Available online at www.sciencedirect.com
ScienceDirect
Pattern Recognition 40 (2007) 2696–2705
PATTERN RECOGNITION
THE JOURNAL OF THE PATTERN RECOGNITION SOCIETY
www.elsevier.com/locate/pr

Neutral offspring controlling operators in genetic programming

Liang Zhang, Asoke K. Nandi*

Signal Processing and Communications Group, Department of Electrical Engineering and Electronics, The University of Liverpool, Brownlow Hill, Liverpool L69 3GJ, UK

Received 19 December 2005; received in revised form 27 September 2006; accepted 20 October 2006

Abstract

Code bloat, one of the main issues of genetic programming (GP), slows down the search process, destroys program structures, and exhausts computer resources. To deal with these issues, two kinds of neutral offspring controlling operators are proposed—non-neutral offspring (NNO) operators and non-larger neutral offspring (NLNO) operators. Two GP benchmark problems—symbolic regression and 11-multiplexer—are used to test the new operators. Experimental results indicate that NLNO is able to confine code bloat significantly and improve performance simultaneously, which NNO cannot do.

Keywords: Genetic programming; Neutral offspring; Code bloat; Parsimony pressure

1. Introduction

The tendency of programs to grow drastically without significant corresponding improvement of performance (code bloat), one of the main issues of GP, has been noticed since GP was first proposed [1], and well-documented in the literature [2–6]. It also occurs in other evolutionary techniques with variable length representations [5,7]. Code bloat limits the search of the solution space by slowing down the breeding and evaluation of the candidate solutions; it also ceases the evolutionary process by destroying the structure of the programs; additionally, huge programs exhaust available memory and computational resources. Many efforts have been made to understand the causes of code bloat and to find solutions to it. Three main hypotheses about the causes behind this phenomenon have been explored in the literature [8–11], and given empirical analysis [12,13]. The terms of *intron* (also called inviable, inoperative, non-effective, or neutral code) and *exon* are usually used when discussing code bloat [14,15].

- *Introns* are the nodes in which any change causes no change to the program's performance.
- *Exons* are the nodes that are not introns.

These hypotheses are:

- Protective effect against destructive nature of modifying operators [8]. GP derived from Genetic Algorithm (GA) is the operation of modifying operators (crossover and mutation), which replace a piece of code (a node or a subtree in GP) of a program. Unfortunately unlike GA, the removed piece of code of a GP program does not serve the same function as the added piece. This added piece of code is, to a large extent, random to the receiver. Therefore, such modifying operations, if they work on exons, usually produce poorer offspring; while operations on introns generate neutral offspring (bearing the same performance with its parent) [16]. Therefore programs with larger proportion of introns have a better chance to produce a neutral offspring, instead of a usually poorer one. Hence programs with more introns have a better chance to survive through the evolutionary process, i.e., there is a bias towards generating more introns in the population.

* Corresponding author. Tel.: +44 151 794 4525; fax: +44 151 794 4540.
E-mail address: a.nandi@liverpool.ac.uk (A.K. Nandi).

0031-3203/$30.00
doi:10.1016/j.patcog.2006.10.001

- Drift towards larger programs. Langdon and Poli [17] argued that the distribution of the semantically equivalent programs in the solution space has a positive effect on code bloat. Because of the existence of introns, given a program, infinitely larger programs with the same behavior can be produced by adding different nodes under introns in the program tree. Therefore the population is likely to drift towards larger programs because there are more of them with better performance. Furthermore, exons have similar effect too. Given a program, there are limited smaller ones and unlimited larger ones, therefore there are many more larger programs with very similar performance and different exon structures than smaller ones. Hence there is a better chance to form larger programs of similar performance with less effective code. Therefore by this hypothesis, programs drift towards larger size by increasing introns and less effective exons.
- Removal bias. Soule and Foster [10] in their paper proved that introns tend to occur further from the root than exons do. Even for the exons, the nearer a node is to the root, it is generally more significant to the output. We can conclude that removing larger subtrees has more effect on the program behavior, hence it tends to decrease the performance to a greater extent. Meanwhile, the added subtree, due to its randomicity, does not have the same functional contribution as the removed one. Therefore in a subtree substitution operation, there is a negative relationship between the size of the removed subtree and the offspring performance, whereas there is no evident relationship between the size of the added subtree and the offspring performance.

