ORDERTREE: A New Test Problem for Genetic Programming

Tuan-Hao Hoang
School of IT & EE
University of New South Wales @ Australian Defence Force Academy, Northcott drv, Canberra ACT 2600,Australia
t.hao@adfa.edu.au

Nguyen Xuan Hoai
School of IT, Vietnamese Military Technical Academy,100 Hoang Quoc Viet St, Hanoi, Vietnam
nxhoai@gmail.com

Nguyen Thi Hien
School of IT, Vietnamese Military Technical Academy,100 Hoang Quoc Viet St, Hanoi, Vietnam
hien_cpqn@yahoo.com

RI McKay
Structural Complexity Laboratory, School of Computer Science & Engineering, Seoul National University, South Korea
rim@cse.snu.ac.kr

Daryl Essam
School of IT & EE
University of New South Wales @ ADFA, Northcott drv, Canberra ACT 2600, Australia
daryl@cs.adfa.edu.au

ABSTRACT

In this paper, we describe a new test problem for genetic programming (GP), ORDERTREE. We argue that it is a natural analogue of ONEMAX, a popular GA test problem, and that it also avoids some of the known weaknesses of other benchmark problems for Genetic Programming. Through experiments, we show that the difficulty of the problem can be tuned not only by increasing the size of the problem, but also by increasing the non-linearity in the fitness structure.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search – *graph and tree search strategies.*

General Terms

Theory, Performance, Languages

Keywords

Genetic Programming, Benchmark Problems, Problem Difficulty.

1. INTRODUCTION

Since its introduction [3, 13, 23], Genetic Programming (GP), has gradually become a mature field with many applications. As the number of different systems increases, there is a need to design benchmark problems providing a common ground for comparing their relative robustness. These problems (often simplified versions of real-world problems) can also help to understand the behavior and theoretical aspects of genetic programming.

In this paper, we describe a new test problem for GP, ORDERTREE. We argue that it is a natural analogue of ONEMAX, a popular Genetic Algorithm (GA) test problem, and that it avoids some of the known weaknesses of other benchmark problems for GP. Through experiments, we show that the problem difficulty can be tuned both by increasing the size of the problem, and by increasing the non-linearity in the fitness structure.

The paper is organized as follows. Section 2 briefly discusses the related work on designing test problems for GA and GP. In section 3, we detail our ORDERTREE problem, and discuss its properties relative to those of other test problems in GP. Section 4 contains some experiments with the proposed problem, and discussion of the results. The paper concludes with section 5, where we discuss some possibilities for extending this work.

2. RELATED WORK

In the field of genetic algorithms (GA), commonly regarded as the predecessor of GP, researchers have proposed a number of benchmark problems. These problems are intended to illuminate how different GA systems work, by comparing their performance. The practice was first initiated in Dejong's thesis with his proposed set of test functions [7]. Since then, a range of test problems have been proposed by GA researchers. Goldberg [9] developed the idea of deceptive functions from the schema processing perspective, where the average fitness of 'deceptive' schemata for these functions (subspaces that actually do not contain the optima of the functions) is higher than the fitness of schemata that would lead to the actual solutions. In [22], Schaffer and Eshelman proposed the ONEMAX problem, using the idea of 'unitation' on binary representation. The value of the unitation function on a binary vector x is the number of 1-bits in x (denoted by u(x)). The optimal solution of the (maximization) ONEMAX problem is the vector with all bits assigned as 1. The primary advantage of the unitation function is that it is easy to describe the fitness equivalence classes (solutions with the same number of bits set to 1), greatly facilitating analysis [20]. In [6], unitation was used to design trap functions as test beds for GA. The designers specified a mapping v from the unitation function's range [0,l] (l being the length of the vector x) to [0,1]. They chose a value $z \in [0,l]$ as the trap location, defining $v(0) = v(l) = 1$, and $v(z) = 0$; thus the composition of u and v defines a trap function, with the minimum being the equivalence class of binary vectors x with $u(x)=z$, and maxima at $u(x)=0$ and $u(x) = l$. Mitchell et al. [16] devised 'Royal Road' functions, aiming to test the building block hypothesis. The main idea is to construct a function from disjoint schemata. A variant of the Royal Road function was the 'Hierarchical-IFF' (HIFF) functions, constructed by Watson et al. [24] to model the dependencies between different schemata.

As in GA, several challenging test problems have been proposed within the genetic programming literature to develop and test new GP systems and/or theories. In the early days, the set of problems presented in [13], such as symbolic regression, artificial ants, multiplexers, and n-parity, were usually used when GP researchers compared the performance of different GP systems, or wanted to investigate theoretical aspects of genetic programming (e.g code bloat phenomena). However, as problem test beds, these problems are imperfect choices.

Some lack an important characteristic of good test beds, namely tunable difficulty. Tunability is important, since it gives an indication of how a GP system scales with problem difficulty

In others (e.g. n-parity), problem difficulty arises primarily from the extremely low density of high-fitness solutions in the search space (needle in the haystack) [15]. Such problems are intrinsically difficult for all search methods (recognizing that change of representation may ameliorate the difficulty - e.g for the n-parity problem, Automatically Defined Functions (ADFs) could help to acquire new boolean functions, such as EQ and XOR, significantly increasing the density of solutions in the search space [14]). The problems thus violate the principle of 'bounded difficulty' used in designing GA test problems [9].

