Available online at www.sciencedirect.com
SCIENCE DIRECT
Computer Physics Communications 174 (2006) 555–559
Computer Physics Communications
www.elsevier.com/locate/cpc

GDF: A tool for function estimation through grammatical evolution ☆

Ioannis G. Tsoulos [a,*], Dimitris Gavrilis [b], Evangelos Dermatas [b]

[a] Department of Computer Science, University of Ioannina, P.O. Box 1186, Ioannina 45110, Greece
[b] Department of Electrical & Computer Engineering, University of Patras, Patras 26500, Greece

Received 19 July 2005; received in revised form 1 November 2005; accepted 5 November 2005

Available online 15 December 2005

Abstract

This article introduces a tool for data fitting that is based on genetic programming and especially on the grammatical evolution technique. The user needs to input a series of points and the accompanied dimensionality n and the tool will produce via the genetic programming paradigm a function $f: R^n \to R$ which is an approximate solution to the symbolic regression problem. The tool is entirely written in ANSI C++ and it can be installed in any UNIX system.

Program summary

Title of program: GDF
Catalogue identifier: ADXC
Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland
Program summary URL: http://cpc.cs.qub.ac.uk/summaries/ADXC
Computer for which the program is designed and others on which is has been tested: The tool is designed to be portable in all systems running the GNU C++ compiler
Installation: University of Ioannina and University of Patras, Greece
Programming language used: GNU-C++
Memory required to execute with typical data: 200 KB
No. of bits in a word: 32
No. of processors used: 1
Has the code been vectorized or parallelized?: No
No. of bytes in distributed program, including test data, etc.: 33 469
No. of lines in distributed program, including test data, etc.: 5704
Distribution format: tar.gz
Solution method: Functional forms are being created by genetic programming which are approximations for the symbolic regression problem.
© 2005 Elsevier B.V. All rights reserved.

PACS: 02.30.Mv; 02.60.-x; 02.60.Ed; 07.05.Mh

Keywords: Function approximation; Stochastic methods; Genetic programming; Grammatical evolution

1. Introduction

The problem of function estimation consists of finding a function that will best approximate a set of n-dimensional points given their output. Function estimation finds many applications in physics, chemistry, signal processing etc. and it can be formulated as follows: *Given M points and associated values (x_i, y_i), $i = 1, \ldots, M$, with $x_i \in R^n$ estimate a function*

☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author.
E-mail address: sheridan@cs.uoi.gr (I.G. Tsoulos).

0010-4655/$ – see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.cpc.2005.11.003

$f : R^n \to R$ that minimizes the least squares "Error"

$$E_T = \sum_{i=1}^{M} \big(f(x_i) - y_i\big)^2. \quad (1)$$

Through the years many methods have been proposed for this problem, such as spline based [2,3] or neural network based [4,5]. Although these techniques have been applied successfully to many data fitting problems, they produce functional forms which consist of applications of specific functions such as polynomials or sigmoidal functions. The proposed programming tool takes as input the points (x_i, y_i) and creates a functional form which is an approximate solution to the symbolic regression form minimizing the quantity in Eq. (1) through the procedure of Grammatical Evolution [1]. Grammatical Evolution is an evolutionary processes that can create programs in an arbitrary language. The production is performed using a mapping process governed by a grammar expressed in Backus Naur Form. Grammatical Evolution has been applied successfully to problems such as symbolic regression [6], discovery of trigonometric identities [7], robot control [8], caching algorithms [9], financial prediction [10], etc. The rest of this article is organized as follows: in Section 2 the contents of the distribution are presented, in Section 3 the installation steps for any UNIX programming environment are expressed in detail, in Section 4 the main parts of the distribution such as the underlying algorithm, the grammar specifications and the `gdf` program are thoroughly analyzed and finally in Section 5 some conclusions from the application of the tool are listed.

2. Distribution

The package is distributed in a tar.gz file named `GDF.tar.gz` and under UNIX systems the user must issue the following commands to extract the associated files:

1. gunzip `GDF.tar.gz`
2. tar xfv `GDF.tar`

These steps create a directory named `GDF` with the following contents:

1. **bin**: It is a directory which initially it contains the test file xsinxx.data. After the compilation two files will be added to this directory: the executable `gdf` and a text file named grammar.txt. The first file is the created programming tool and the second one is an auxiliary file that contains the grammar of the tool, expressed in BNF format.
2. **include**: The directory which contains the header files for all the classes of the package.
3. **src**: The directory with the source files of the package.
4. **Makefile**: It is the file that will be read by the make utility in order to built the tool. There is no need for the user to modify this file.

