Can Tree Adjunct Grammar Guided Genetic Programming be Good at Finding a Needle In a Haystack? A Case Study

N.X.Hoai[1], *R.I. McKay*[2], *and D. Essam*[2]
School of Computer Science,
University of New South Wales,
ADFA campus, Canberra, ACT 2600, Australia,
[1] x.nguyen@student.adfa.edu.au
[2]rim,daryl@cs.adfa.edu.au

Abstract – In this paper we experiment TAG3P on the even parity problems in order to investigate the robustness of tree-adjunct grammar guided genetic programming [3] (TAG3P) on the problems classified as "finding a needle in a haystack" [9]. We compare the result with grammar guided genetic programming [15] (GGGP) and genetic programming [7] (GP). The results show that TAG3P does not work well on the problems due to the nature of the search space and the structure of the solution.

Keywords: Genetic Programming, Grammar – Guided Genetic Programming, Tree Adjunct Grammars, Even-Parity Problem.

I. Introduction

Tree adjunct grammar guided genetic programming [3] (TAG3P) is a grammar guided genetic programming system that uses tree adjunct grammars along with context free grammars as means to set bias for the evolutionary process. In [4], we showed that TAG3P outperforms significantly GGGP and GP on the symbolic regression problem whereby the target functions and the search space are suitable for the promotion of building blocks. In this paper, we experiment TAG3P on the even parity problem of which the nature of search space is like a needle in a haystack that makes it difficult to solve with any progressive search techniques [9]. The result is then compared to GGGP and GP on the same problem with similar settings on parameters. The organization of the remainder of the paper is as follow. In section 2, we give some basic concepts of GP, GGGP, TAG3P, and the concepts of building blocks. Section 3 describes the even parity problem. Section 4 contains our experimental setups. The results will be given and discussed in section 5. Section 6 concludes the paper and discusses future work.

II. Backgrounds

In this section, we briefly overview some basic components and operations of the three different genetic programming systems, namely, canonical genetic programming [2] (GP), grammar guided genetic programming [15] (GGGP), tree adjunct grammar guided genetic programming [3] (TAG3P) and the concept of building blocks.

I.1. Genetic Programming

Genetic programming (GP) can be classified as an evolutionary algorithm, in which computer programs are the evolutionary targets. An early definition, model, techniques and problems of genetic programming can be found in [7]. For a good survey of genetic programming, [1] is recommended. A basic genetic programming system consists of five basic components [7]: representation for programs (called genome structure), a procedure to initialize a population of programs, a fitness to evaluate the performance of the program, genetic operators, and parameters. In [7], the structure of programs is the structured tree of S-expressions; fitness of a program is evaluated by its performance; main genetic operators are reproduction, crossover, and mutation. Reproduction means some programs are copied to the next generation based on their fitness, crossover can be carried out between two tree-based programs by swapping two of their sub-trees,[1] and a tree-based program can be mutated by replacing one of its sub-trees by a randomly generated tree. Parameters are population size, maximum number of generations and probabilities for genetic operators. The evolutionary process is as follows. At the beginning, a population of tree-based programs is randomly generated. Then, the new population is created by applying genetic operators to the individuals chosen from the existing population based on their fitness. This process is repeated until the desired criteria are satisfied or the number of generations exceeds the maximum number of generation. GP has been used successfully in generating computer programs for solving a number of problems in a wide range of areas [7].

II.2 Grammar Guided Genetic Programming

Grammar guided genetic programming systems are genetic programming systems that use grammars to set syntactical constraints on programs. The use of grammars also helps these genetic programming systems to overcome the closure requirement in canonical genetic programming, which cannot always be fulfilled [7].

Using grammars to set syntactical constraints was first introduced by Whigham [15] where context-free grammars were used. We shall refer Whigham's system as GGGP for the rest of the paper. Basically, GGGP has the same components and operations as in GP; however, there are a number of significant differences between the two systems. In GGGP, a program is represented as its derivation tree in the context free grammar. Crossover

[1] The ideas of using tree-based representation of chromosomes and swapping sub-trees as crossover operator was first introduced in [2].

between two programs can only be carried out by swapping their two sub-derivation trees that start with the inner nodes labelled by the same non-terminal symbol in the grammar. In mutation, a sub-derivation tree is replaced by a randomly generated sub-derivation tree that is derived from the same non-terminal symbol. GGGP demonstrated positive results on the 6-multiplexer problem and subsequently on a wide range of other problems.