For the purpose of understanding the behavior and theoretical aspects of GP, the problems in [13] are generally difficult to analyze For instance, it is often difficult to detect all introns (non fitness-contributing parts), which means that the study of introns and code bloat on those problems can only be approximate. Some other practical difficulties, such as numeric overflow, add further complication to their use in understanding GP.

In [19], Punch et al. proposed the Reoyal Tree problem, a natural extension of the GA Royal Road problem in GA, as a GP benchmark. The function set consists of alphabetic characters 'A','B','C',... with increasing arity (i.e nodes labeled with 'A' should have 1 child, with 'B' should have 2 children, and so on); the terminal set contains only one character 'X'. The fitness of a program is calculated based on the positions of nodes, and the optimal solution is the tree with all nodes positioned correctly. Figure 1 depicts the optimal solution for the Royal Tree problem with a problem size of 3 ('A', 'B', 'C').

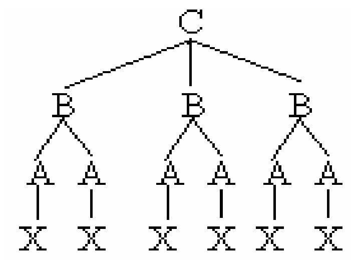

Figure 1. Optimal solution for Royal Tree problem of size 3.

The Royal Tree problem is difficulty-tunable by increasing the size of the problem (and therefore the size of the solution). However, the optimal solution is unique, and highly specific in shape – i.e it must be the fullest tree. As pointed out in [4], fullest trees are among the most difficult tree structures to find by genetic programming, regardless of the contents of tree nodes. Thus the problem is atypical of the class of problems which GP may be expected to solve effectively.

In [8], Gathercole and Ross introduced the MAX problem to GP, to highlight some deficiencies in the standard (subtree) crossover operator, resulting from the tree (program) depth limit imposed in most practical GP systems. The function set for the MAX problem consists of arithmetic multiplication (×) and addition (+), and the terminal set is the numeric value 0.5. The task required of GP is to find the tree with maximal value, subject to the constraint that depth cannot exceed D (maximal depth). Figure 2 shows an example solution for D=4.

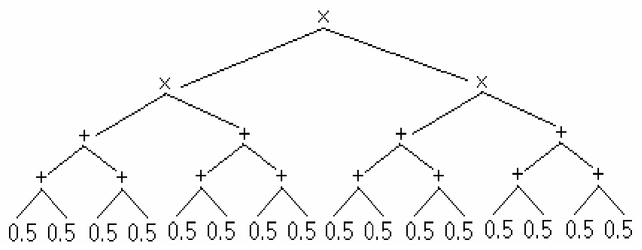

Figure 2. The optimal solution for MAX with D=4.

Good solutions for the MAX problem have + nodes deep in the tree (to build values greater than 1) so that multiplication may be applied effectively. Once the value of the subtrees exceeds 1 (2 being the simplest value to construct), they should be combined by × to maximize the overall value (since multiplication increases faster than addition once the operands exceed 1). The problem becomes very difficult for GP (with increasing D) because × and + which appear in the wrong places are difficult to move with standard crossover without violating the depth bound (D). A comprehensive analysis of GP on the MAX problem can be found in [15] (chapter 10).

The MAX problem is a difficulty-tunable problem for GP. MAX also, unlike the Royal Tree problem, has multiple solutions for each D. For instance, a variant of the optimal solution shown in Figure 2 could replace some depth-1 nodes × with + (for D-3), since 2×2=2+2. However, these multiple solutions become exponentially rarer (as a proportion of the solution space) as D increases, so the advantage over Royal Tree is limited. Otherwise, MAX shares the same weakness as the Royal Tree problem, in that the shape of optimal solutions is again the fullest tree.

In [11], Goldberg and O'Reilly devised two difficulty-tunable problems for GP, ORDER and MAJORITY, to model the dependencies between different components within a program tree. The function set of these two problems contains only a binary term called JOINT, while the terminal set has two subsets: positive - P_1,P_2,...P_n, and negative - N_1,N_2,...N_n. In the ORDER problem, a leaf is 'expressed' if it is labeled with P_i (i=1,..n), and there is no node before that node (taking the preorder traversal of the tree) labeled with N_i. In the MAJORITY problem, a leaf is 'expressed' if it is labeled with a P_i (i=1,...n) and the number of P_i contained in the (program) tree is larger than the number of N_i. The task for GP on ORDER and MAJORITY is to find trees with the maximal number of different expressed leaves (in fact, n). In [18], the authors used the problems to investigate how different fitness structures affect the performance of GP. Sastry et al. used the ORDER problem to analyse population sizing in GP [21]. Figures 3 and 4 depict two of many possible optimal solutions for ORDER and MAJORITY respectively (with n=3).
It was noted in [11] that the ORDER and MAJORITY problems resemble the GA ONEMAX problem, in that the fitness contribution of each expressed leaf to the fitness of the whole tree resembles the contribution of 1-bits in unitation. It was also argued that the problems share some aspects with the real

problems usually used in the GP literature. They are difficulty tunable (by changing problem size n), there are multiple optimal solutions, and introns may appear at any place in a program tree. This reduces the shape bias of optimal solutions compared with earlier test problems such as Royal Tree and MAX. However these problems differ in one important characteristic of GP problems, in which the content of internal nodes is important in determining fitness, and the dependencies between sub-tree components. In this respect, it is arguable that they resemble typical GA problems more than GP problems. For instance, if we have an arithmetic tree such as shown in Figure 5 where the left branch evaluates as zero, the content of the root node even determines whether the right part is an intron. If the content of the root node is multiplication, then the right part is an intron, while it is effective code if the content of the root node is addition.