Many different approaches have been considered to fight against code bloat: parsimony pressure (modifying fitness value to penalize large programs) [18,19], code editing [20], and many modifications to the operators [3,4,21,22]. We focus on neutral offspring (the offspring with the same performance with their parents). Neutral offspring are believed to be mainly created by operating on introns, because it is reasonable to assume that most offspring that are created by modifying exons change their fitness values, especially for problems with continuous fitness values in the solution space, such as symbolic regression. Hence dealing with neutral offspring provides a chance to control introns, and thereby code bloat.

Section 2 presents the neutral offspring controlling operators. Section 3 presents a diversity preserving mechanism that is necessary for applying the new methods. Section 4 discusses parsimony pressure used in our experiments. Section 5 introduces offspring distribution, which is used to analyze intron growth. Section 6 describes the two problems under test. Experiments, both with and without parsimony pressure, are discussed in Section 7, which are summarized in Section 8, and concluded in Section 9.

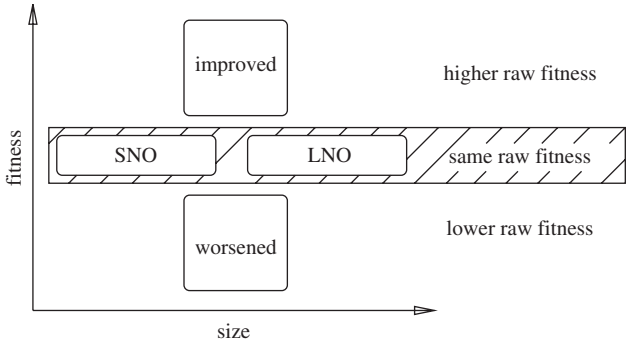

Fig. 1. Offspring division with respect to the comparison of fitness and size with their parents.

2. Neutral offspring controlling operators

Offspring can be divided into *improved offspring*, *neutral offspring* and *worsened offspring*, with respect to their fitness in comparison with their parents. We can further separate the neutral offspring that are larger in size than their parents (LNO) and those that are smaller in size than their parents (SNO) (Fig. 1). An LNO actually has the same exon structure with its parent, but with more introns, and SNO has the same exon structure with less introns. By the protective hypothesis, the evolutionary process favors LNOs, which results in intron growth with no performance improvement. By the consideration that neutral offspring only affect introns, and do not change the behavior of the program, we propose the approach that discards all neutral offspring, named *non-neutral offspring* operators (NNO). Meanwhile since SNO reduces intron growth, introns should be even confined if we keep SNOs and discard LNOs only. We shall name this approach *non-larger neutral offspring* (NLNO). These two approaches both confine intron growth, but with different degree. The protective hypothesis argues that code bloat occurs as intron growth; the drift hypothesis also assumes that code bloat is caused by the growth of introns and less effective code. Hence these two hypotheses are weakened by the two newly proposed approaches. The removal bias hypothesis, however, does not seem to be weakened by the new methods. Therefore, by applying these two approaches, introns should be limited, so is code bloat thereby.

As some or all the neutral offspring are to be discarded, other programs have to be selected to fill the new generation. One straightforward way is to copy the parent program to the new generation. However, while these offspring are discarded and their parents are copied, the crossovers and mutations become reproductions. Thus with much more reproduction than necessary, the best program of each generation gets multiple copies every generation, which inevitably leads the population to premature convergence, and halt the further search. To cope with this matter, we introduce a mechanism to preserve diversity.