Genetic Programming, Probabilistic Incremental Program Evolution, and Scalability

Radovan Ondas[1], Martin Pelikan[2], and Kumara Sastry[3]

[1] Dept. of Math. and Computer Science, University of Missouri at St. Louis, CCB 331, 8001 Natural Bridge Rd., St. Louis, MO 63121, USA
ondasr@umsl.edu
[2] Dept. of Math. and Computer Science, University of Missouri at St. Louis, CCB 320, 8001 Natural Bridge Rd., St. Louis, M O 63121, USA
pelikan@cs.umsl.edu
[3] Dept. of General Engineering, University of Illinois at Urbana-Champaign, 117 TB, 104 S. Mathews Ave., Urbana, IL 61801, USA
ksastry@uiuc.edu

Summary. This paper discusses scalability of standard genetic programming (GP) and the probabilistic incremental program evolution (PIPE). To investigate the need for both effective mixing and linkage learning, two test problems are considered: `ORDER` problem, which is rather easy for any recombination-based GP, and `TRAP` or the deceptive trap problem, which requires the algorithm to learn interactions among subsets of terminals. The scalability results show that both GP and PIPE scale up polynomially with problem size on the simple `ORDER` problem, but they both scale up exponentially on the deceptive problem. This indicates that while standard recombination is sufficient when no interactions need to be considered, for some problems linkage learning is necessary. These results are in agreement with the lessons learned in the domain of binary-string genetic algorithms (GAs). Furthermore, the paper investigates the effects of introducing unnecessary and irrelevant primitives on the performance of GP and PIPE.

Key words: Genetic Programming, PIPE, scalability, order problem, trap problem

1 Introduction

To solve large and complex problems, scalability is among the primary concerns of an optimization practitioner. However, only few studies [14, 15] exist that study scalability in genetic programming (GP) [7]. The same holds for simple approaches to using probabilistic recombination in GP within the es-

timation of distribution algorithm (EDA) framework [8, 9, 11], such as the probabilistic incremental program evolution (PIPE) [13].

The purpose of this paper is to study the scalability of standard GP and PIPE on two decomposable GP problems: `ORDER` and `TRAP` [14]. The two algorithms perform as expected and they solve `ORDER` scalably while failing to scale up on `TRAP`. Additionally, the paper studies the effects of introducing unnecessary and irrelevant primitives. Both GP and PIPE are shown to deal with these two sources of difficulty well. The results presented in this paper confirm that binary-string genetic algorithms (GAs) have a lot in common with GP and PIPE, and thus the lessons learned in the design, study, and application of standard GAs and their extensions should carry over to GP as argued for example in [5, 14, 15].

The paper starts by describing the algorithms investigated in this paper: GP and PIPE. Section 3 explains test problems. Section 4 provides and discusses experimental results. Section 5 summarizes the paper. Finally, Section 6 concludes the paper and presents important topics for future work.

2 Methods

Both GP and PIPE work with programs encoded as labeled-tree structures and both can be applied to the same class of problems. This section describes GP and PIPE. Both GP and PIPE were implemented using the `lilgp` library developed by GARAGe at the Michigan State University.

2.1 Genetic Programming

Genetic programming (GP) [7] is a genetic algorithm (GA) [3] that evolves programs instead of fixed-length strings. Programs are represented by trees where nodes represent functions and leaves represent variables and constants.

GP starts with a population of random candidate programs. Each program is evaluated on a given task and its fitness is then computed based on the evaluation. A population of promising programs is then selected using one of the standard GA selection operators, such as tournament or truncation selection. Some of the selected programs can be directly copied into the new population, the remaining ones are copied after applying variation operators, such as crossover and mutation. Crossover usually proceeds by exchanging randomly selected subtrees between two programs, whereas mutation usually replaces a randomly selected subtree of a program by a randomly generated one. This process is repeated until termination criteria are met.

Since standard GP variation operators proceed without considering interactions between different components of selected programs, they are likely to experience difficulties with solving problems where different program components interact strongly. However, problems that can be decomposed into subproblems of order one should be easy for any standard GP based on recombination. This intuition is verified with experiments in Section 4.