II. 3 Tree Adjunct Grammar Guided Genetic Programming

Tree adjunct grammar guided genetic programming [3] (TAG3P) uses tree adjunct grammars along with context free grammars to set syntactical constraints as well as search bias for the evolution of programs. In this subsection we will first give the basic concepts of tree adjunct grammars then the basic components of TAG3P.

II.3.1 *Tree Adjunct Grammars*

Tree-adjunct grammars are tree-rewriting systems, defined in [5] as follows:

Definition 1: a tree-adjunct grammar comprises of 5-tuple (T, V, I, A, S), where T is a finite set of terminal symbols; V is a finite set of non-terminal symbols ($T \cap V = \varnothing$); $S \in V$ is a distinguished symbol called the start symbol. I is a set of trees called initial trees. An initial tree is defined as follows: the root node is S; all interior nodes are labelled by non-terminal symbols; each node on the frontier is labelled by a terminal symbol. A is a finite set of trees called auxiliary trees, which can be defined as follows: internal nodes are labelled by non-terminal symbols; a node on the frontier is labelled by a terminal or non-terminal symbol; there is a special non-terminal node on the frontier called the foot node. The foot node must be labelled by the same (non-terminal) symbol as the root node of the tree. We will follow the convention in [6] to mark the foot node with an asterisk (*).

The trees in $E= I \cup A$ are called elementary trees. Initial trees and auxiliary trees are denoted α and β respectively; and a node labelled by a non-terminal (resp. terminal) symbol is sometime called a non-terminal (resp. terminal) node. An elementary tree is called X-type if its root is labelled by the non-terminal symbol X.

The key operation used with tree-adjunct grammars is the adjunction of trees. Adjunction can build a new (derived) tree γ from an auxiliary tree β and a tree α (initial, auxiliary or derived). If a tree α has a non-terminal node labelled A, and β is an A-type tree then the adjunction of β into α to produce γ is as follows. Firstly, the sub-tree α_1 rooted at A is temporarily disconnected from α. Next, β is attached to α to replace this sub-tree. Finally, α_1 is attached back to the foot node of β. γ is the final derived tree achieved from this process. Adjunction is illustrated in Figure 1.

The tree set of a TAG can be defined as follows [5]:

T_G = {all tree t / t is completed and t is derived from some initial trees}

A tree t is completed, if t is an initial tree or all of the leaf nodes of t are non-terminal nodes; and a tree t is said to be derived from a TAG G if and only if t results from an adjunction sequence (the derivation sequence) of the form: $\alpha\ \beta_1(a_1)\ \beta_2(a_2)...\ \beta_n(a_n)$, where n is an arbitrary integer, α , β_i (i=1,2..n) are initial and auxiliary trees of G and a_i (i=1,2..n) are node address where adjunctions take place. An adjunction sequence may be denoted as (*). The language L_G generated by a TAG is then defined as the set of yields of all trees in T_G.

$L_G = \{w \in T^* / w$ is the yield of some tree $t \in T_G\}$

The set of languages generated by TAGs (called TAL) is a superset of context-free languages; and is properly included in indexed languages [6]. More properties of TAL can be found in [6]. One special class of tree-adjunct grammars (TAGs) is lexicalized tree-adjunct grammars (LTAG) where each elementary tree of a LTAG must have at least one terminal node. It has been proved that for any context-free grammar G, there exists a LTAG G_{lex} that generates the same language and tree set with G (G_{lex} is then said to strongly lexicalize G) [6].

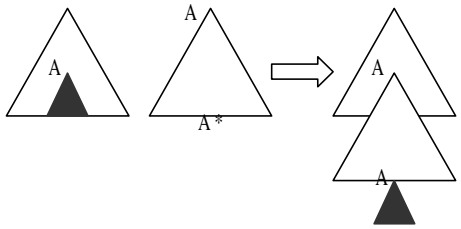

Fig.1 Adjunction.

II.3.2 *Tree Adjunct Grammar Guided Genetic Programming*

In [3], we proposed a grammar guided genetic programming system called TAG3P, which uses a pairs consisting of a context-free grammar G and its corresponding LTAG G_{lex} to guide the evolutionary process. The main idea of TAG3P is to evolve the derivation sequence in G_{lex} (genotype) rather than evolve the derivation tree in G as in [15]. Therefore, it creates a genotype-to-phenotype map. As in GP [7], TAG3P comprises of the following five main components:

Program representation: a modified version of the linear derivation sequence (*), but the adjoining address of the tree β_i is in the tree β_{i-1}. Thus, the genome structure in TAG3P is linear and length-variant. Although the language and the tree set generated by LTAGs with the modified derivation sequence is yet to be determined, we have found pairs of G and G_{lex} conforming to that derivation form for a number of standard problems in genetic programming [4].