3. Installation

The following steps are required in order to build the tool:

1. Uncompress the tool as described in the previous section.
2. Issue the command `cd GDF`.
3. Type `make`.

After the compilation the binary file `gdf` will be placed in the bin subdirectory accompanied with the text file `grammar.txt`. The tool is entirely written in GNU C++ version 3.2.3, but it can be installed in systems with different ANSI C++ compiler. The only modification required is to replace the line

```
CC = c++
```

in the file `Makefile` under the `src` subdirectory with the following one

```
CC = mycpp
```

where `mycpp` is the name of the corresponding ANSI C++ compiler in the hosting operating system.

4. Program details

4.1. The underlying algorithm

The programming tool is based on the following stochastic algorithm:

Initialization step:

1. The program reads the data to be fitted from a text file.
2. The program reads the used grammar from a text file.
3. Every chromosome in the genetic population is initialized. The initialization is performed by a randomly selection of a number in the range $[0, 255]$ for every element of each chromosome.
4. The values for the parameters **selection rate** and **mutation rate** are selected. The **selection rate** denotes the fraction of the number of chromosomes that will go through unchanged to the next generation. That means that the probability for crossover is set to 1-**selection rate**. The value of **mutation rate** controls the average number of changes inside a chromosome. The values for the parameters selection rate and mutation rate are mutually independent and they must be in the range $[0, 1]$.
5. Set $k = 0$, where k is the amount of the generations.
6. Set the value for the parameter $\max K$, where $\max K$ is the maximum allowed number of generations.

Evolution step:

1. For every chromosome in the population, a function is created through the process of Grammatical Evolution.
2. The fitness of each member of the population is evaluated.
3. The chromosomes are sorted in descending order according to their fitness values.
4. The best (selection rate) × (population size) chromosomes are copied to the next generation. The rest (1-selection rate) × (population size) chromosomes are created using the

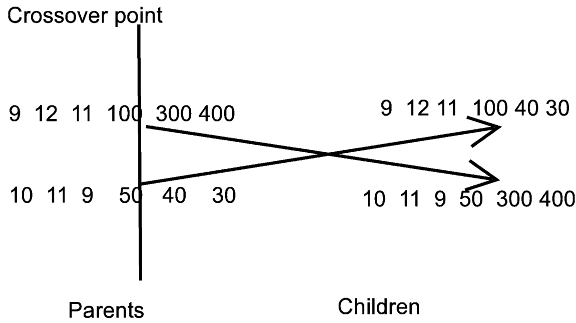

Fig. 1. One-point crossover.

```
<S> ::= <expr>
<expr> ::= ( <expr> )
         | <expr> <op> <expr>
         | <func> ( <expr> )
         | <terminal>
<op> ::= +
<op> ::= -
<op> ::= {*}
<func> ::= sin
<func> ::= cos
<func> ::= log10
```

Fig. 2. Grammar specification.

crossover procedure and are copied to the next generation. Every new chromosome is formed from two selected individuals (parents) of the current population with one-point crossover. In that procedure the chromosomes are cut at a randomly chosen point and their right-hand side subchromosomes are exchanged, as shown in Fig. 1. For every new chromosome the selection of every parent is performed through tournament selection, i.e.:

(a) A group of $N > 1$ randomly selected chromosomes is created.

(b) The chromosome with the best fitness value in the group is selected, the others are discarded.

5. The mutation procedure is applied to each member of the population with probability equal to **mutation rate**.
6. **Set** $k = k + 1$.
7. If $k > \max K$ or the best fitness value has fallen below a predefined threshold, then the **Evolution** step is terminated.

4.2. Grammar specification

The file that contains the grammar specification must be determined by the user with the -g option from the command line. The grammar must be specified in any simple text (ASCII) file with the format shown in Fig. 2.