2.2 PIPE

In the probabilistic incremental program evolution (PIPE) algorithm [13] computer programs or mathematical expressions are evolved like in GP [7]. The initial population is also generated at random. All programs in the population are then evaluated and selection is applied to select the population of promising programs. Instead of applying crossover and mutation to a part of the selected population to generate new programs, PIPE builds a probabilistic model of the selected programs in the form of a tree. This probabilistic model is then sampled to generate new candidate programs that form the new population. The process is repeated until the termination criteria are met.

The probabilistic model in PIPE is a tree with the structure corresponding to the structure of candidate programs. Since different programs may be of different structure and size, the population is first parsed to find the smallest tree that covers every structure in the selected population. If there are functions of different arities, the number of children of each node in the probabilistic model is equal to the maximum arity of a function in this node and for a function of smaller arity, the first children are interpreted as arguments of this function. Each node of a program in the selected population then directly corresponds to one node in the model, whereas the children of each internal node represent arguments of the function in this node.

PIPE then parses the selected population and computes the probabilities of different functions and terminals in each node of the probabilistic model. The nodes of the probabilistic model thus consist of tables of probabilities, and there is one probability for each function or terminal in each node.

Sampling of the probabilistic model starts in the root of the probabilistic model. The same recursive procedure is used to generate each node. First, a function or terminal is generated in the current node based on the distribution encoded by the table of probabilities in this node. If the function requires several arguments, a necessary number of children are generated recursively. The recursive generation terminates in a node whenever a terminal is generated in this node and thus no children have to be generated. Since the probabilistic model is built from an actual population of programs, the sampling will never cross the boundaries of the model.

3 Test problems

In order to test scalability, we need a class of problems where size can be modified while the inherent problem difficulty does not grow prohibitively fast. In fixed-length string GAs, decomposable problems of bounded difficulty [4] can be used as a challenging but solvable class of problems. Two types of decomposable problems for fixed-length string GAs are common: onemax and concatenated traps.

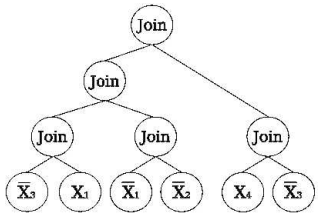

Fig. 1. This figure shows an example of a candidate solution for a 4-primitive `ORDER` problem. The sequence of leaves visited during the inorder parse is $\{\overline{X}_3, X_1, \overline{X}_1, \overline{X}_2, X_4, \overline{X}_3\}$, the expression of this sequence is $\{X_1, \overline{X}_2, \overline{X}_3, X_4\}$, and the fitness of this solution is thus 2.

Similar problems to onemax and concatenated traps were also created for GP where candidate solutions are represented by program trees [5, 14]. Two classes of problems from [14] are considered: (1) `ORDER` (onemax-like, GP-easy problem), and (2) `TRAP` (deceptive-trap-like, GP-difficult problem).

`ORDER` should be easy for any recombination-based GP. However, since standard variation operators do not consider interactions between different program components, `TRAP` can be expected to lead to exponential scalability of both standard GP and PIPE. The problems are described next.

3.1 Problem 1: Order

The primitive set of an l-primitive `ORDER` problem [14] consist of a binary function `JOIN` and l complementary pairs of terminals X_i and $\overline{X}_i$ for $i \in \{1, 2, \ldots, l\}$. A candidate solution of the `ORDER` problem is a binary tree with `JOIN` in all internal nodes and either X_i's or $\overline{X}_i$'s at its leaves. The candidate solution's output is determined by parsing the program tree inorder (from left to right). The program expresses X_i if, during the inorder parse, X_i is encountered before its complement $\overline{X}_i$ and neither X_i nor its complement are encountered earlier. For all $i \in \{1, 2, \ldots, l\}$, if X_i is unexpressed, $\overline{X}_i$ is expressed instead. One terminal is thus expressed from each pair X_i and $\overline{X}_i$.