Initialization procedure: a procedure for initializing a population is given in [3]. To initialize an individual, TAG3P starts with selecting a length at random; next, it picks up randomly an α tree of G_{lex} then a random

sequence of β trees and adjoining addresses. It has been proved that this procedure can always generate legal genomes of arbitrary and finite lengths [3].

Fitness Evaluation: the same as in canonical genetic programming [3].

Genetic operators: in [3], we proposed two types of crossover operators, namely one-point and two-point crossover, and three mutation operators, which are replacement, insertion and deletion. The crossover operators in TAG3P are similar to those in genetic algorithms; however, the crossover point(s) is chosen carefully so that only legal genomes are produced. In replacement, a gene is picked up at random and the adjoining address of that gene is replaced by another adjoining address (adjoining address replacement); or, the gene itself is replaced by a compatible gene (gene replacement) so that the resultant genome is still valid. In insertion and deletion, a gene is inserted into or deleted from the genome respectively. With these carefully designed operators, TAG3P is guaranteed to produce only legal genomes. Selection in TAG3P is similar to canonical genetic programming and other grammar-guided genetic programming systems. Currently, reproduction is not employed by TAG3P.

Parameters: minimum length of genomes, MIN_LENGTH, maximum length of genomes MAX_LENGTH, size of population - POP_SIZE, maximum number of generations – MAX_GEN and probabilities for genetic operators.

Some analysis of the advantages of TAG3P can be found in [3, 4].

II.4 Other Grammar Guided Genetic Programming Systems

Wong and Leung [16] used logic grammars to combine inductive logic programming and genetic programming. They have succeeded in incorporating domain knowledge into logic grammars to guide the evolutionary process of logic programs.

Ryan and his co-workers [14] proposed a system called grammatical evolution (GE), which can evolve programs in any language, provided that this language can be described by a context-free grammar. Their system differs from Whigham's system in that it does not evolve derivation trees directly. Instead, genomes in GE are binary strings representing eight-bit numbers; each number is used to make the choice of the production rule for the non-terminal symbol being processed. GE has been shown to outperform canonical GP on a number of problems [14].

II.4 Building Blocks in Genetic Programming Systems

GP building blocks are low order and compact schemas that is above the average observed performance and expected of appearing at an exponential rates in future generation [11]. In [11] the GP hypothesis building blocks was stated as the combination of the low order, compact, and highly fit schemata to make even better schemata. To date there has been several ways of defining building blocks (or schemata) in genetic programming systems [12]. In [4], we propose the concept of building blocks for TAG3P as trunks of beta trees in the chromosome. TAG3P has been proved to work very efficiently in combining and replicating building blocks [4].

III. Even Parity Problems

The even parity problem is the symbolic regression problem on boolean domain where the target function is the even-n-parity function. The even-n-parity function is the boolean function of n binary variables; it returns true when the number of 1-input bits is even and return false otherwise.

In the literature, this function is believed to be the hardest boolean function to learn because the solutions are very sparse in the search space and they become exponentially sparser when n is increased [7, 9].

The grammar G and the tree adjunct grammar G_{lex} for the problem is as follow:

G=(T, V, P, {EXP}) where T={X_1, X_2,.., X_n, AND, OR, NAND, NOR}, V={EXP, OP, VAR}, and P={EXP→EXP OP EXP, OP→AND, OP→OR, OP→NAND, OP→NOR, EXP→VAR, VAR→X_1, VAR→X_2,..., VAR→X_n}.

G_{lex}= (T,V, I, A, {EXP}) where T and V are the same as in G; I and A are shown in Figure 2.

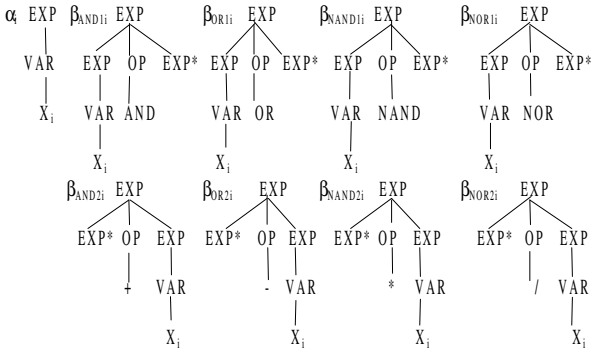

Fig. 2 Elementary trees for G_{lex} with i=1,..,n.

