH SISPCEISTA NUTWHT iter 43, score 9: FALASENATH SISPCEISTA NUTWHT iter 44, score 9: FALASENATH SISPCEISTA NUTWHT iter 45, score 9: FALASENATH SISPCEISTA NUTWHT iter 46, score 9: FALASENATH SISPCEISTA NUTWHT iter 47, score 9: FALASENATH SISPCEISTA NUTWHT iter 48, score 9: FALASENATH SISPCEISTA NUTWHT iter 49, score 9: FALASENATH SISPCEISTA NUTWHT iter 50, score 9: FALASENATH SISPCEISTA NUTWHT iter 51, score 9: FALASENATH SISPCEISTA NUTWHT iter 52, score 9: FALASENATH SISPCEISTA NUTWHT iter 53, score 9: FALASENATH SISPCEISTA NUTWHT iter 54, score 8: FALAS NATH SINJW ISTA NUTWHT iter 55, score 8: FALAS NATH SINJW ISTA NUTWHT iter 56, score 8: FALAS NATH SINJW ISTA NUTWHT iter 57, score 8: FALAS NATH SINJW ISTA NUTWHT iter 58, score 8: FALAS NATH SINJW ISTA NUTWHT iter 59, score 8: FALAS NATH SINJW ISTA NUTWHT iter 60, score 8: FALAS NATH SINJW ISTA NUTWHT iter 61, score 8: FALAS NATH SINJW ISTA NUTWHT iter 62, score 8: FALAS NATH SINJW ISTA NUTWHT iter 63, score 8: FALAS NATH SINJW ISTA NUTWHT iter 64, score 8: FALAS NATH SINJW ISTA NUTWHT iter 65, score 8: FALAS NATH SINJW ISTA NUTWHT iter 66, score 8: FALAS NATH SINJW ISTA NUTWHT iter 67, score 7: FALAS NATH SINJH ISTA HUTWHT iter 68, score 7: FALAS NATH SINJH ISTA HUTWHT iter 69, score 7: FALAS NATH SINJH ISTA HUTWHT iter 70, score 7: FALAS NATH SINJH ISTA HUTWHT iter 71, score 7: FALAS NATH SINJH ISTA HUTWHT iter 72, score 6: FALAS NATH SINJH ISTA HUTWHT iter 73, score 6: FALAS NATH SINJH ISTA HUTWHT iter 74, score 6: FALAS NATH SINJH ISTA HUTWHT iter 75, score 6: FALAS NATH SINJH ISTA HUTWHT iter 76, score 6: FALAS NATH SINJH ISTA HUTWHT iter 77, score 6: FALAS NATH SINJH ISTA HUTWHT iter 78, score 5: FALAS NATH SINJH ISTA HUTWHT iter 79, score 5: FALAS NATH SINJH ISTA HUTWHT iter 80, score 5: FALAS NATH SINJH ISTA HUTWHT iter 81, score 5: FALAS NATH SINJH ISTA HUTWHT iter 82, score 5: FALAS NATH SINJH ISTA HUTWHT iter 83, score 5: FALAS NATH SINJH ISTA HUTWHT iter 84, score 5: FALAS NATH SINJH ISTA HUTWHT iter 85, score 5: FALAS NATH SINJH ISTA HUTWHT iter 86, score 5: FALAS NATH SINJH ISTA HUTWHT iter 87, score 5: FALAS NATH SINJH ISTA HUTWHT iter 88, score 5: FALAS NATH SINJH ISTA HUTWHT iter 89, score 5: FALAS NATH SINJH ISTA HUTWHT iter 90, score 5: FALAS NATH SINJH ISTA HUTWHT iter 91, score 5: FALAS NATH SINJH ISTA HUTWHT iter 92, score 5: FALAS NATH SINJH ISTA HUTWHT iter 93, score 5: FALAS NATH SINJH ISTA HUTWHT iter 94, score 4: FALAS NATH SINJH ISTA HUTAHT iter 95, score 4: FALAS NATH SINJH ISTA HUTAHT iter 96, score 4: FALAS NATH SINJH ISTA HUTAHT iter 97, score 4: FALAS NATH SINJH ISTA HUTAHT iter 98, score 4: FALAS NATH SINJH ISTA HUTAHT iter 99, score 4: FALAS NATH SINJH ISTA HUTAHT iter 100, score 4: FALAS NATH SINJH ISTA HUTAHT iter 101, score 4: FALAS NATH SINJH ISTA HUTAHT iter 102, score 4: FALAS NATH SINJH ISTA HUTAHT iter 103, score 4: FALAS NATH SINJH ISTA HUTAHT iter 104, score 4: FALAS NATH SINJH ISTA HUTAHT iter 105, score 4: FALAS NATH SINJH ISTA HUTAHT
5. Conclusion