The start symbol (<S>) is required and must be specified in the above form. The start symbol can give only one non-terminal symbol (e.g., <expr>). The available non-terminal symbols are <expr>, <func>, <op>. The available terminal symbols are +, -, *, /, (,). The numbers are represented as lists of digits (including ".") and can be specified by <terminal>. The subrules for the <terminal> symbol are fixed in the code and they cannot be changed by the user. These rules are shown in Fig. 3. The symbol d in the rule for <xxlist>

```
<terminal>::=<xxlist>
            |<digitlist>.<digitlist2>
<xxlist>::=x1 | x2 |... | x-d
<digitlist>::=<digit>
            |<digit><digitlist>
<digit>::= 0 | 1 | 2 |3 |4 |5 |6 |7 |8 |9
<digitlist2>::=<digit>
            |<digit><digit>
```

Fig. 3. The rules of the symbol <terminal>.

D
M

x_{11}	x_{12}	$\cdots$	x_{1D}	y_1
x_{21}	x_{22}	$\cdots$	x_{2D}	y_2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
x_{M1}	x_{M2}	$\cdots$	x_{MD}	y_M

Fig. 4. Data fitting format.

denotes the dimensionality of the objective function. The available functions are: sin, cos, log, exp, log10, tan, abs, sqrt, int, atan, acos, asin. If a non-terminal specification has more that one rules, those rules can be specified with a "|" instead of typing the entire left hand (e.g., in the second rule of <expr>, "<expr> ::=" is replaced by "|"). In this way, the user can easily alter the program parameters by specifying a different grammar. If, for example, it is known that log or log 10 cannot exist in the desired output, the user can remove them from the grammar specification.

4.3. The main program gdf

The created executable gdf takes the following series of parameters in the command line

1. **-h**: The program prints a help screen to the user with a description for each command line parameter and it terminates.
2. **-g** grammar_file: The parameter grammar files determines a file with a valid grammar for the tool. The user must have read access to the specified file. The default value for this parameter is grammar.txt, which is the default grammar and it is copied after the installation in the subdirectory `bin`.
3. **-p** problem_file: The parameter problem_file determines a file containing the points where the data fitting procedure will be applied. The user must have read access to the specified file and the contents of the file must conform to the format of Fig. 4. The integer number D in the file determines the dimensionality of the specific problem, the number M determines the amount of points in the file and each consecutive line defines a point where the data fitting procedure will be applied. This parameter is the only one required from the program.
4. **-t** test_file: The parameter test_file determines a file in the same format as the problem_file, where the produced function will be tested after the termination of the genetic algorithm. The user must have read access to the specified file

and the dimension in the file test_file must be the same with that of the problem_file.

5. **-c** count: The parameter count specifies the number of chromosomes in the genetic population. The default value for this parameter is 500. Small values for this parameter have as a consequence fast genetic operations but the optimal solution may not be discovered, while big values cause slow genetic operations but with additional certainty for the finding of the optimal solution.
6. **-l** length: The parameter length specifies the length of each chromosome in the genetic population. The default value for this parameter is 100. The standard GE approach uses variable-length chromosomes, but the tool GDF uses chromosomes with static length in order to prevent it from creating extremely large chromosomes. Small values for this parameter cause fast genetic operations and probably simple function forms. On the other hand, big values for this parameter yield slow genetic operations and probably complex function forms.
7. **-s** srate: The parameter srate specifies the value for the parameter **selection rate** of the genetic algorithm. The default value for this parameter is 0.1 (10%). This parameter controls the fraction of chromosomes that will go through unchanged to the next generation and as a consequence the number of the performed crossovers. High values for **selection rate** make the algorithm fast but without many crossovers, and as a consequence the algorithm can be trapped to suboptimal solutions. Low values slow down the algorithm but the performed crossovers can help to identify the optimal solution.
8. **-m** mrate: The parameter mrate determines the value for the parameter **mutation rate** of the genetic algorithm. The default value for this parameter is 0.05 (5%). The procedure of mutation, controlled by the parameter **mutation rate**, can be seen as an exploration procedure of the search space. Low values for this parameter make the algorithm to converge faster, although the discovered solution may be a suboptimal one. High values cause an exhaustive exploration of the search without certainty about the convergence of the algorithm.
9. **-n** generations: The integer parameter generations determines the maximum number of the generations allowed for the genetic algorithm. This parameter controls the speed and the efficiency of the method because low values can lead to a fast and probably premature convergence, while high values can have as a consequence better estimation of the objective function with the cost of additional time. The default value for this parameter is 2000.
10. **-r** seed: The parameter seed specifies the seed for the random number generator. This parameter is very crucial for the method, because the method is a stochastic procedure and as a consequence different random numbers can lead to different functions. The default value for this parameter is 1.