The fitness is defined as the number of positive terminals X_i that were expressed during the inorder parse. The fitness can thus be seen as onemax applied to a binary string where the ith bit is 1 if X_i was expressed, whereas the ith bit is 0 if $\overline{X}_i$ was expressed (see Figure 1).

3.2 Problem 2: Deceptive Trap

In standard GAs, deceptive functions [2, 4] are designed to thwart the very mechanism of selectorecombinative search by punishing any localized hill-climbing and requiring the mixing of whole building blocks at or above the order of deception. Using such adversarially designed functions is a stiff test of algorithm performance. The idea is that if an algorithm can beat adversarially

designed test functions, it can solve other problems that are equally hard or easier than the adversary.

TRAP is designed to test the same mechanisms in GP. Fitness is computed so that if interactions between different components of the program are not considered, optimization may be misled away from the global optimum. Similarly as with standard GAs on deceptive functions, standard GP is expected to fail in solving TRAP scalably, indicating the need for linkage learning in GP.

Programs in TRAP also consist of one binary function JOIN and l pairs of complementary primitives X_i and $\overline{X}_i$. The expression mechanism of the program for TRAP is identical to that of ORDER. The difference is in the fitness evaluation procedure.

In TRAP, the expressed set of primitives is first mapped to an l-bit binary string. The ith bit of the string is 1 if and only if X_i was expressed; otherwise, the ith bit of the string is 0. The resulting binary string is then partitioned into groups of k bits each (the partitioning is fixed during the entire run) and a trap function is applied to each group:

$$trap_k(u) = \begin{cases} 1 & \text{if } u = k \\ (1-\delta)\left(1 - \frac{u}{k-1}\right) & \text{otherwise} \end{cases} \quad (1)$$

where u is the number of ones in the input string of k bits. The difficulty of the trap function can be adjusted by modifying the values k and δ; in this paper we use traps with $k = 3$ and $\delta = 1$. TRAP fitness function is then computed by adding the contributions of all groups of k bits together.

An important feature of additively separable trap functions is that if looking at the performance of any strict subset of k bits corresponding to one trap, it seems to be better to propagate 0s than 1s (here $\overline{X}_i$ would be propagated at the expense of X_i); however, the optimum is in the string of all 1s (X_i is expressed for any i). As shown in [14], if interactions between different components of the program are not considered, it can be expected that GP will scale up poorly on this problem.

3.3 Other primitives

The purpose of additional tests was to determine how GP and PIPE respond to more complex interactions and unnecessary program primitives. Two additional types of primitives were added into ORDER problem: (1) NEG_JOIN function and (2) JUNK terminals.

NEG_JOIN affects all its descendant terminals by expressing each primitive X_i as its negation $\overline{X}_i$; analogically, all descendants $\overline{X}_i$ are expressed as X_i. If a terminal has more NEG_JOIN ancestors, only one of them is considered and the terminal is negated only once. NEG_JOIN is unnecessary for solving ORDER. Furthermore, NEG_JOIN introduces interactions into ORDER because the best value in each leaf depends on its ancestors. Nonetheless, these interactions

are relatively simple as many leaves are expected to contain NEG_JOIN on the path to the root.

JUNK terminals represent unnecessary primitives that are irrelevant for the particular problem. In biological terms, JUNK terminals correspond to junk code in DNA. During the expression phase, JUNK terminals are simply ignored and thus they do not influence the overall fitness at all. The influence of JUNK terminals can be tuned by changing the number of unique JUNK terminals.

4 Experiments

This section compares the performance of GP and PIPE on three variants of ORDER and one variant of TRAP.

4.1 Description of experiments

The scalability of GP and PIPE was tested on four classes of problems: (1) ORDER (no JUNK or NEG_JOIN), (2) TRAP (no JUNK or NEG_JOIN), (3) ORDER with NEG_JOIN, and (4) ORDER with JUNK terminals where the number of unique JUNK terminals is set to $l/5$.

