Adapting the Fitness Function in GP for Data Mining

J. Eggermont, A.E. Eiben, and J.I. van Hemert

Leiden University, P.O. Box 9512
2300 RA, Leiden, The Netherlands
{jeggermo,gusz,jvhemert}@cs.leidenuniv.nl

Abstract. In this paper we describe how the Stepwise Adaptation of Weights (SAW) technique can be applied in genetic programming. The SAW-ing mechanism has been originally developed for and successfully used in EAs for constraint satisfaction problems. Here we identify the very basic underlying ideas behind SAW-ing and point out how it can be used for different types of problems. In particular, SAW-ing is well-suited for data mining tasks where the fitness of a candidate solution is composed by 'local scores' on data records. We evaluate the power of the SAW-ing mechanism on a number of benchmark classification data sets. The results indicate that extending the GP with the SAW-ing feature increases its performance when different types of misclassifications are not weighted differently, but leads to worse results when they are.

1 Introduction

In constraint satisfaction problems (CSP) a set of variables is given together with their domains and a number of constraints on these variables. The task is to find an instantiation of the variables such that all constraints are satisfied. A commonly used approach to constraint satisfaction problems in evolutionary computation is the use of penalty functions. Thus, an evolutionary algorithm (EA) operates on populations consisting of vector instantiations as candidates and the fitness of an arbitrary candidate is computed by adding up the penalties for violating the given constraints. Formally, the fitness function f is defined as:

$$f(x) = \sum_{i=1}^{k} w_i \cdot \chi(x, i) \ . \tag{1}$$

where k is the number of constraints, w_i is the penalty (or weight) assigned to constraint i, and

$$\chi(x, i) = \begin{cases} 1 \text{ if } x \text{ violates constraint } i \\ 0 \text{ otherwise} \end{cases} \ . \tag{2}$$

One of the main drawbacks to this approach is that the penalties, or weights, for constraints need to be determined in accordance with the hardness of the

R. Poli et al. (Eds.): EuroGP'99, LNCS 1598, pp. 193–202, 1999.

constraints. After all, the EA will primarily 'concentrate' on satisfying those constraints that carry the highest penalties. Nevertheless, to determine how weights should be assigned to constraints appropriately, might require substantial insight into the problem, which might not be available, or only at substantial costs.

In the Stepwise Adaptation of Weights (SAW) mechanism this problem is circumvented by letting the EA defining the weights itself. In a SAW-ing EA the weights are initially set at a certain value (typically as $w_i = 1$) and these weights are repeatedly increased with a certain step size Δw during the run. The general mechanism is presented in Figure 1.

On-line weight update mechanism
  set initial weights (thus fitness function f)
  while not termination **do**
    for the next T_p fitness evaluations **do**
      let the EA go with this f
    end for
    redefine f and recalculate fitness of individuals
  end while

Fig. 1. Stepwise adaptation of weights (SAW)

Redefining the fitness function happens by adding Δw to the weights of those constraints that are violated by the best individual at the end of each period of T_p fitness evaluations. This mechanism has been successfully applied to hard CSPs, such as graph 3-coloring [4,5], 3-SAT [1,3], and randomly generated binary CSPs [6,9]. Extensive tests on graph coloring and 3-SAT [2,8,13] showed that algorithm performance is rather independent from the values of T_p and Δw, thus they need not to be fine tuned.

Looking carefully at the SAW-ing mechanism one can observe that its applicability is not restricted to constrained problems. The basic concept behind SAW-ing is that the overall quality of a candidate solution is determined by 'local scores' on some elementary units of quality judgment, like constraints in a CSP. Then, the quality of a candidate solution (the fitness used in an EA) can be defined as a weighted sum of these local scores, where the weights should reflect the importance, respectively hardness of the elementary units of quality judgment.

