Software Project Effort Estimation Using Genetic Programming

Y. Shan, R.I. McKay, C.J. Lokan, D.L. Essam*
School of Computer Science, UC, University of New South Wales
ADFA, Northcott Drive, Canberra,ACT 2600, Australia
{shanyin,rim,cjl,daryl}@cs.adfa.edu.au

Abstract: Knowing the estimated cost of a software project early in the development cycle is a valuable asset for management. In this paper, an evolutionary computation method, Grammar Guided Genetic Programming (GGGP), is used to fit models, with the aim of improving the prediction of software development costs. Valuable results are obtained, significantly better than those obtained by simple linear regression. In this research, GGGP, because of its flexibility and the ability of incorporating background knowledge, also shows great potential in being applied in other software engineering modeling problems.

Keywords: genetic programming, grammar-guided genetic programming, software engineering, software cost estimation

I Introduction

Knowing the estimated cost of a particular software project early in the development cycle is a valuable asset. Management can use cost estimates to evaluate a project proposal or to manage the development process more effectively. Therefore, the accurate prediction of software development cost may have a large economic impact: in fact, some 60% of large projects significantly overrun their estimates and 15% of the software projects are never completed due to the gross misestimation of development [1].The main driver of cost is effort. Thus cost estimation is largely a problem of effort estimation.

A large range of metrics have been proposed for early estimation of software project effort. A number of authors have suggested that the standard sets have too many parameters, and a number of reduced sets have been suggested (large metric sets have high collection costs, and also risk generating over-fitted models). The reductions have relied on linear methods to eliminate metrics, and linear models for estimating size and effort from the metric sets, but there is a risk that some of the dependencies may be non-linear. Researchers elsewhere have begun to investigate alternative methods of developing predictive models, including fuzzy logic, neural networks, and regression trees. Exploration of evolutionary approaches has just begun. [2, 3].

In this paper, an evolutionary approach, Grammar Guided Genetic Programming (GGGP), is used to fit nonlinear models to a dataset of past projects, aiming to determine appropriate metric sets and improve the prediction of software development effort.

In the following Section 2, GGGP is introduced very briefly. The application of GGGP in evolution of software development effort estimation programs is discussed in Section 3. This includes data preparation, GP details and results obtained. The results are analyzed in Section 4. Section 5 draws conclusions.

II Grammar Guided Genetic Programming

One limitation of canonical Genetic Programming (GP) [4, 5] is its requirement of closure. It implies that the function set should be well defined for any combination of arguments. Closure makes many program structures difficult to express. To overcome it, one early approach was Strongly Typed Genetic Programming [6]. Subsequently, a number of authors used grammars to impose syntactical constraints [7, 8, 9, 10]. In a number of these approaches, grammars supplied both a way to rep-

*Corresponding author

resent syntactical constraints, and a way to incorporate background knowledge to guide the process. In this paper, Grammar Guided GP is used to model the software development effort due to the salient advantages of GGGP. Grammars in GGGP can bias the GP individual structure to more efficiently find optimal or near-optimal results. Compared with canonical GP, GGGP has the following advantages:

- With the grammar constraint, the closure requirement in canonical GP is removed so that more expressive program structure can be evolved.
- The grammar in GGGP provides a natural and formalized way to represent background knowledge. With background knowledge, the search space is reduced dramatically.
- Problem related building blocks, a kind of *a priori* knowledge, can be represented through the grammar, further improving search efficiency.
- During the GP search process, the grammar itself can be evolved leading to incremental learning.
- During the overall data mining process, the grammar can be readily modified manually and incrementally to reflect the user's increasing familiarity with the problem. This turned out to be important in this particular research effort, permitting a rapid exploration of the problem.

III Evolution of Software Development Effort Estimation Programs

III.1 Data preparation

The data of 423 software development projects was collected. The projects are drawn from the public ISBSG Data Repository [1]. They range from 9 to 5700 function points, and from 17 to 43000 hours. For each project, there are 32 attributes (Table 1), for example, software size (measured in adjusted function points), effort (measured in hours), team size, defects, platform, development language, team ability, etc. These attributes capture the nature of the project itself, the development environment and techniques used, and strengths and weaknesses perceived in individual projects. They are divided into four categories: numeric variables, unordered categorical variables, ordered categorical variables and boolean variables. Most are known early in a project, which is important for estimation.