In each generation the program prints in the screen the following quantities:

1. The number of generations passed.
2. The best discovered function.
3. The fitness value of the best discovered function.

4.4. Test runs

The performance of the proposed tool was measured by using 5 different datasets: one for the continuous function $f(x) = x\sin(x^2)$ and 4 real life problems.

4.4.1. The function $f(x) = x\sin(x^2)$

The tool was tested on this function using a dataset with 100 random points from the function in the range $[-2, 2]$. The tool was issued with the following command:

```
./gdf -pxsinxx.data
```

where xsinxx.data is the file containing the points for the data fitting. The last 10 lines from the output of the above program shown in Fig. 5.

4.4.2. The Ailerons problem

This problem has 40 attributes and consists of 7150 points. This data set addresses a control problem, namely flying a F16 aircraft. The attributes describe the status of the aeroplane, while the goal is to predict the control action on the ailerons of the aircraft. The original owner of the database is Rui Camacho (rcamacho@garfield.fe.up.pt). The program `gdf` was trained with 200 points from the dataset and the resulting expression was tested over the rest 6950 points. Ten independent experiments were conducted with different random seeds each time and in all cases the absolute value of the fitness was below 3×10^{-6}. The best discovered function was

$$f(x) = \frac{x_9}{227.3} - \frac{\sin(x_{16}/\sqrt{x_{39}})}{\sqrt{1187 - x_8 - 756.92x_3}}$$

with fitness -1.65×10^{-6} and test error per point 1.07×10^{-7}. The resulting function depends only on 5 from the 40 attributes.

4.4.3. The Elevators problem

The original source of this problem is the experiments of Rui Camacho (rcamacho@garfield.fe.up.pt). The problem has 18 attributes and this data set is also obtained from the task of controlling a F16 aircraft, although the target variable and attributes are different from the ailerons domain. In this case the goal variable is related to an action taken on the elevators of the aircraft. From this dataset 200 points were used for training and 8452 for testing. The best discovered function was

$$f(x) = \frac{5.7}{866.7} - 9.8x_{13} - x_{18} + \frac{x_8}{3} - 9.91x_{10}\exp(x_4)$$

with fitness value -1.64×10^{-3} and mean test error 3.41×10^{-5}.

4.4.4. The Pyrimidines problem

The source of this dataset is the URL:

http://www.ncc.up.pt/~torgo/Regression/DataSets.html

```
generation=156 f(x)=sqrt(sqrt((log(2.72)))){*}sin(abs(exp(log(((x1)){*}x1)))){*}x1 fitness=-1.260575367e-06
generation=157 f(x)=sqrt(sqrt((log(2.72)))){*}sin(abs(exp(log(((x1)){*}x1)))){*}x1 fitness=-1.260575367e-06
generation=158 f(x)=sqrt(sqrt((log(2.72)))){*}sin(abs(exp(log(((x1)){*}x1)))){*}x1 fitness=-1.260575367e-06
generation=159 f(x)=sqrt(sqrt((log(2.72)))){*}sin(abs(exp(log(((x1)){*}x1)))){*}x1 fitness=-1.260575367e-06
generation=160 f(x)=sqrt(sqrt((log(2.72)))){*}sin(abs(exp(log(((x1)){*}x1)))){*}x1 fitness=-1.260575367e-06
generation=161 f(x)=sqrt(sqrt((log(2.72)))){*}sin(abs(exp(log(((x1)){*}x1)))){*}x1 fitness=-1.260575367e-06
generation=162 f(x)=sqrt(sin((log(4.72)))){*}sin(abs(exp(log(((x1)){*}x1)))){*}x1 fitness=-4.10576003e-07
generation=163 f(x)=sqrt(sin((log(4.72)))){*}sin(abs(exp(log(((x1)){*}x1)))){*}x1 fitness=-4.10576003e-07
generation=164 f(x)=sqrt(sin((log(4.72)))){*}sin(abs(exp(log(((x1)){*}x1)))){*}x1 fitness=-4.10576003e-07
generation=165 f(x)=sqrt(sin((log(4.82)))){*}sin(abs(exp(log(((x1)){*}x1)))){*}x1 fitness=-4.832536538e-11
```