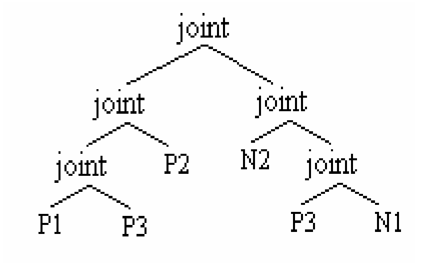

Figure 3. An optimal solution for ORDER with n=3.

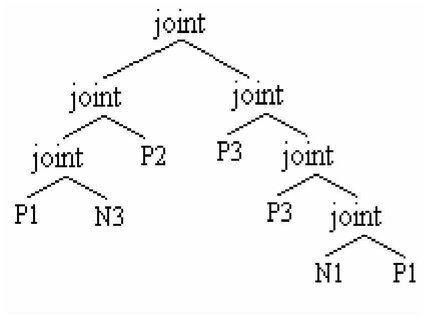

Figure 4. An optimal solution for MAJORITY with n=3.

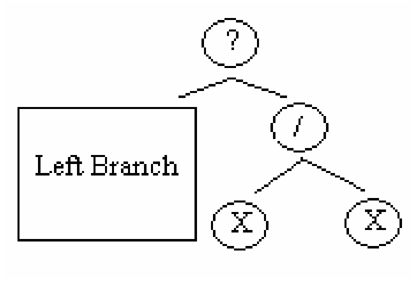

Figure 5. Content of an internal node is important.

Daida et al. [4] investigated the binomial-3 problem (symbolic regression problem with the target function $(1+x)^3$). They found that by varying the range of the ephemeral random constants, the problem becomes tunably difficult for GP. Subsequently, by observing the evolution of program tree shapes of GP on the problem, they discovered that structure alone could pose great difficulties to GP. To validate this hypothesis, they subsequently designed a test problem for GP called LID [5]. The function set for the LID problem includes only one binary term, “joint”, which plays a similar role to “joint” in the ORDER and MAJORITY problems described above, and the terminal set contains only one terminal, LEAF. The fitness of a program tree is measured by how close in shape (using a shape metric defined in [5]) it is to the target program tree. By experimenting with the LID problem, they were able to demonstrate the inherent problem of structural difficulty in GP. In particular, they found a separation of the tree shape space into three areas. In the first, it is easy for GP to find solutions. In the second, GP finds solutions only with great difficulty. In the last, including fuller and thinner trees, it is almost impossible for GP to find any solutions.

The problem of structural difficulty is important to GP for a number of reasons. When solving problems whose simplest solutions have shapes in the second or third areas, GP may have to accumulate introns in order to find equivalent solutions that lie in the first area. This may provide one reason why introns emerge in GP systems, and why GP usually obtains long and complex solutions even when short and simple solutions exist. Moreover, it also means that it is possible that, for some of the test problems proposed in GP literature, the source of difficulties not only come from the claimed characteristics of the problems (such as the increase of problem/solution size/depth) but also from the biased shape of the optimal solution. Royal Tree and MAX problems are two examples of such problems. The test problem proposed in this paper (detailed in the next section) was designed with the awareness of Daida’s problem of structural difficulty and so attempts to remove the shape bias in the optimal solutions.

While the above are the best-known test problems, a number of other difficulty-tunable test problems have been proposed. Burke et al. devised the N-prisoner puzzle based on the N-hat game [2]. Nguyen et al. [17] used simple symbolic regression problems, in which the target functions are a family of polynomial functions of increasing order, to compare some different GP and grammar-guided GP systems. In [12], Gustafson et al. also used simple symbolic regression, in which the target functions are a family of randomly generated polynomials functions with increasing order, to derive some analysis for GP runs.

3. PROBLEM DESCRIPTION

The design of the ORDERTREE problem was inspired by the ONEMAX problem in GA, and some of the test problems described in the previous section, particularly the MAX, ORDER, and MAJORITY problems. In this section, we first describe a simple version of the problem which is tunably difficult, but still has the weakness that the optimal solutions are relatively fixed and biased in shape (fullest tree). Next, we amend that deficiency by introducing the ‘left-neutral- walk’ into the fitness calculation. We then discuss how the ORDERTREE problem relates to test problems mentioned in the previous section.

3.1 The restricted version of ORDERTREE

The function set for ORDERTREE of size n is defined as follows:

F= {‘0’,’1’,’2’,....’n-1’}

The function nodes are labeled with numbers from the set {0,1,2,...n-1}. All the functions are of arity 2 (i.e each function has two arguments). The terminal set is defined as:

T={‘0’,’1’,’2’,....’n-1’}

The terminal nodes are also labeled with numbers from the set {1,2,...n-1}. Algorithm A below calculates the fitness for a tree.