Variable Name	Description
Numeric Variable:	
effort	Total project effort (hours)
size	Project size (Function Points)
team_size	Max size of the development team
duration	Duration of the project (months)
defects	Defects found in the 1st month
Unordered Categorical Variable:	
organisation	Organisation type
business	Business area type
application	Application type
development	Development type
platform	Development platform
language_type	Nature of programming language
language	Primary programming language
dbms	Which database system used
intended_market	Relationshp betwn developer&client
Ordered Categorical Variable:	
year	Year completed
user_groups	No. of user groups for determining requirements
importance	The importance of the project
team_ability	development team skills and ability
user_involvement	user involvement
manager_ability	manager's ability or experience
team_experience	development team's experience
requirements	characteristics of requirements
techniques	impact of particular development techniques?
env_tools	suitability/stability of environ./tools
Boolean Variable:	
generic	Written to be portable?
timeboxing	Time-boxing used?
regressiontest	Regression testing used?
prototyping	Prototyping used?
oo	Object-oriented techniques used?
mfnteam	Multi-function teams used?
radjad	RAD/JAD used?
classical	Classical system modelling used?
case	CASE tools used?

Table 1: Project attributes

These 423 records were randomly divided into training and testing data sets, with 211 projects in the training set and 212 in the testing set. In this research, we used GGGP to fit a model, with the aims of determining appropriate metric sets containing the most relevant attributes, and improving

[1] See http://www.isbsg.org.au

the prediction of software development effort.

III.2 Target language

One of the important preparatory steps for GGGP is to identify a suitable target language in which to evolve programs. On one hand, the language should be expressive enough to cover potential solution space. On the other hand, too general a language may ruin the efficiency of execution. Too general a language also obviates one of the most important advantages of employing a grammar: constraining search space. This trade-off needs to be carefully considered.

Two languages were used for evolving software development effort estimation programs. The context free grammars [11, 12] for these languages are in Figure 1. Each of the grammars describes a language, whose expressions may be viewed as mathematic models with numeric return values, and may be interpreted as predictions for software development effort.

Grammar 1:
EXP = PREOP EXP | EXP OP EXP | NUMERIC | CONSTVAR
BOOL = BOOL and BOOL | BOOL or BOOL | not BOOL | EXP CP EXP | ORDERED CP ORDERED_VALUE | UNORDERED in UNORDERED_VALUE_SET | BOOLVAR
PREOP = exp | sqrt | log
OP = + | - | * | / | power
CP = < | > | =
NUMERIC = p1 |···|p4
ORDERED = p5 |···| p13
UNORDERED = p14|··· |p23
BOOLVAR = p 24 |···| p32
CONSTVAR = if BOOL CONSTVAR CONSTVAR | CONSTVAR OP CONSTVAR | <ephemeral const>

Grammar 2:
EXP = PREOP EXP | EXP OP EXP | if BOOL EXP EXP | NUMERIC | CONST
CONST = <ephemeral const>
All the other production rules same as Grammar 1

Figure 1: Grammar of the GP languages

Grammar 1 generates common mathematic expressions with all 32 independent variables and operators, such as +,-,*,/,exp, etc. Note in this grammar, the production for constant:
CONSTVAR = if BOOL CONSTVAR CONSTVAR
This is designed to allow the non-numeric variables to have an influence on the formula through altering the values of constants. This reflects an underlying hypothesis, that all software projects undergo similar processes, and hence may be modelled by a similar expression, but that categorical variables such as implementation language, platform etc may alter the relative contribution of different components of the model.

Grammar 2 is more general, allowing a more complex if-then. It is a superset of Grammar 1. The underlying assumption is that the non-numeric variables may actually change the processes of the software project.

Most of the details of these two grammars should be self-explanatory. ORDERED_VALUE is a constant for the corresponding ordered categorical variables. For example, for ordered categorical variable user_groups, there are three different values: one, one_to_five, over_five. Hence, for user_group, its ORDERED_VALUE is one of these three values. UNORDERED_VALUE_SET is constant value set for corresponding unordered categorical variables. For example, for unordered categorical variable dbms, its possible values set is {access, adabas, db2, ims, ingres, oracle, rdb, others}, which contains eight elements. Therefore, for variable dbms, its corresponding UNORDERED_VALUE_SET is any subset of this set. <ephemeral const> is a randomly generated floating-point constant between 0 and 10. GP parameters are summarized in Table 2.

Parameter	Value
Terminals, non-terminals	(see Fig. 1)
Fitness function	Mean square error
Generation type	Steady state
Selection scheme	Tournament, 3
Population	1000
Max. generations	200
Runs	5
Init. population tree size	Ramped half&half
Min/max depth initial popn	6/9
Probability crossover	0.9
Probability mutation	0.1
Probability internal crossover	0.9
Probability terminal mutation	0.75

Table 2: GP parameters

Mean square error (MSE) was used as fitness function:

$$MSE = (\sum_{i=1}^{N}(estimated\_effort - actual\_effort)^2)/N$$

where N is total number of fitness cases (423), esti-

mated_effort is the prediction from the model, and actual_effort is the actual value.

