A Online Demo of Evolutionary Programming Using a Mixed Mutation Strategy for Solving Function Optimization

Hao Wu, Jun He and Xin Yao

Abstract—This paper presents an online demo of evolutionary programming using a mixed mutation strategy for solving function optimization problems. The strategy combines three different mutation operators: Gaussian, Cauchy and Lévy mutations. The algorithm has been implemented by a client-server web application, which is convenient for users to access through Internet. The web application architecture is divided into two parts: a client to deal with users’ input and a server to execute the computation. Experiments have shown that EP using mixed mutation strategies can produce high quality solutions on most of 14 benchmark functions.

I. INTRODUCTION

Several mutation operators, Gaussian, Cauchy and Lévy mutations [1], [2], [3] have been proposed in evolutionary programming (EP in short). According to the no free lunch theorem [4], none of mutation operators is efficient in solving all optimization problems. Each method is suitable only for a subset of problems. Experiments have also confirmed this point. For example, Gaussian mutation has a good performance for some unimodal functions and multimodal functions with only a few local optimal points; Cauchy mutation works well on multimodal functions with many local optimal points [2].

An approach to improve conventional EP using a single mutation operator is to apply different mutation operators simultaneously and integrate their advantages together. Such a strategy can be called a mixed mutation strategy, in the term of game theory [5]. There are different ways to design a mixed strategy. An early implementation is a linear combination of Gaussian and Cauchy distributions [6]. This combination can be viewed as a new mutation operator, whose probability distribution is a convolution of Gaussian and Cauchy’s probability distributions. Improved fast EP (IFEP) [2] takes another technique: each individual implements Cauchy and Gaussian mutations simultaneously and generates two individuals; the better one will be chosen in the next generation. Recently the idea of IFEP is applied in mixing Lévy mutations with various scaling parameters [3]. The advantage of these two mixed mutation strategies are their simplicity in implementation. A more advanced, however more complex, mixed strategy is inspired from game theory [7].

This paper still follows the idea of IFEP [2]. It combines three mutation operators: Gaussian, Cauchy and Lévy mutations. A parent generates three new individuals using the three mutations independently. And then a new offspring population is chosen from the offspring by a tournament selection.

A demo of the above EP using a mixed strategy is developed in this paper. Although many evolutionary computation software tools have appeared in the past decade, most of them are designed as independent software or application. This paper designs the demo in a different way, i.e., based on a client-server web application. This application is similar to web service application of evolutionary computation, e.g., “Web service for parameter estimation of non-linear systems with evolutionary algorithms” [8], which focuses on a specific application area.

The demo in this paper aims to provide a general EP solver for function optimization. The client-server application has the following advantages: first it is convenient for end users. Users only need to input their function through a web interface. Secondly, the server side can utilize the power of computation ability of server, e.g., a Linux cluster.

This paper is organized as follows: Section II presents an EP using a mixed mutation strategy; Section III describes the architecture of web application of EP and the design of the online demo; Section IV reports the experimental results on the benchmark problems; Section V is devoted to the conclusion.

II. EVOLUTIONARY PROGRAMMING USING A MIXED MUTATION STRATEGY

A. Evolutionary Programming

In the demo, an EP algorithm using a mixed mutation strategy is used to find a minimum \( \hat{x}_{\text{min}} \) of a continuous function \( f(\hat{x}) \), that is,

\[
f(\hat{x}_{\text{min}}) \leq f(\hat{x}), \quad \hat{x} \in D,
\]

where \( D = [a_i, b_i]^n \) is a hypercube in \( R^n \), \( n \) is the dimension. The details of the algorithm is described as follows:

1) Initialization: Generate an initial population consisting of \( \mu \) individuals at random. Each individual is represented a set of real vectors \( (\bar{x}_1, \bar{\sigma}_1) \), for \( i = 1, \cdots, \mu \),

\[
\bar{x}_i = (x_{i1}, x_{i2}, \cdots, x_{in}), \quad \bar{\sigma}_i = (\sigma_{1i}, \sigma_{2i}, \cdots, \sigma_{ni}).
\]

where \( \bar{x}_i \) is a point in the \( n \)-dimensional space \( R^n \) and \( \bar{\sigma}_i \) is a parameter to control the search step.