IV. Experimental Design

We experiment TAG3P and GGGP on the even parity problems with the number of input variables are 3, 4, and 5. The results for GP on these problems are taken from [7]. Table 1 summarises our experiment setups. We tried to set the parameters as same as in [7] to provide a fair basis for comparison.

V. Results and Discussions

For each problem, we conducted 100 runs (50 for TAG3P and 50 for GGGP). For GP, we used the results in [7]. The overall results are summarised in table 2. Although the probability of success for GP on even-5-parity problem was not given in [7], in [8] Koza stated that, on average, GP needs up to

6,528,000 processed individuals to yield a solution with a probability of 99%. We experiment GP on the even-5-parity problem and the number of solutions found were zero. The cumulative frequencies of GGP and TAG3P on the even-3and 4-parity problems are depicted in Figure 3 and 4.

Objective	Induce the even-n-parity function from its data.
Terminal Operands	$X_1, X_2,..., X_n$
Terminal Operators	AND, OR, NAND, NOR
Fitness Cases	2^n cases of the function.
Raw fitness	The number of correct classification
Standardized Fitness	The number of misclassification.
Hits	Same as raw fitness
Genetic Operators	Tournament selection, crossover, and mutation.
Parameters	Population size= 4000, crossover rate=0.9, mutation rate=0.1, tournament size=3, number of generations=50.
Success predicate	An individual scores 2^n hits

Table1. The experiment setup.

	Even-3	Even-4	Even-5
TAG3P	26 (52 %)	0 (0%)	0 (0%)
GGGP	45 (90 %)	16 (38%)	0 (0%)
GP	50 (100 %)	24 (48 %)	0 (0%)

Table2. The probability of success for three systems on even-3, 4, and 5-parity problems.

The results show that TAG3P is not comparable to GGGP and GP on the problems. Even for the unsuccessful runs, the average standardized fitness of the best individuals in GGGP is smaller than in TAG3P.

One of the possible explanations for the poor performance of TAG3P on the problems is that the nature of the search space is not suitable for the promotion of building blocks in TAG3P. In [9], the search space of the problem was analysed; the solutions are very sparse. It is like a needle in a haystack and not suitable for progressive search techniques. The more a system near to random search the better it can cope with the problems.

Moreover, in [16], it is known that the solutions in an even-n-parity problem have a recursive structure. For example, if EXP is the solution of an even-n-parity problem then AND (OR (X_{n+1}, EXP), NAND (X_{n+1}, EXP)) is the solution for the even-n+1-parity problem. Therefore, the number of operators (and the length) of the solution in even-n+1-parity problem should be more than twice of that in the even-n-parity problem[2]. In addition, the standardised fitness of EXP in the even-n+1-parity problem is only $2^n/2$, which is highly unfit.

[2] In our experiments and [7], the length of the solutions was always very long, e.g. it was more than or equal to 20 operators for even-3-parity problem.

Consequently, a genetic programming system needs to preserve and combine the long and unfit blocks of code in order to induce the even-n-parity function. In that sense it is contrary to the building blocks hypothesis.

In TAG3P the blocks of codes are the trunks of beta trees adjoined together. Since they are unfit, they will die out quickly in the course of evolution. Even when some survive and combine to make a longer trunk of beta trees the probability of being destroyed by crossover and mutation later will be bigger. In contrast, GP and GGGP seem to be better than TAG3P in preserving these unfit blocks of codes because they define the blocks of codes as sub-trees.

Aware of that problems, some approaches to packing and reuse the code have been proposed such as automatic define functions (ADF) [7, 8], auto defining module [13] to name but a few. It is interesting to see TAG3P with similar approach in the future.

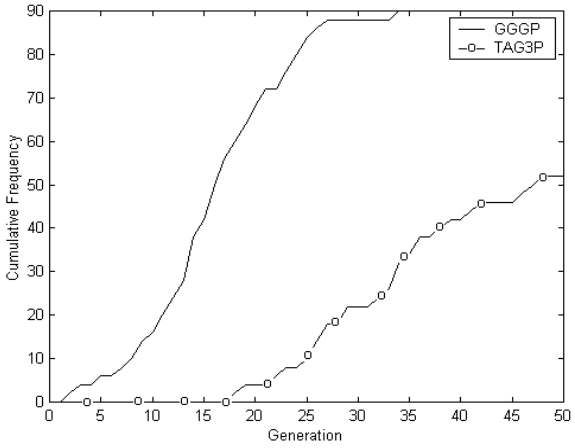

Fig. 3 Cumulative frequencies of TAG3P and GGGP on the even-3-parity problem.