III.3 Results

Using grammar 1, five GP runs were generated. On the same training and testing data sets, five linear and log-log equations were also derived using standard regression techniques.

Although the GP equations were found by minimizing MSE, and regression also attempts to minimize MSE, estimation equations are commonly assessed using three other criteria. These are: R^2, the amount of variation in the dependent variable explained by variation in the independent variables; Mean Magnitude of Relative Error (MMRE):

$$MMRE = \sum_{i=1}^{N} \frac{|estimated\_effort - actual\_effort|}{actual\_effort} \times \frac{1}{N}$$

and Pred(l), the fraction of projects for which the relative error is less than l%.

Each model was measured on each of these criteria in each of the 5 runs. The means and standard deviations are summarized in table 3.

The first thing to note is that the errors are very large, using any of the models. With MMRE, relative errors average over 100%; only about 20% of projects are predicted to within an error of 25%, and only 40–50% of projects are predicted to within an error of 50%. Large errors are not surprising given the widely varied nature of the data. Better accuracy is obtained in more specialized data sets (as seen in [2, 3]). Large errors are also common in estimates made early in a project [13].

On the basis of table 3, our results are quite promising.

Firstly, it is clear that the intuition which gave rise to the use of log regression has misfired. The training error of log regression is significantly *larger* than that of linear regression, although it generalizes marginally better on the testing data set than linear regression.

Secondly, GP has been able to dramatically improve the situation. The GP MSE is far better than both linear and log regression models on the training data. Although there is some loss of accuracy on the test set — i.e. the GP model does not generalize perfectly — it is still far more accurate in testing error than the two regression models.

GP performs better than linear regression models in all respects. Log regression models perform much worse than GP on MSE, about the same as GP on R^2 and Pred(25%), and better than GP on MMRE and Pred(50%). One way of viewing this is that GP has more effectively fit the objective, namely minimising MSE, at the cost of increased error on other measures. Since GP is indifferent to the particular objective function it maximises, GP could also be used to minimise MMRE, in which case its performance on MMRE would be expected to considerably outperform log regression.

According to Table 3, in terms of MSE, linear and log-log regression perform better on testing dataset than on training dataset. After analyzing the dataset, we found that the reason is in our limited number of trials (5 in our experiment) with small number of training and testing cases (211 and 212 respectively). Dominant cases, which impact the result, are distributed unevenly in training and testing datasets.

		Linear		Log		GP	
		Mean	STD	Mean	STD	Mean	STD
MSE	Train	18.1	3.0	20.5	3.5	2.90	0.47
	Test	15.4	2.6	14.5	3.8	5.40	0.61
R^2	Train	0.45	0.04	0.48	0.04	0.44	0.07
	Test	0.36	0.05	0.41	0.08	0.40	0.08
MMRE	Train	2.51	0.12	1.29	0.19	2.67	0.81
	Test	1.94	0.23	1.04	0.10	1.91	0.67
Pred (25%)	Train	0.17	0.01	0.22	0.04	0.19	0.02
	Test	0.20	0.03	0.23	0.02	0.21	0.02
Pred (50%)	Train	0.35	0.01	0.45	0.07	0.39	0.06
	Test	0.37	0.03	0.51	0.02	0.40	0.06

Table 3: Mean and standard deviation of comparison measures for linear, log-log and GP models. (MSE scaled by 10^{-6})

IV Discussion

In the linear and log regression models, size and team_size are the main drivers of project effort. Size is most important. Two typical best-performed GP models are listed in Fig 2. Generally, the best-performed GP models include a linear component related to size (this term usually dominates the predicted effort). In addition to the linear component, the best-performed GP models also include various terms involving team size, and non-linear terms involving size (e.g. log(size)). This suggests that project effort is not just a linear function of project size, but is actually a complex function involving team size as well.

effort = size × (if application in {dss,missing}
then 5.34 else 1.68)
+ 18.5 × team_size × log(size)
+ 92.7 × log(size)

effort = 2×team_size^2 + 2×team_size×sqrt(size) + 4×size
+ 2×team_size + 1246

Figure 2: Typical GP models

With the successful application of GP in this software engineering program, we turned to Grammar 2, which is more general than Grammar 1. Not only common mathematical expressions but also complex if-then clauses can be generated in Grammar 2. In Grammar 1, if-then performs a relational test in order to evolve proper constants for the whole model while, in Grammar 2, the statement part in if-then can be highly complicated. However, no better result was found. After studying the results of Grammar 1, we discovered that this outcome is not surprising. In most of the best-of-run individuals discovered by Grammar 1, there is no if-then clause. This suggests that even in the evolution of constants, those non-numerical attribute only play a small role, as non-numeric variables can only appear in the condition part of if-then. Therefore, it is natural to expect that these non-numeric attributes would not impact much on the evolved programs of Grammar 2.