A classification problem, as perceived in the rest of this paper, is defined by a data set consisting of data records, where each of the records is assigned a label, its class. The task is to find a model that takes a record as input and gives the class of the record as output. A natural way of comparing models is by their classification accuracy on the whole data set, the perfect model would generate the right class for every (known) record. It is obvious that this problem fits the above description: the overall quality of a candidate solution (the accuracy of a model on the whole data set) is determined by local scores on some elemen-

tary units of quality judgment (data records). Thus, an evolutionary algorithm searching for a good model classifying the given records in a data set D could use any suitable representation of such models and the fitness function can be defined similarly to Equation 1 as follows.

$$f(x) = \sum_{r \in D} w_r \cdot \chi(x, r) \; . \tag{3}$$

where $\chi(x, r)$ is now defined as

$$\chi(x, r) = \begin{cases} 1 \text{ if } x \text{ classifies data record } r \text{ incorrectly} \\ 0 \text{ otherwise} \end{cases} \tag{4}$$

More generally, the following formula can be used.

$$f(x) = \sum_{r \in D} w_r \cdot \mathit{error}(x, r) \; . \tag{5}$$

where $\mathit{error}(x, r)$ is a measure of misclassification, generalizing the simple good/no-good characterization by $\chi(x, r)$ in Equation 4.

2 A Library for Evolutionary Algorithm Programming

All experiments reported here are performed using our Library for Evolutionary Algorithm Programming (LEAP). This library differs from the many libraries for programming evolutionary algorithms that have been build as a toolkit, i.e., a loosely connected set of building blocks, that put together in the right way provides a user with a functioning program. The problem with a toolkit is that it is often difficult to learn and to maintain. Also most of them are aimed at one specific area within evolutionary computation.

By using a *framework* instead of a toolkit we can overcome these problems. A framework does not supply the user with loosely connected building blocks, instead it provides an almost running algorithm. The user only has to put in the last pieces of the puzzle and maybe has to change the parts of the framework that are not appropriate for the problem. The framework will then provide a running evolutionary algorithm using the provided pieces, substituting the changed parts.

When additions are made to the library, programs made with it can easily make use of these additions, by only changing some lines of code in a specific and predetermined place. As long as the new method is compatible with the old one, in a high level specification sense, the library will produce a new evolutionary algorithm without much extra work. A selection mechanism could easily be tested on different kind of algorithms, thus providing an easy way of sharing techniques between different areas of research.

A preliminary version of LEAP can be downloaded from its Internet site[1]. The library is programmed in C++, using the Standard Template Library (STL)[2]

[1] LEAP: `http://www.wi.leidenuniv.nl/~jvhemert/leap/`
[2] Available at: `http://www.sgi.com/Technology/STL/`

and comes equipped with a programmers manual [10]. It has been build using techniques from the paradigm Design Patterns [7]. A more detailed description of LEAP can be found in [9].

3 An adaptive GP for data classification

The adaptive GP we study in this paper deviates in two aspects from the usual GP for data classification. The first, and most important modification is the usage of the SAW-ing mechanism that repeatedly redefines the fitness function. The second modification concerns the representation. Namely, here we apply a representation based on so-called atomic expressions as leaves and only Boolean operators in the body of the trees. This implies some changes in the implementation of the mutation operator. The most important parameters of our GP are summarized in Table 1, for more details we refer to [9].

Table 1. Main parameters of the adaptive GP

Parameter	Value
Function set	{`and`, `or`, `nand`, `nor`}
Atom set	attribute greater or less than a constant
Initial maximum tree depth	5
Maximum number of nodes	200
Initialization method	ramped half-and-half
Algorithm type	steady-state
Population size	1000
Parent selection	linear ranking
Bias for linear ranked selection	1.5
Replacement strategy	replace worse
Stop condition	perfect classification or 40000 fitness evaluations
Mutation type	1. subtree replacement 2. subatomic mutation
Subatomic d parameter	0.1
Mutation probability	0.1
Crossover	swap subtrees
Crossover probability	0.9
Crossover functions:atoms ratio	4:1
SAW-ing update interval T_p	100
SAW-ing Δw parameter	1
SAW-ing initial weights	1

3.1 Representation and mutation using atoms

In the design of the representation we are deviating from the idea of having a function set of real valued operators and a special operator in the root of the

tree that chooses the class depending on the values given by the subtrees [11]. Instead, we process numerical information at the leaves of a tree, transform it into Boolean statements, and apply only Boolean functions in the body of the tree. This is meant to increase the readability of the emerging models.

An *atom* is syntactically a predicate of the form *operator*(*var*, *const*), built up from the following three items:

1. a variable indicating a field in the data set,
2. a constant between 0 and 1, and
3. a comparing operator, denoted by $A_<$ and $A_>$.