2) Mutation: For each parent \( (\bar{x}^{(t)}, \bar{\sigma}^{(t)}) \) (where \( t \) represents generation), create \( k \) offsprings \( (\hat{x}_l^{(t)}, \hat{\sigma}_l^{(t)}) \) (where
where \( L_j(\beta) \) is a Lévy random variable with scale parameter \( \beta \) is generated anew for each value of component \( j \). \( \beta = 1.4 \) is used in this paper.

III. ARCHITECTURE OF WEB APPLICATION AND ITS DESIGN

A. Architecture of Web Application and its Functions

The web application consists of two parts:

1) Client side: a web page interface and interface application. It accepts the input from users and sends parameters to the server for calling EP solver.

2) Server side: an expression parser and an EP solver. It is used to implement the task of computation and return result to the client.

The client application is loaded on the web so that users can utilize the application of EP through Internet. The client side provides interface program which is designed by Java Applet. Its main functions includes:

- Parameters initialization: Initialize the range of variables in the function; initialize parameter setting in EP: population size and generation size. And throw exceptions if input invalid characters.
- Mathematical function initialization: Users input the function for the EP to solve; throw exceptions if invalid function input.
- Predefined test functions: Users can choose 14 benchmark functions for testing the EP algorithm.
- View of results: The result is obtained from the server side. The client side displays the results. And users also have the option to display results step by step.

Server side provides a powerful computing ability. The server is designed by Java Servlet, a widely used server designing tool. The Server side provides the following functions:

Mathematic expression parser: It checks the validity of the function provided by users. At the present version, users is allowed to enter up to three specify variables \((x, y, z)\) in a function from expression input panel. It supports simply operations include e.g., \(\sin, \cos, \tan, \arcsin, \arccos, \arctan, \ln, \exp\), beside them, two constant symbols \(\pi\) and \(e\) are also provided. For a complex function with more than 3 variables, users had to write a separate Java function class.

EA solver: To apply an EP to solve the function optimization problem. EA part receives the parsed mathematic expression. The solver is a Java implementation of the EP using mixed mutation strategy, described in the previous section.

B. Design of Client Interface

a) Procedure of Client: This demo provides an easy-to-understand and easy-to-use for customers to input the function, constraints, and set parameters used in the evolutionary algorithms.

The tasks of client is divided into four subtasks, shown as follows:
C. Design of Server Side

c) Procedure of Server: The design of server in the demo aims at implementing two tasks:

1) to parse mathematical expression;
2) to apply EAs to solve the function optimization problem.

The procedure of server is shown as follows:

begin
  1 Receive the data sent by the client.
  2 Parse the expression.
  3 Run the EP.
  4 Generate results.
  5 Send results to the client.
end

The server is developed from a base HttpServlet and overrides the two methods in HttpServlet.

d) Decompose Expression: When the server receive the expression from the client, the expression parse will decompose the expression into individual elements, called “tokens”. For example:

\[ x \ast y -(z + 10) \]

Expression (8) includes \( x, \ast, y, -, (, z, +, 10 \text{ and }) \). Each token is an indecomposable unit. So the first step in the expression calculation processing is to get tokens from expression. In decompose expression-processing stage, first we have to define the delimiters, according to the basic mathematic general knowledge and the requirement of the demo \(+, -, \ast, /, (, )\) are defined as delimiters. Of cause, delimiters are also tokens. java.util.StringTokenizer class is used to break an expression string into tokens.

Tokens can be separated into three kinds: number, operator and constant. Here variables are regarded as constants too, because variables will be assigned before the expression calculation.

The calculation of operators follows the following order:

- Each operator has a specific priority.
  1) Priority 1: the mathematic symbols, e.g., \( \sin, \cos, \tan \);
  2) Priority 2: \(+, -\) (positive and negative);
  3) Priority 3: \(\ast, /\) (multiplication and division);
  4) Priority 4: \(+, -\) (addition and subtraction);
- Expressions are evaluated from left to right.
- The expression in the \((\) (Parentheses) has the highest priority, first in evaluation.