Algorithm A. *Calculation of fitness.*

```
Input: A program tree t for ORDER problem
Output: Fitness of t.
Global variable F to contain the total
fitness.

1. Procedure NodeCal (p :node){
2.       l = p->LeftChild;
3.       r = p->RightChild;
4.   if (VALUE(p) < VALUE(l)){
5.        F++;
6.       NodeCal(l);
7.    }
8.   if (VALUE(p)<VALUE(r)) {
9.      F++;
10.      NodeCal(r);
11.   }
12. }
The main program:
1. F=0;
2. Return NodeCal(root node of t);
```

The fitness of a program tree is calculated in top-down fashion. A node contributes one unit to the total fitness of the tree if its content (numeric value of the label) is bigger than its parent node content, and if its parent node is also fitness-contributing (by default, the root node is always fitness-contributing). The function VALUE in algorithm A returns the numeric value of the label of each node (e.g VALUE('1') returns 1). Figure 6 shows some program trees, together with their fitness for the ORDERTREE problem of size 4. Broken lines indicate the parts of the tree which are not fitness-contributing (introns).

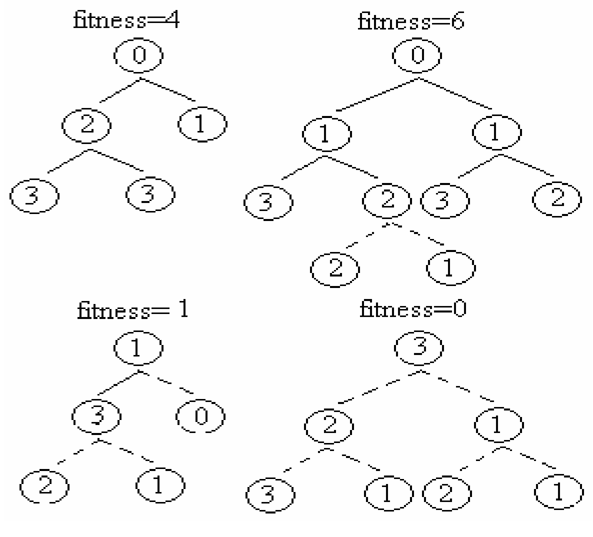

Figure 6. Some examples of individuals and their fitness values for ORDERTREE (n=3).

The task for GP is to find solutions (individuals) with maximal fitness. One trivial optimal solution for ORDERTREE problem is the fullest tree of depth n where node at depth D (D < n) is labeled with 'D', and the (maximal) fitness is $2^n - 2$. Figure 7 shows this optimal solution when n=4.

The only variants of the optimal solution shown in Figure 7 are deeper trees, having the top part identical with the optimal tree (as in Figure 7) and introns appearing under it (the contents of those nodes must have values smaller or equal to n). Figure 8 gives an example of such trees.

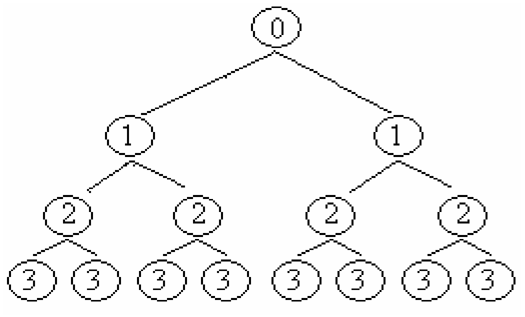

Figure 7. An individual with optimal fitness for ORDERTREE (n=4).

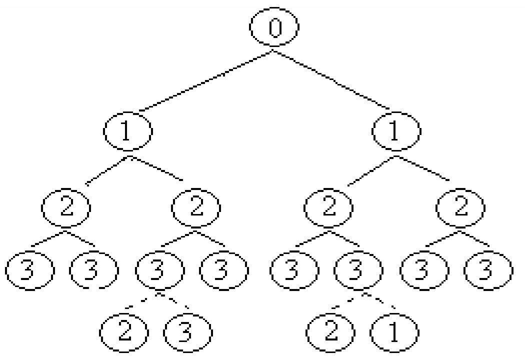

Figure 8. Individual with optimal fitness for ORDERTREE (n=4) the nodes under broken links are introns.

3.2 The complete version of ORDERTREE

As with the previously discussed problems, the shape restriction is an important weakness of the restricted version of ORDERTREE – it means that the problem is atypical of the class of problems which GP may be expected to solve readily [5]. To remove this bias in shape and positions of introns, we allow a 'left-neutral-walk' procedure in the process of fitness calculation. In the restricted version, the fitness calculation process terminates the calculation of a branch, if the node content is less than or equal to its parent node. In the full problem version, if the value of a node is equal to that of its parent node, the fitness calculation continues by visiting the left child. If that new node's value is less than its own parent, the process terminates as before, and no fitness contribution results, the whole subtree being treated as an intron. If the node value is greater, the subtree is evaluated, and the fitness contribution is passed to the parent. If the value is equal to its parent's, its left child is evaluated recursively. In all cases, the fitness contribution of the right child is zero, so that the right subtree acts as an intron. The process is detailed in Algorithm B.

For the complete version, an optimal solution need not be a full tree – it can be of a wide variety of shapes, with the introduction of introns at any depth of the tree due to the 'left-neutral-walk' procedure. Figure 9 depicts one such solution. We note that with the introduction of introns, the optimal solutions for ORDERTREE of size n do not necessary have depth n-1 (i.e it could be bigger). The 'left-neutral-walk' procedure is by no means the only way to solve the problem of bias in shape and positions of introns for the restricted version of ORDERTREE. Other possibilities are discussed, though not investigated, in the final section of the paper.