Fig. 5.

and it is a problem of 27 attributes and 74 number of patterns. The task consists of Learning Quantitative Structure Activity Relationships (QSARs) and provided by [11]. From the above dataset 50 patterns were used for training and 24 for testing. The best discovered function was:

$$f(x) = \cos\big(\cos(\sqrt{x_{20}})\big)\cos\big(x_{22} - \log\big(\sin\big(\exp(x_6)\big)\big)\big) + \frac{\sin(x_8)}{1.3}\sqrt{x_{15}\sin\big(\sin(x_9)\big)}$$

with fitness value -1.33×10^{-1} and mean test error 7.25×10^{-3}.

4.4.5. *The Basketball problem*

The source of this dataset is from Smoothing Methods in Statistics available from the:

ftp://stat.cmu.edu/datasets

which is a problem of four attributes and it tries to identify the points scored per minute from the attributes "assists per minute", "player height", "time played" and "player age". From the 96 available patterns 60 were used for training and 36 for testing. The best discovered function was:

$$f(x) = \sin(8.59)\frac{\cos\big(\sqrt{\exp(\sin(x_1))}\big)}{\sqrt{\cos\big(\frac{x_3}{33.1-0.245\cos(x_3 x_2)}\big)}}$$

with fitness value -3.78×10^{-1} and mean test error 6.99×10^{-3}.

5. Conclusions

The introduced tool is a program aimed to fit a function in a series of points of an arbitrary dimension. As it can be easily verified from the conducted experiments that the tool was able to find the analytical solution for the case of $x\sin(x^2)$ and a functional form for the other cases where no analytical solution is provided in the usual literature. The algorithm tries to find the optimal function f, but it does not guarantee an optimal solution since it is an evolutionary approach. Also, the suggested approach can create very complex functions, even though the data set comes from a simple function. However, the user can apply the tool even in cases where the existence of an analytical solution is difficult to find. Also, the tool is provided with the ability of changing the underlying grammar according to the problem.

References

[1] M. O'Neill, C. Ryan, IEEE Trans. Evolutionary Comput. 5 (2001) 349–358.
[2] C. De Boor, A Practical Guide to Splines, Springer-Verlag, New York, 1978.
[3] D. Kincaid, W. Cheney, Numerical Analysis, Brooks/Cole Publishing Company, 1991.
[4] K. Hornik, M. Stinchcombe, H. White, Neural Networks 2 (1989) 359.
[5] G. Cybenko, Math. Control Signals Systems 2 (1989) 303–314.
[6] M. O'Neill, C. Ryan, Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language, Genetic Programming, vol. 4, Kluwer Academic Publishers, 2003.
[7] C. Ryan, M. O'Neill, J.J. Collins, in: Proc. of Mendel 1998: 4th Internat. Mendel Conf. Genetic Algorithms, Optimization Problems, Fuzzy Logic, Neural Networks, Rough Sets, Brno, Czech Republic, Technical University of Brno, Faculty of Mechanical Engineering, 1998, pp. 111–119.
[8] J.J. Collins, C. Ryan, in: Proc. of AROB 2000, 5th Internat. Symp. on Artificial Life and Robotics, 2000.
[9] M. O'Neill, C. Ryan, in: K. Miettinen, M.M. Mkel, P. Neittaanmki, J. Periaux (Eds.), Evolutionary Algorithms in Engineering and Computer Science, Jyvskyl, Finland, 1999, pp. 127–134.
[10] A. Brabazon, M. O'Neill, in: H.R. Arabnia (Ed.), Proc. Internat. Conf. on Artificial Intelligence, vol. II, CSREA Press, 2003, pp. 492–498.
[11] R.D. King, S. Muggleton, R. Lewis, M.J.E. Sternberg, Proc. Nat. Acad. Sci. USA 89 (23) (1992) 11322–11326.