Evaluating an atom *operator*(*var*, *const*) on a record r within this syntax amounts to determining whether the value standing in the field *var* of r is smaller (for $A_<$) or larger (for $A_>$) than the given constant *const*, returning a Boolean value as answer. Notice that our GP fulfills the closure property because atoms produce a Boolean output after processing the numerical arguments. In this representation the conditional part of a rule could look like:

$$(A_>(r_1, 0.347) \texttt{ nor } A_<(r_0, 0.674)) \texttt{ and } A_>(r_1, 0.240)$$

which, in turn, is represented by the tree in Figure 2.

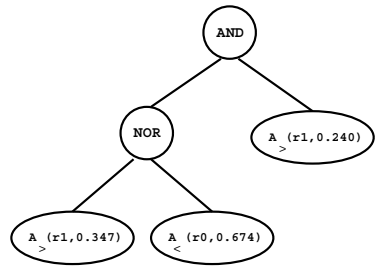

Fig. 2. Representation of a classification rule as a tree.

The new representation also gives rise to a new operator, called subatomic mutation. Every time an individual is selected for a mutation, we first choose a node in the tree to work on. If this node is part of the function set, a subtree mutation will be performed. If this node is a leave (an atom), we choose with equal chance if this will be a subtree mutation or a subatomic mutation. A *subatomic mutation* works by first selecting, with equal chance, if the operation will be performed on the variable or on the constant. In case of a variable we just randomly select a new variable. In case of the constant c a small number Δc ($-d < \Delta c < d$) is generated which is then added to the constant as follows[3]:

$$c' = \begin{cases} 0, & \text{if } c + \Delta c < 0, \\ 1, & \text{if } c + \Delta c > 1, \\ c + \Delta c, & \text{otherwise.} \end{cases}$$

[3] The values for all records are between 0 and 1 in the data sets we consider.

3.2 SAW-ing method for GP

As explained in the introduction, extending a GP with the SAW-ing technique can be done by using a fitness function in the spirit of Equation 3 or 5. The test suite we use for the experiments consists of binary classification problems, where exactly two disjoint classes are present. Denoting these classes by A and $\bar{A}$, it is thus sufficient to evolve models (classification rules) for class A only in the form of

$$\mathit{condition}(r) \longleftrightarrow \mathit{class}(r) = A,$$

where r is a record and the candidate solutions (trees in the population) represent the expression $\mathit{condition}(\cdot)$. Due to the Boolean typing of our trees, we can directly interpret a tree x on a record r as true or false. According to the above formula we have that a tree x classifies r into A iff x evaluates to true on r. To simplify explanations later on we introduce the notation $\mathit{class}(r)$ for the real class of r and

$$\mathit{predicted}(x, r) = \begin{cases} A, \text{ if } x \text{ is true on } r, \\ \bar{A}, \text{ if } x \text{ is false on } r. \end{cases}$$

for the class where a tree x classifies r into. With this notation we redefine χ from Equation 4 to become:

$$\chi(x, r) = \begin{cases} 1 & \mathit{predicted}(x, r) \neq \mathit{class}(r) \\ 0 & \mathit{predicted}(x, r) = \mathit{class}(r) \end{cases}$$

and the fitness function is to be minimized. The weights w_r used in the fitness function are initially set to one. During a run, after every T_p evaluations the GP is interrupted and the weights are changed. A weight w_r belonging to record r is increased with Δw if the best individual does not classify r correctly. After this phase all individuals have to be re-evaluated using the new fitness function.

Note that the stop condition of our algorithm is based on counting the number of fitness evaluations. This is in accordance with the common practice of generate-and-test style search methods. Because each newly created candidate solution is immediately evaluated, the number of evaluations equals the number of points visited in the search space. Using the SAW-ing technique, however, an extra overhead of re-evaluations is created and it could be argued that we should count the re-evaluations too. We do not do this for two reasons. Firstly, it is the number of points visited in the search space that we really want to count. Secondly, we can minimize the re-evaluation overhead by simply caching the classification results for each individual. That is, the first time an individual is evaluated it is assigned an additional bit-vector with the length of the number of records. In this vector we store the result of each classified record. This enables us to skip the time consuming evaluation of the individual x on each r when x is re-evaluated, instead we can suffice with looking up the values from this vector.