Mutation and recombination and the fitness setting trust on our choice of representation scheme for Evolutionary Algorithms. The illustration should consequently make mutation and recombination behave like the biological concepts they represent. For example, a small change in an individual's gene [2] (if we discuss Evolutionary Algorithm relating Genetic Algorithm) should make only a small to moderate change in its fitness features. Similarly, uniting parts of the genes of two individuals should yield an individual that shares some of its parents' features. However, the result need not be simply an average of the parents; there may be interaction between different parts of the genes. In cracking difficult glitches with EAs, finding a good representation scheme with good recombination and mutation operations can often be the toughest piece of the enigma. There is no mystic guidance for selecting the correct representation, and in addition to obeying to these guidelines, the choice must be possible to implement.

Mutating the alphabets to get the target string in above code, we searched the “Solution Space” (the set of all possible inputs) of this difficult problem for the best solutions successfully and not like existing random or “Brute-force Search”. Our program execution takes 182 iterations only to mutate the parent string contacting 26 characters from given set of alphabet with a blank character. Armed with our given principles, we implemented our own simplified EA using mutant function which works successfully in test cases. Lastly it's important to understand how the implementation details affect our Evolutionary Algorithm.

REFERENCES

[1] Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M., and Grunert da Fonseca, V.: Performance Assessment of Multiobjective Optimizers: An Analysis and Review, IEEE Transactions on Evolutionary Computation, 7(2), pp. 117-132, 2003.

[2] G.S. Hornby and J.B. Pollack. Creating high-level components with a generative representation for body-brain evolution. Artificial Life, 8(3):223–246, 2002.

[3] Jeff Clune, Benjamin Beckmann, Charles Ofria, and Robert Pennock. "Evolving Coordinated Quadruped Gaits with the
HyperNEAT Generative Encoding”. Proceedings of the IEEE Congress on Evolutionary Computing Special Section on Evolutionary Robotics, 2009. Trondheim, Norway.

[4] J. Clune, C. Ofria, and R. T. Pennock, “How a generative encoding fares as problem-regularity decreases,” in PPSN (G. Rudolph, T. Jansen, S. M. Lucas, C. Poloni, and N. Beume, eds.), vol. 5199 of Lecture Notes in Computer Science, pp. 358–367, Springer, 2008.

[5] Ferreira, C., 2001. Gene Expression Programming: A New Adaptive Algorithm for Solving Problems. Complex Systems, Vol. 13, issue 2: 87-129.

[6] [MSV93a] Mühlenbein, H. and Schlierkamp-Voosen, D.: Predictive Models for the Breeder Genetic Algorithm: I. Continuous Parameter Optimization. Evolutionary Computation, 1 (1), pp. 25-49, 1993.

[7] Bäck, T.: Optimal Mutation Rates in Genetic Search. In [ICGA5], pp. 2-8, 1993.

[8] Bäck, T.: Evolutionary Algorithms in Theory and Practice - Evolution Strategies, Evolutionary Programming, Genetic Algorithms. New York, Oxford: Oxford University Press, 1996.

[9] Mühlenbein, H.: The Breeder Genetic Algorithm - a provable optimal search algorithm and its application. Colloquium on Applications of Genetic Algorithms, IEE 94/067, London, 1994.