Algorithm B. *Calculation of fitness.*

```
Input: A program tree t for ORDER problem
Output: Fitness of t.
Global variable F to contain the total
fitness.

1. Procedure NodeCal (p :node){
2.       l is left child of p;
3.       r is the right child of p;
4.   if (VALUE(p) < VALUE(l)) {
5.        F++;
6.       NodeCal(l);
7.    }
7b.   q=l->leftchild;
8   if (VALUE(p)==VALUE(l)){
9.      FOUND = FALSE;
10.  while ((l>LeftChild!=NULL)  &&
(VALUE(l)==VALUE(q)) && ! FOUND){
12.  q=l->LeftChild;
13.  if (VALUE(q)>VALUE(p))
14.   FOUND=TRUE;
15. }
16. if (FOUND){
17.     F++;
18.     NodeCal(q);
19. }
20. if (VALUE(p)<VALUE(r)){
21.      F++;
22.      NodeCal(r);
23.  }
24.  if (VALUE(p)==VALUE(r)){
25.      FOUND = FALSE;
25b. q=r->Leftchild;
26.  while ((l->LeftChild!=NULL) &&
(VALUE(q)==VALUE(r)) ! FOUND){
27.  q=r->LeftChild;
28.  if (VALUE(q)>VALUE(p))
29.   FOUND=TRUE;
30. }
31. if (FOUND){
32.     F++;
33.     NodeCal(q);
34. }
35.}
The main program:
1. F=0;
2. Return NodeCal(root node of t);
```

3.3 Discussions of ORDERTREE problem

The ORDERTREE problem shares a number of common properties with some of the benchmark problems described in the previous section. As with ONEMAX, it uses the idea of unitation, i.e each node used in the fitness calculation process contributes one unit to the total fitness. However, in the next section, we will also consider other kinds of fitness structures.

As with ORDER and MAJORITY, dependencies between the nodes within a tree are modeled in ORDERTREE. In this case, the dependency is the relationship between parent and child nodes. This relationship determines whether a node contributes to the total fitness of the individual (tree). This situation happens quite often in practice, and Figure 5 depicts one example of it. The primary difference between ORDERTREE and ORDER/ MAJORITY lies in the importance of structure information – it is ignored in the latter, but emphasized in the former.

The restricted version of ORDERTREE is similar to MAX in that the optimal solutions for both must be the full tree of maximal depth. Moreover, if the depth limit for GP is n-1, when n is the size of the problem, ORDERTREE would illustrate the same difficulty with standard subtree crossover as MAX. However, this difficulty is at least partially removed in the complete version of ORDERTREE.

Finally, like all the problems described in the previous section, the size of the ORDERTREE problem can be tuned, and in the next section, we show that it leads to increasing difficulty for GP.

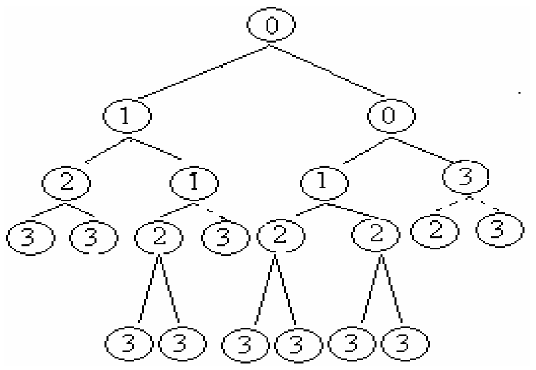

Figure 9. Individual with optimal fitness for ORDERTREE (n=3) the nodes under broken links are introns.

4. EXPERIMENTS & RESULTS

Two experiments were conducted with GP on the ORDERTREE problem. In the first experiment, GP was tried on ORDERTREE with problem sizes (n) 5, 6, 7, and 8 to investigate how GP copes with scalability in problem size. In the second experiment, different fitness structures were used to test how they affect the performance of GP. The parameter settings for GP in every run of the two experiments are summarized in Table 1. For each problem instance, the number of runs conducted was 100. The GP system used in all runs is an implementation of GP, as described in [13], with tree representation, subtree crossover, and subtree mutation.

4.1 Experiment 1

In this experiment, GP was tried on ORDERTREE with problem size n=5, 6, 7, 8. The proportions of success were 95%, 78%, 42%, 8% respectively. These results show that as the size (n) increases, the problem becomes increasingly difficult for GP. Figure 10 shows the average standardized fitness of the population over time. It can be seen that the GP population did converge in terms of fitness. The similar slopes of the curves suggest that the increase of the problem size just delayed the convergence of the GP population, without changing the speed of convergence. Figures 11-14 depict the fitness distribution of the GP population over time, where the z-axis is the number of individuals (over 500) that have the same fitness value (y-axis) at each generation (x-axis). It clearly shows that for n=5 or 6, the whole population converged very quickly to the optimal solutions. Figures 15-18 show how the average size and average effective size of the individuals in the GP population evolve over time. The effective size is the size of the individual minus the size of introns. All four figures depict the same pattern: while the effective size was relatively stable after 100 generations, the size of the whole individuals kept increasing very fast. In the end, introns account for almost 6/7 of individual trees on average. This is not surprising, in view of the general understanding of the prevalence of bloat in GP [1].

Table 1. Parameter settings for GP.