The scalability experiments were performed by testing both algorithms on problem instances with an increasing number l of primitives. Additionally, the effects of increasing the number of unnecessary primitives on the performance of GP and PIPE were studied by testing GP and PIPE on a 20-primitive ORDER with an increasing number of JUNK terminals (from 5 to 40).

Binary tournament selection was used in both GP and PIPE. The probability of crossover in GP was set to 1.0. To focus on the effects of recombination, no mutation was used. The initial population in both methods was generated using the standard half-and-half method. Maximum tree depth was set to be one more than the depth of the minimum tree to store the global optimum. The population size that is within 10% of the minimum population size required to solve 30 independent runs was used. The population size was determined using a bisection method. The runs were terminated when the algorithms found the global optimum or when the number of generations was too large for the particular problem (based on experience).

4.2 Results

Figure 2 shows the scalability of GP and PIPE on ORDER without NEG_JOIN or JUNK terminals. Problem instances of different size were examined; more specifically, $l = 5$, 10, 20, 40, 60, 80, and 100. The figure shows the average number of function evaluations of 30 successful runs with respect to the problem size l. The results indicate that PIPE is slightly more efficient than GP but both GP and PIPE scale up with a low-order polynomial. These results are in agreement with the behavior observed in binary-string GAs on the

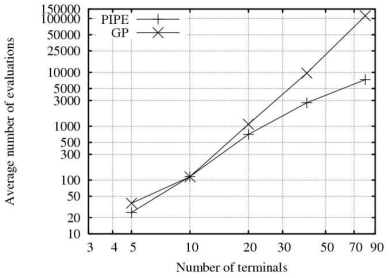

Fig. 2. Scalability of GP and PIPE on ORDER.

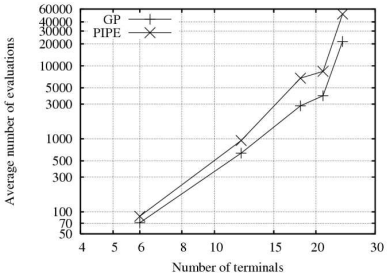

Fig. 3. Scalability of GP and PIPE on TRAP.

simple onemax problem. On onemax, both simple GA and UMDA find the optimum in low-order polynomial time [4,6,10,12]; however, UMDA performs slightly better [12] because it uses a more effective recombination for this type of problems.

Figure 3 compares the scalability of GP and PIPE on TRAP without NEG_JOIN or JUNK terminals. Problem instances of different size were examined; more specifically, $l = 6$, 12, 18, 21, 24, and 33. On TRAP, GP performs slightly better than PIPE. This can be explained by its weaker recombination operator because here recombination causes disruption of important partial solutions [16] as can be hypothesized based on the performance of standard GAs on similar problems. Nonetheless, both GP and PIPE scale up poorly and they indicate an exponential growth of the number of function evaluations with problem size.

Figure 4 compares the scalability of GP and PIPE on ORDER with NEG_JOIN. Problem instances of different size were examined; more specifically, $l = 5$, 10, 20, 40, 60, 80, and 100. Both GP and PIPE perform similarly as on basic ORDER without NEG_JOIN, but there is a slight decrease in their performance because of the interactions introduced by NEG_JOIN.

Figure 5 compares the scalability of GP and PIPE on ORDER with $l/5$ unique JUNK terminals. For example, a problem instance with $l = 20$ positive terminals contains 4 unique JUNK terminals. Both GP and PIPE seem to be capable of dealing with these irrelevant terminals and achieve performance comparable to that on basic ORDER.