The algorithm using the SAW-ing mechanism will be called GP+SAW and the one without GP.

4 Experiments and results

We have compared the GP and GP+SAW techniques to other data classifying algorithms on four different data sets from the Statlog[4] [12] data set. All the experiments involve an n-fold cross validation. The results are obtained by averaging the classification error or misclassification cost over the n folds. We compare our GP and GP+SAW with four algorithms selected from the Statlog project. All the selected algorithms were best in one of the data sets, and showed a reasonable performance in the others. Also the default performance is reported, which is calculated by using the very simple classifying rule of always going for the safe prediction. For example in heart disease it is safe to say everyone has a heart disease.

The first experiments involve the Australian Credit Approval and Pima Indians Diabetes data sets. The Australian Credit Approval data set has 690 records and a 10-fold cross validation is performed. The Pima Indians Diabetes set has 768 records, and here a 12-fold cross validation is performed. When running the algorithms GP and GP+SAW we used the fitness function according to Equation 3 and the definition of χ above. The comparison is based on the average percentage of wrongly classified records in the test sets. The results in Table 2 show that the performance of GP+SAW is better than GP. For the Diabetes data set GP+SAW even manages to beat one (NaiveBay) of the Statlog algorithms.

Table 2. Classification error for the test phase on the Australian Credit Approval and Pima Indians Diabetes data sets

algorithm	Australian	Diabetes
Cal5	0.131	0.250
Discrim	0.141	0.225
LogDisc	0.141	0.223
NaiveBay	0.151	0.262
GP	0.246	0.283
GP+SAW	0.242	0.258
Default	0.440	0.350

The German Credit and Heart Disease data sets have respectively a size of 1000 and 270 records. Here the experiments consist of a 10-fold and a 9-fold cross validation test. The results are compared using a measure called the Average Misclassification Cost. This is calculated by using a cost matrix, which assigns a cost to each pair of true (rows) and predicted (columns) values. The cost matrix for these two data sets is the same:

class	good/absent	bad/present
good credit/heart disease absent	0	1
bad credit/heart disease present	5	0

[4] Available at: `http://www.ncc.up.it/liacc/ML/statlog`

According to this table we ran the algorithms GP and GP+SAW on these two problems using the fitness function in Equation 5 and the definition of $error(x, r)$ as follows.

$$error(x,r) = \begin{cases} 0 \ predicted(x,r) = class(r) \\ 1 \ predicted(x,r) = A \text{ and } class(r) = \bar{A} \\ 5 \ predicted(x,r) = \bar{A} \text{ and } class(r) = A \end{cases}$$

where class A stands for 'good credit' and 'heart disease absent', respectively.

Results show that for the German Credit data set, GP has a reasonable performance, obtaining a classification cost close to the best. However for the Heart Disease data set, both GP and GP+SAW have an inferior performance, although still better than the Default measure.

Table 3. Average Misclassification Costs for the German Credit and Heart Disease data set

algorithm	German	Heart
Cal5	0.603	0.444
Discrim	0.535	0.393
LogDisc	0.538	0.396
NaiveBay	0.703	0.374
GP	0.579	0.456
GP+SAW	0.943	0.537
Default	0.700	0.560

5 Conclusions and further research

The basic motivation for this study comes from the observation that the composition of the fitness function in a penalty based EA for CSPs is very similar to that of an EA (GP) for data classification. It is thus a natural question to investigate whether the SAW-ing mechanism, which can substantially increase the performance of an EA on CSPs, can lead to improvements on data classification problems as well.

Regarding the results of our experiments two cases can be distinguished. In case of the Australian Credit Approval and the Pima Indians Diabetes data sets GP+SAW clearly outperforms GP alone. This confirms that SAW-ing forms a useful extension of the standard machinery, leading to better results at low costs. On the German Credit and the Heart Disease data sets, however, the outcomes are reversed. When looking for an explanation of this result it immediately occurs that the latter two problems differ from the first two, because a cost matrix biases the measurement of misclassifications. It could be hypothesized that this 'skewed' measurement interferes negatively with the re-weighting mechanism of SAW-ing, misleading the GP+SAW algorithm. Namely, the SAW-ing mechanism

increases the weights of misclassified records, regardless to the low (1) or high (5) contribution of this misclassification to the fitness value (to be minimized).