Objective	Find a tree with maximal fitness
Terminal Operands	0, 2, 3, ..., n-1 (with n=5,6,7,8)
Terminal Operators	The binary function symbols: 0, 2, 3, ..., n-1 (with n=5,6,7,8)
Raw fitness	The fitness of the tree calculated by Algorithm B
Standardized Fitness	Max Fitness – Raw Fitness.
Genetic Operators	Tournament selection (TOUR_SIZE=6), subtree crossover and subtree mutation.
Parameters	The operator probability: Crossover: 0.9; mutation: 0.1. Population Size =500, Max Generation =201. Max Depth =20, Max Initial Depth = 6.
Success predicate	An individual that has Max Fitness

4.2 Experiments 2

In the second series of experiments, similarly to [18], we investigated the effect of changing fitness structures on GP performance. In procedure NodeCal of algorithm B, each time the content of a node is bigger than the content of its parent node, the total fitness of the tree (variable F) is increased by one (lines 6, 12, 25, and 39). In other word, when a node is in the 'right place', it contributes one unit to the total fitness (this is denoted as fitness structure f1). In this experiment, we used three other ways of calculating the fitness contribution for each such node, namely, using the node value itself as the contribution (linear fitness structure – f2), using its square (square fitness structure – f3), and using its exponent base 3 (exponential fitness structure – f4). Specifically, in line 6 in algorithms B:

```
6.      F++;
```

is replaced by:

```
6.  F= F+ l;            /* linear – f2 */
```

Instead of adding one unit to the fitness, we add the node value

```
6.  F=F+ l*l;           /* square – f3 */
6. F=F+ 3^l;            /* exponential – f4 */
```

In moving from f1 to f4, we increase the non-linearity in the contribution of each 'correct' node to the fitness of the tree. In simple terms, this makes the problem more deceptive, because for these cases, GP is given an immediate reward for each larger value. However, if this is done too high in the tree, GP has a longer-term liability, as it will be difficult to create below it a large evaluated subtree, which would have instead been of greater worth.

Table 2 shows the results (proportion of success) of GP on ORDERTREE problem (with n=5, 6, 7, 8) with different fitness structures (f1, f2, f3, f4).

Table 2. Performance of GP on different fitness structures.

	n=5	**n=6**	**n=7**	**n=8**
f1	95%	78%	42%	8%
f2	87%	71%	29%	3%
f3	78%	66%	21%	7%
f4	72%	36%	12%	2%

From Table 2, it can be seen that, except for the case n=8, the increase of the non-linearity of the fitness structures significantly deteriorates GP performance. When n=8, ORDERTREE is very difficult for GP, and the effect of increasing size seems to override the effect of changing fitness structures.

5. CONCLUSION AND FUTURE WORK

In this paper, we have introduced ORDERTREE, a new test problem for GP based on ideas from the ONEMAX problem in GA. We argued that, while it shares a number of interesting properties with other known GP benchmark problems, it avoids some of their deficiencies. Through experiments, we showed that ORDERTREE has tunable difficulty. This difficulty can be tuned in at least two ways: by increasing the size of the problem (solution); and by decreasing the linearity in the fitness structures.

Future work will test alternative strategies (other than 'left-neutral-walk') for reducing the shape bias of optimal solutions in ORDERTREE. One alternative is 'max-value-branch', i.e instead of going to the left branch of a node that has content equal to its parent content, the search process could be carried out in both branches of the node, and the branch returning the bigger value for fitness contribution is selected (while the other becomes an intron).

In the longer term, we believe that the ORDERTREE problem can make a contribution to understanding a range of issues in GP, including rooted schemata theory [15] and population sizing [21].

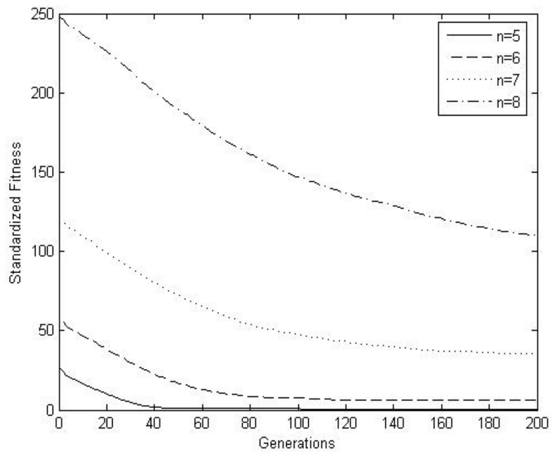

Figure 10. Average standardized fitness of the populations.

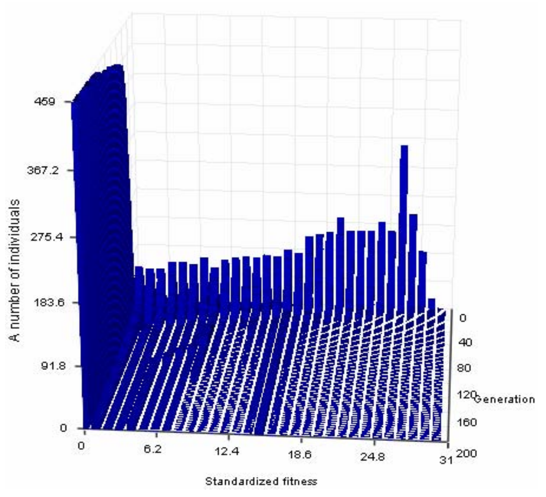

Figure 11. Fitness distribution n=5.