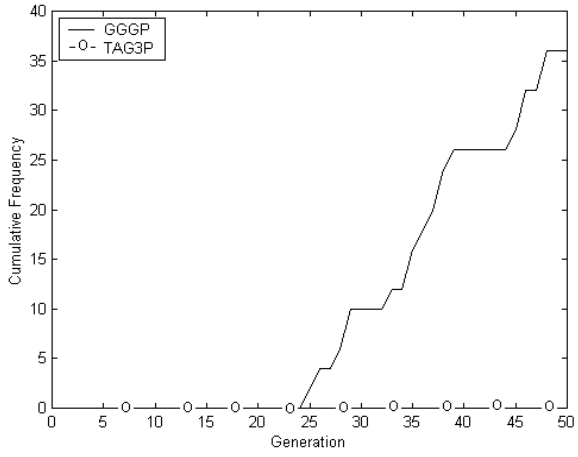

Fig. 4 Cumulative Frequencies of TAG3P and GGGP on the even-4-parity problem.

VI. Conclusion and Future Work

In this paper, we experiment TAG3P on the even parity problems and compare with GGGP and GP. The result show that TAG3P does not work well on the problems due to the nature of the search space and the structure of the solution, which requires the preservation and combination of unfit blocks of codes.

In future, we will investigate further the effects of genetic operators in TAG3P on the problem. As mentioned in the previous section, a mechanism of automatic finding, packing and reusing the blocks of codes for TAG3P will be studied.

References

[1] W. Banzhaf, P. Nordin, R.E. Keller, and F.D. Francone, Genetic Programming: An Introduction, Morgan Kaufmann Pub, 1998.

[2] N. L. Cramer, "A representation for the Adaptive Generation of Simple Sequential Programs', Proceedings of an International Conference on Genetic Algorithms and the Applications, pp. 183 – 187, Lawrence Erlbaum Associates, July 1985.

[3] N.X. Hoai and R.I. McKay, "A Framework for Tree Adjunct Grammar Guided Genetic Programming", Proceedings of the Post-graduate ADFA Conference on Computer Science (PACCS'01), pp. 93-99, 2001.

[4] N.X. Hoai, R.I. McKay, D. Essam, and R. Chau, "Solving Symbolic Regression Problem with Tree-Adjunct Grammar Guided Genetic Programming: The Comparative Results", to Appear in the Proceedings of IEEE Congress on Evolutionary Computation (CEC2002), Hawai, USA, 2002.

[5] A.K. Joshi, L.S. Levy, and M. Takahashi, "Tree Adjunct Grammars", *Journal of Computer and System Sciences*, Vol. 10:1, pp. 136-163, 1975.

[6] A.K. Joshi and Y. Schabes, "Tree Adjoining Grammars", Handbook of Formal Languages, Springer-Verlag, pp. 69-123, 1997.

[7] J. Koza, Genetic Programming, The MIT Press, 1992.

[8] J. Koza, Genetic Programming II: Automatic Discovery of Reusable Programs., The MIT Press, 1994.

[9] B. Langdon, "Why "Building Blocks" Don't Work on Parity Problems", Tecnical Report CSRP-98-17, The University of Birmingham, 1998.

[10] M. O'Neill and C. Ryan, "Grammatical Evolution: A Steady State Approach", Proceedings of the Second International Workshop on Frontiers in Evolutionary Algorithms, pp. 419-423, 1998.

[11] U. M. O'Reilly and F. Oppacher, The Troubling Aspects of a Building Block Hypothesis for Genetic Programming, Foundation of Genetic Algorithms 3, Morgan Kaufmann, 1995, pp. 73-88,.

[12] R. Poli and N.F. McPhee, "Exact Schema Theory for GP and Variable Length Gas with Homologous Crossover", *GECCO*, San Fransisco, pp. 104-111, 2001.

[13] J.P. Rosca, and D.H. Ballard, "Genetic Programming with Adaptive Representations", Technical Report 489, The University of Rochester, Feb 1994.

[14] C. Ryan, J.J. Collin, M. O'Neill, "Grammatical Evolution: Evolving Programs for an Arbitrary Language", Lecture Note in Computer Science 1391, Proceedings of the First European Workshop on Genetic Programming, Springer-Verlag, pp. 83-95, 1998.

[15] P. Whigham, "Grammatically-based Genetic Programming", Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications, Morgan Kaufmann Pub pp. 33-41, 1995.

[16] M.L. Wong and K.S. Leung, "Evolving Recursive Functions for Even-Parity Problem Using Genetic Programming", Advances in Genetic Programming, The MIT Press, pp. 221-240, 1996.