Current research is concerned with further investigation of the German Credit and the Heart Disease problems. The first experiments support the hypothesis that the inferiority of GP+SAW is caused by the 'skewed' measurement of misclassifications. Evaluating the outcomes of the experiments disregarding the cost matrix during validation, counting only the percentage of misclassifications as a result, yields better results for GP+SAW as can be seen in Table 4.

Table 4. Classification error for the test phase for the German Credit and Heart Disease data set

algorithm	German	Heart
GP	0.382	0.278
GP+SAW	0.351	0.270

Although these outcomes are not anymore comparable with the benchmark techniques from Statlog, they provide evidence that SAW-ing can increase GP performance on the third and fourth data sets too. Presently we are working on a modified version of the SAW-ing mechanism that does take the cost matrix into account when re-defining weights of records. We are also testing the GP and the GP+SAW algorithms on large real world data sets from finance.

Future research is divided into two main parts. The first part concerns the implementation of the traditional GP approach to data classification tasks, and its comparison with the present variant where we use an atom-based representation and a steady-state population model. The second part is the development of LEAP, where the main focus will be on the extension of the functionalities and the possible integration of LEAP with another library called EO (Evolvable|Evolutionary objects)[5].

References

1. Th. Bäck, A.E. Eiben, and M.E. Vink. A superior evolutionary algorithm for 3-SAT. In V. William Porto, N. Saravanan, Don Waagen, and A.E. Eiben, editors, *Proceedings of the 7th Annual Conference on Evolutionary Programming*, number 1477 in LNCS, pages 125–136. Springer, Berlin, 1998.
2. A.E. Eiben and J.K. van der Hauw. Graph coloring with adaptive evolutionary algorithms. Technical Report TR-96-11, Leiden University, August 1996. Also available as `http://www.wi.leidenuniv.nl/~gusz/graphcol.ps.gz`.
3. A.E. Eiben and J.K. van der Hauw. Solving 3-SAT with adaptive Genetic Algorithms. In *Proceedings of the 4th IEEE Conference on Evolutionary Computation*, pages 81–86. IEEE Press, 1997.

[5] Available at `http://geneura.ugr.es/~jmerelo/EO.html`

4. A.E. Eiben and J.K. van der Hauw. Adaptive penalties for evolutionary graph-coloring. In J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificial Evolution'97*, number 1363 in LNCS, pages 95–106. Springer, Berlin, 1998.
5. A.E. Eiben, J.K. van der Hauw, and J.I. van Hemert. Graph coloring with adaptive evolutionary algorithms. *Journal of Heuristics*, 4(1):25–46, 1998.
6. A.E. Eiben, J.I. van Hemert, E. Marchiori, and A.G. Steenbeek. Solving binary constraint satisfaction problems using evolutionary algorithms with an adaptive fitness function. In A.E. Eiben, Th. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Proceedings of the 5th Conference on Parallel Problem Solving from Nature*, number 1498 in LNCS, pages 196–205, Berlin, 1998. Springer.
7. E. Gamma, R. Helm, R. Johnson, and J.Vlissides, *Design Patterns: elements of reusable object-oriented software.* Addison-Wesley, 1994.
8. J.K. van der Hauw. Evaluating and improving steady state evolutionary algorithms on constraint satisfaction problems. Master's thesis, Leiden University, 1996. Also available as `http://www.wi.leidenuniv.nl/MScThesis/IR96-21.html`.
9. J.I. van Hemert. Applying adaptive evolutionary algorithms to hard problems. Master's thesis, Leiden University, 1998. Also available as `http://www.wi.leidenuniv.nl/~jvhemert/publications/IR-98-19.ps.gz`.
10. J.I. van Hemert. *Library for Evolutionary Algorithm Programming (LEAP).* Leiden University, LEAP version 0.1.2 edition, 1998. Also available at `http://www.wi.leidenuniv.nl/~jvhemert/leap`.
11. J.R. Koza. *Genetic Programming.* MIT Press, 1992.
12. D. Michie, D.J. Spiegelhalter, and C.C. Taylor, editors. *Machine Learning, Neural and Statistical Classification.* Ellis Horwwod, February 1994.
13. M. Vink. Solving combinatorial problems using evolutionary algorithms. Master's thesis, Leiden University, 1997.