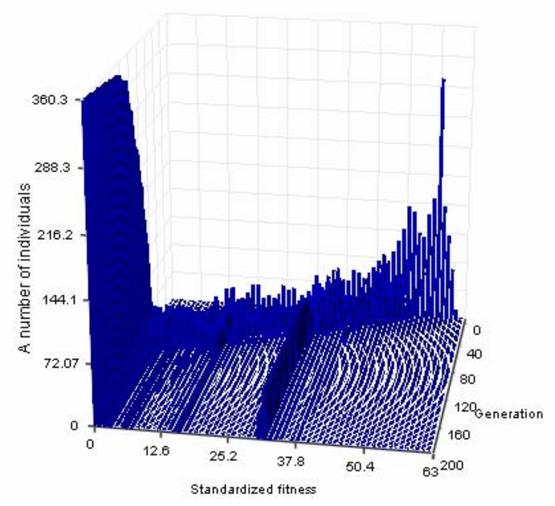

Figure 12. Fitness distribution n=6.

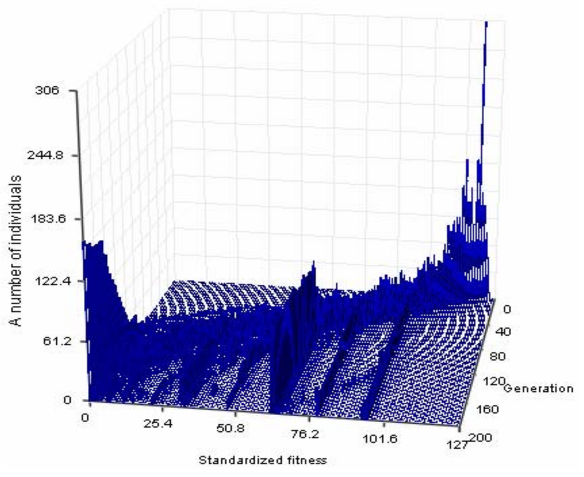

Figure 13. Fitness distribution n=7.

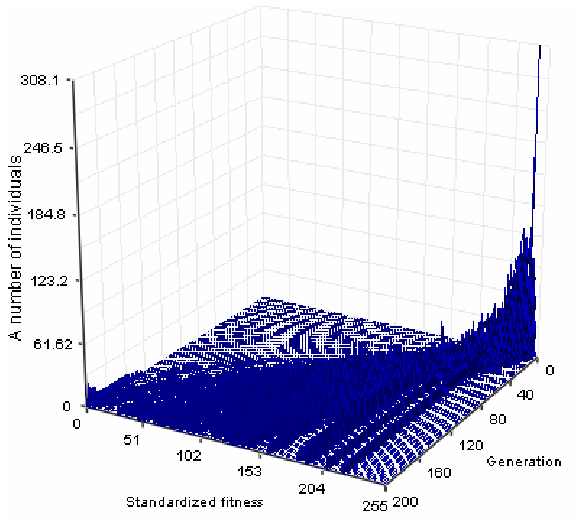

Figure 14. Fitness distribution n=8.

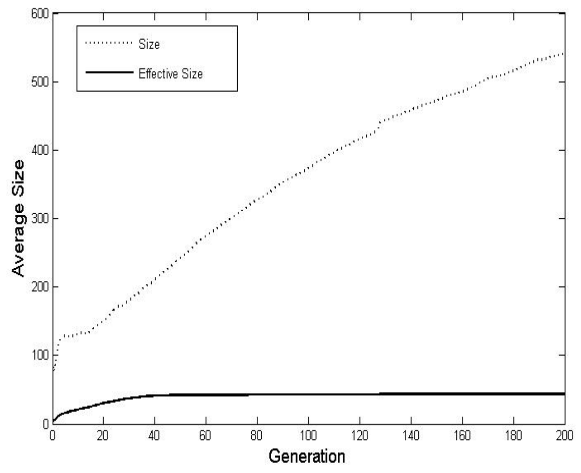

Figure 15. Evolution of size and effective size n=5.

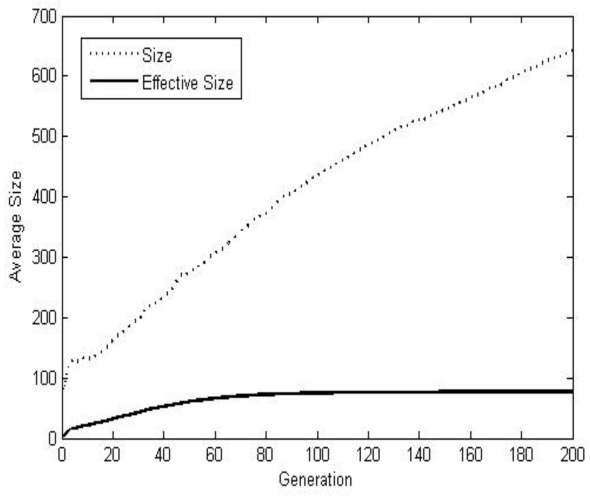

Figure 16. Evolution of size and effective size n=6.

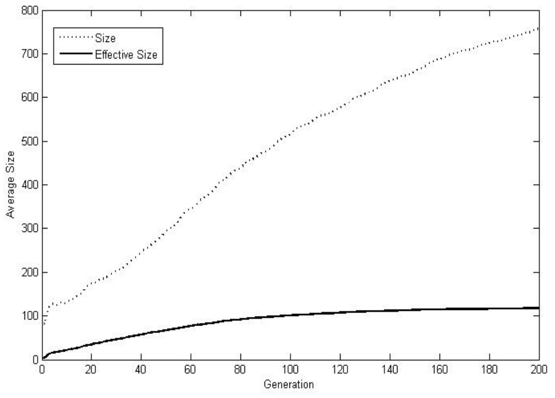

Figure 17. Evolution of size and effective size n=7.