But why should non-numerical attributes show such little influence on estimated software development effort? After all, experienced software engineers have collected these attributes precisely because they believed them to be relevant to software development effort.

Two obvious conjectures are that

- the non-numerical attributes are not closely related to the software development effort or
- the combination of these non-numerical attributes are too complex to be discovered by GP.

We incline to the latter. Intuitively, some non-numerical attributes, such as whether a fourth generation language is used, or whether object oriented techniques are employed, should influence the software productivity perceptibly. However we believe that the complexity of the search space, combined with the relative paucity of data points, masks these effects. Our further research will consider in more detail how to discover and model these factors.

Other further research issues, that we will explore in the future, include:

- As we can see in our experiment, little background knowledge has been incorporated into either of the grammars. We intend to work cooperatively with domain experts to incorporate background knowledge in order to bias the search space, which should lead to more accurate prediction, and may ameliorate the problems of large search space and small dataset.
- The grammar itself can be automatically refined during the GP process. It is likely that novel patterns, leading to increased understanding, may be discovered through the analysis of the automatically refined grammar.
- Another key research issue is how to deal with missing values in the use of genetic programming for data mining. There are a large number of missing values in our data set. This will often be the case for real world data mining applications. In other fields of data mining, this problem has been investigated in detail, and a variety of methods for handling missing values are used. We intend to investigate their applicability in GP-based data mining.

V Conclusions

Accurate estimation of software development effort is important for the software industry. The research in this paper successfully used GP for evolving solutions to this problem. GP found models that are dominated by the same key predictors of effort as traditional models. GP models perform better than traditional linear models — on all criteria, not just on the criterion that was used in the GP fitness function.

Compared with other learning techniques, GP can be used to fit complex function and theoretically its outcome is interpretable. Most of other learning techniques simply can't discover complex functions as we involved using GP. A few other techniques [14, 15] can be used for nonlinear regression but we have to make good pre-guesses as to what functions may be discovered. Artificial neural network can fit the models, but whose outputs would then be black-box while quite often we do expect the model we discovered can be interpreted.

In this experiment, GGGP because of its flexibility and the ability to incorporate background knowledge, also shows great potential in application to other software engineering modeling problems.

References

[1] M. Boraso, C. Montangero, and H. Sedehi. Software cost estimation: an experimental study of model performances. Technical Report TR-96-22, DEPARTIMENTO DI INFORMATATICA, UNIVERSITA DI PISA, Italy, 1996.

[2] J.J. Dolado. On the problem of the software cost function. *Information and Software Technology*, 43(1):61–72, January 2001.

[3] C.J. Burgess and M. Lefley. Can genetic programming improve software effort estimation? a comparative evaluation. *Information and Software Technology*, 43(14):863–873, December 2001.

[4] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[5] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[6] David J. Montana. Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA, 7 May 1993.

[7] Frederic Gruau. On using syntactic constraints with genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 19, pages 377–394. MIT Press, Cambridge, MA, USA, 1996.

[8] P. A. Whigham. Grammatically-based genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, California, USA, 9 July 1995.

[9] Man Leung Wong and Kwong Sak Leung. Combining genetic programming and inductive logic programming using logic grammars. In *1995 IEEE Conference on Evolutionary Computation*, volume 2, pages 733–736, Perth, Australia, 29 November - 1 December 1995. IEEE Press.

[10] Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transaction on Evolutionary Compuation*, 5(4):349–358, 2001.

[11] D. A. Gustafson, W. A. Barrett, R. M. Bates, and J. D. Couch. *Compile construction: Theory and Practice*. Science Research Assoc, Inc., 1986.

[12] P.A. Whigham. *Grammatical Bias for Evolutionary Learning*. PhD thesis, School of Computer Science, University College, Univ. of New South Wales, Australia, 1996.

[13] S.S. Vicinanza, T. Mukhopadhyay, and M.J. Prietula. Software-effort estimation: an exploratory study of expert performance. *Information Systems Research*, 2(4):243–262, December 1991.

[14] S. Dzeroski and L. Todorovski. Discovering dynamics: From inductive logic programming to machine discovery. *Journal of Intelligent Information Systems*, 4:89–108, 1995.

[15] L. Todorovski and S. Dzeroski. Declarative bias in equation discovery. In *Proceedings of Fourteenth International Conference on Machine Learning*, pages 376–384, San Mateo, CA, 1997. Morgan Kaufmann.