After an expression is decomposed into tokens, three main functions, sum(), term(), element(), are used to compare tokens and evaluate them. Function sum() processing \(+\), \(-\) (addition and subtraction) operation, Function term() processing \(\ast\) (multiplication), \(/\) (division) processing and Function element() processing \((\) (parentheses), \(+\) (positive), \(-\) (negative) operation. Another important function, called by function(), is used to compare and processing mathematic symbol and constant symbol operation.

Functions elements() and function() also can throw syntax error exception when detect invalid string or characters. For dealing with variables, in expression parses class, there are get() and set() two functions for each variable. set() is used to assign a value to specify variables, and get() is used to get the value from specify variables.

e) EP Structure and Designing: The main class procedure of EP solver is given in the followings:

begin
  Initialize individuals;
  Parse function expression;
  for \( i = 0 \) to GenerationSize step 1 do
    Mutation;
    Selection;
  od
end

Each step in EP is designed by a Java class, which are described in details as follows:
Individual class: It stores all attributes of each variable, including the value, range, and the step size $\sigma$. Each attribute has its corresponding functions for getting and setting its value.

Cell_ElEMENT class: Its attributes includes the result, the winning times and all attributes of variables which this element class has.

Deviations class: This class includes the random number generators, used in mutation class.

Expression Parser: It is used to calculate the result of each element according to the function and the value of variables which have been assigned in the generation and mutation class.

Generation class: Initialize the first generation elements as parent. Initialize the value of each variable in the function. Combine with the Expression Parser class to gain the value of functions. The population size will be also initialized here.

Mutation class: Create offspring from parents by mutation methods. The mutated variables and $\sigma$ will be assigned in offspring elements. Then combine with the Expression Parse class to gain the value of offspring element.

Selection class: Select a number of elements from parents and offspring as the next generation’s parents.

D. Design of Communication between Client and Server

The communication between server and client is implemented by the standard HTTP socket connection. Usually, the client will submit its request data to server through HTML form. The server can read the data in the form by the method HttpServletRequest.getParameter in java Servlet API.

In more details, first, the Applet opens a connection to the specified Servlet URL. Once this connection is made, the Applet then sends an output stream to Servlet or receives an input stream from the Servlet. This can be implemented by a GET or a POST method.

The POST method is used in the application. The POST method is powerful because both ascii and binary data can be transferred by this method. java.net.URLConnection class is used to POST data to the Servlet. Different from GET method, the Applet using POST method must inform the connection that it will send output stream and accept input stream. And the Applet has to set the content type in the HTTP request header. However, the Servlet will handle the type of data dependent on what the Applet sends.

IV. TEST OF DEMO PROGRAM

Three experiments have been carried out for this demo, which are:

1) To test whether the EP used in the demo can achieve the same high quality solution as those given in the reference, e.g. [3];
2) To test how the setting up of population size will influence the quality of solutions;
3) To test how the setting of lower bounds $\sigma_{\text{min}}$ will influence the quality of solutions.

A. Benchmark Functions

In order to test the demo program, a set of benchmark function set is chosen. These 14 test functions come from [3], which is given in Table I. $n$ is the dimension, which is given in the domain. The parameters of $\alpha_{ij}$, $e_i$ and function $u()$ can be found in [2].

These functions are classified into three classes: $f_1 - f_4$ are unimodal functions; $f_5 - f_9$ are multi-modal functions with many local optimums; and $f_{10} - f_{14}$ multi-modal functions with only a few local optimums.

B. Test of Solution Quality

Four different types of mixed mutation strategies are tested, which are Gaussian + Cauchy, Gaussian + Lévy, Cauchy + Lévy and Gaussian + Cauchy + Lévy. In the experiment, the population size is set to be 100, and the lower step bound $\sigma_{\text{min}} = 0.00001$. The prescribed maximum generation size is 1500. Table II gives a comparative experiment among different combinations of mutation operators and the best results from Table II given in [3]. The results is averaged over 10 runs of EP.