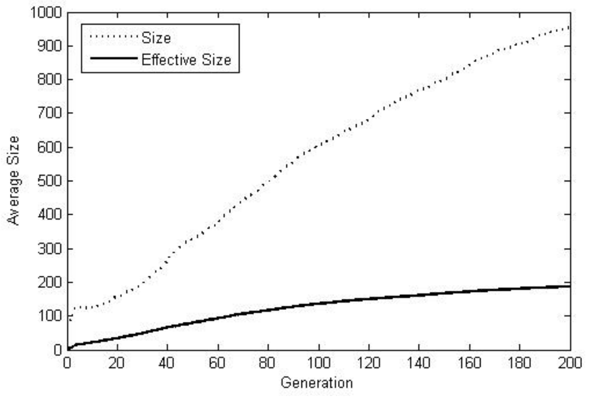

Figure 18. Evolution of size and effective size n=8.

6. REFERENCES

[1] Banzhaf, W.el al. *Genetic Programming: An Introduction*, Morgan Kaufmann, CA, 1998.

[2] Burke E., Gustafson S., and Kendall G., A Puzzle to Challenge Genetic Programming, in *Proceedings of the 5th European Conference in Genetic Programming*, LNCS 2278, Spinger-Verlag, Berlin, 2002, 238-247.

[3] Cramer, N.L. A Representation for the Adaptive Generation of Sequential Programs, *Proceedings of an International Conference on GA and the Applications*, 183-187, 1985.

[4] Daida J.M.et al. What Makes a Problem GP-Hard? Analysis of a Tunably Difficult Problem in Genetic Programming, *Genetic Programming and Evolvable Machines*,2001, 165-191.

[5] Daida J.M et al What Makes a Problem GP-Hard? Validating a Hypothesis of Structure Causes, Proceedings of Genetic Algorithms and Evolutionary Computation Conference (GECCO2003), LNCS 2724, Springer-Verlag 2003,1665-1677.

[6] Deb K. and Goldberg D.E. Analyzing Deception in Trap Functions, Foundations of Genetic Algorithms 2, Morgan Kaufmann, CA, 1993, 93-108.

[7] Dejong, K.A. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, Doctoral Dissertation, University of Michigan, Ann Arbor. Michigan, 1975.

[8] Gathercole C. and Ross P. An Adverse Interaction between Crossover and Restricted Tree Depth in Genetic Programming, in *Proceedings of the First Annual Conference on GP*, MIT Press, CA, 1996, 28-31.

[9] Goldberg, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning,* Addison-Wesley, MA, 1989.

[10] Goldberg, D.E. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms,* Kluwer Academic Publisher, 2002.

[11] Goldberg D.E. and O'Reilly U.M. Where Does the Good Stuff Go and Why?, In *Proceedings of The First European Conference on Genetic Programming (EuroGP)*, Springer-Verlag, 1998.

[12] Gustafson S. et al. Problem Difficulty and Code Growth in Genetic Programming, *Genetic Programming and Evolvable Machines*, 5, 2004, 271-290.

[13] Koza J.R. *Genetic Programming: On the Programming of Computers by Natural Selection*, MIT Press, MA, 1992.

[14] Koza J.R. *Genetic Programming II: Automatic Discovery of Reusabe Programs*, MIT Press, MA, 1994.

[15] Langdon W.B. and Poli R. *Foundations of Genetic Programming, Springer-Verlag*, Berlin, 2002.

[16] Mitchell M., Holland J.H., and Forrest S. When Will a Genetic Algorithm Outperform Hill Climbing?, In Advances in Neural Information Processing Systems 6, Morgan Kaufmann, CA, 1994.

[17] Nguyen X. H. et al. Solving the Symbolic Regression Problem with Tree Adjunct Grammar Guided Genetic Programming: The Comparative Result, In *Proceedings of Congress on Evolutionary Computation (CEC 2002)*, IEEE Press, 2002, 1326-1331.

[18] O'Reilly U.M. and Goldberg D.E. How Fitness Structure Affects Subsolution Acquisition in Genetic Programming, in *Proceedings of the Third Annual Conference on Genetic Programming*, Morgan Kaufmann, 1998, 269-277.

[19] Punch W.F., Zongker D., and Goodman E.D. The Royal Tree Problem, a Benchmark for Single and Multi-population Genetic Programming, in Advances in Genetic Programming 2, MIT Press, MA, 1996, 299-316.

[20] Reeves C.R. and Rowe J.E. *Genetic Algorithms: Principles and Perspectives*, Kluwer Academic Publisher, 2003.

[21] Sastry K., O'Reilly U.M., and Goldberg D.E. Population Sizing for Genetic Programming based on Decision Making, in *Genetic Programming Theory and Practice II*, Springer-Verlag, 2004, 49-65.

[22] Schaffer J.D. and Eshelman L.J. On crossover as an evolutionary viable strategy. In *Proceedings of the 4th International Conference on Genetic Algorithms,* Morgan Kaufmann, 1991, 61-68.

[23] Schmidhuber, J. *Evolutionary Principles in Self-Referential Learning*, Diploma Thesis, Technische Universitat, Muchen, 1987.

[24] Watson, R.A. Hornby G.S, and Pollack J.B. Modeling Building Blocks Dependency, in *Parallel Problem-Solving from Nature 5*, Springer-Verlag, Berlin, 97-106.