For Table II, it is observed that the EPs using mixed strategies can produce high quality results as the same as the best EP using a single mutation operators on functions $f_1 - f_{11}$. Their performance is less efficient than the best EP using a single mutation operator on functions $f_{12} - f_{14}$. The reason of why this phenomenon happens is that Gaussian mutation is the best for these functions, any combination of other mutations is worse than Gaussian mutation.

Among the four mixed mutation strategies, it is found that the combination of Gaussian, Cauchy and Lévy mutations is the best, better than any combination of two mutation operators. So it seems a diversity of mutation operators may lead to a better performance.

C. Test of different parameter setting

In this test, two experiments have been implemented: one is to test different population size; another is to test different lower bound $\sigma_{\text{min}}$.

In the first test, the EP using a combination of three mutation: Gaussian, Cauchy and Levy mutation. And the lower bound $\sigma_{\text{min}}$ is fixed to be 0.00001. The population size is taken to 100. Four functions $f_1$, $f_4$, $f_7$ and $f_{12}$ are chosen in the test. For each functions, EP runs 10 times and the final function value is the average over 10 runs.

Figures 1 to 4 shows the results for function $f_1$, $f_4$, $f_7$ and $f_{12}$. From these figures, it is clear that the final results is better as the population size increases. However, the curves are not monotonically decreasing in these figures. It is caused by the results being averaged only over 10 runs. If the EP runs more times, e.g., 100 times, it should be monotonically decreasing.

The second test still uses a combination of three mutation operators. And the population is fixed to be 100. As the same as in the first test, four functions $f_1$, $f_4$, $f_7$ and $f_{12}$ are chosen in the test. For each functions, EP runs 10 times and the final function value is the average over 10 runs. The results is give in Table III.
From Table III, it is seen for $f_1$ and $f_2$, the lower bound $\sigma_{\text{min}} = 0.0001$ is best; for $f_2$, $\sigma_{\text{min}} = 0.00001$ is best; for $f_{12}$, $\sigma_{\text{min}} = 0.01$ is best. This confirmed that for different functions, the optimal lower bound is different. So a mixture of using different lower bounds is needed to improve the EP demo.

V. CONCLUSION AND FUTURE WORK

This paper has presented an online demo of EP using a mixed mutation strategy for solving function optimization problem. The mixed mutation strategy aims at integrating different mutation operators into one algorithm. As shown in the experiments, this strategy can increase the diversity of mutation operators and enhance the search ability of the EP algorithm.

The online demo has been developed based on client-server web application and evaluated on a set of test function. The web application can provide a more convenient way for users to solve function optimization by using evolutionary technique, without complex programming.

Comparative experiments have demonstrated that the initial version of the EP online demo using a mixed mutation strategy can produce high quality solutions on 11 of 14 benchmark functions. But it still needed to improve on 3 test functions.

The future work of the demo program will include: to improve the EP with mixed lower bounds; to improve the mixed strategy using game theory; to solve function optimization problems with complex constraints; to connect the server side with a Linux cluster.

Acknowledgement: This work is partially supported by Engineering and Physical Research Council under Grant (EP/C520696/1).
function value

population size

function value

population size

function value

population size

function value

population size

function value

population size

function value

population size

function value

population size

function value

population size

function value

population size

lower bound $\sigma_{\text{min}}$ | 0.01 | 0.001 | 0.0001 | 0.00001 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>0.017686</td>
<td>3.8075e-4</td>
<td>2.290e-5</td>
<td>4.582e-5</td>
</tr>
<tr>
<td>$f_4$</td>
<td>-10770.26165</td>
<td>-10979.12889</td>
<td>-11080.7385</td>
<td>-11287.865</td>
</tr>
<tr>
<td>$f_7$</td>
<td>0.01307</td>
<td>0.0104</td>
<td>2.5946e-4</td>
<td>0.0188798</td>
</tr>
<tr>
<td>$f_{12}$</td>
<td>-7.24377</td>
<td>-6.951344</td>
<td>-6.765256</td>
<td>-7.144876</td>
</tr>
</tbody>
</table>

**TABLE III**

The relationship between the lower bound and the final result.

**REFERENCES**