The last two sets of experiments are similar in that they show how the performance of GP and PIPE changes when adding irrelevant terminals into the representation. ORDER with $l = 20$ terminals is used with the number of JUNK terminals ranging from 5 to 40 (5, 10, 15, 20, and 40). The experiments differ in the bound on the maximum tree depth. Figure 6 shows the results with the depth limited to at most 6 (so there are at most 7 levels including the root). Figure 7 shows the results with the depth limited to at most 7 (so there are at most 8 levels including the root). The problem with the smaller maximum depth is more difficult for both GP and PIPE because JUNK terminals obstruct the creation of an optimal solution that is only slightly

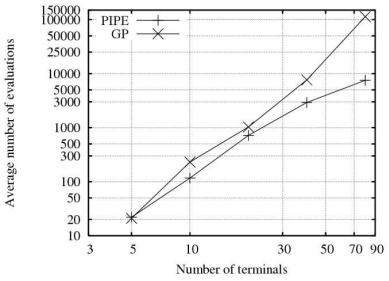

Fig. 4. Scalability of GP and PIPE on `ORDER` with `NEG_JOIN`.

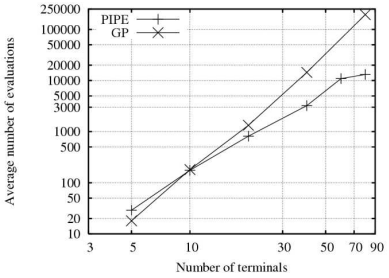

Fig. 5. Scalability of GP and PIPE on `ORDER` with $l/5$ copies of `JUNK` terminals.

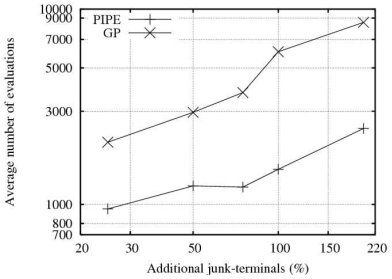

Fig. 6. Scalability of GP and PIPE on `ORDER` with an increasing number of `JUNK` terminals (from 5 to 40).

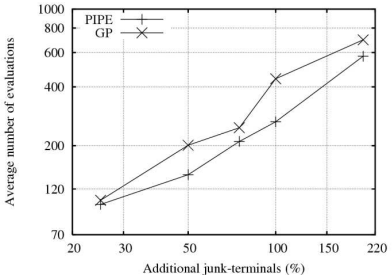

Fig. 7. Scalability of GP and PIPE on `ORDER` with an increasing number of `JUNK` terminals (from 5 to 40).

larger than the maximum allowed tree. PIPE deals better with this "lack of space" than GP does. However, in both cases, the number of evaluations still appears to grow with a low-order polynomial as irrelevant terminals are added.

5 Summary

This paper focused on the scalability of two GP algorithms: standard GP and PIPE. Two basic test functions were used: `ORDER` and `TRAP`. Both functions were defined using one binary function `JOIN` and l complementary terminal pairs. `ORDER` can be solved without considering interactions between different program components, whereas `TRAP` introduces strong interactions, which make this function difficult for both standard crossover and mutation of GP, as well as the probabilistic recombination of PIPE.

The scalability of GP and PIPE was tested on basic `ORDER` and `TRAP`. Additionally, `ORDER` was extended by adding either of the following two primitives: (1) a binary function `NEG_JOIN` and (2) `JUNK` (or irrelevant) terminals. Thus, there were 4 problem types examined.

On all four problem types, the scalability of GP and PIPE was first tested by applying these algorithms to problem instances of different size (number l

of positive terminals). Then, the sensitivity of GP and PIPE to the proportion of irrelevant terminals to the relevant ones was examined.

6 Conclusions and Future Work

The results presented in this paper indicate that the behavior of different variants of GP can be expected to be similar to that of standard binary-string GAs. There are two important consequences of this fact. First, as it was indicated in [14], to solve some classes of GP problems scalably, linkage learning may have to be incorporated into GP in order to identify and exploit interactions between different program components. Second, the lessons learned in the design and application of binary-string GAs should carry over to GP as argued for example in [5,15]; the first steps along this direction are represented by the decision-making model of the population sizing in GP [15], which was based on the decision-making population-sizing model for standard GAs [4,6].

The results also indicate that if the recombination operator captures interactions in the problem properly, increasing the mixing effects of recombination leads to better performance. That is why PIPE outperformed standard GP on problems where program components could be treated independently. This fact together with the need for linkage learning should encourage the application of probabilistic recombination operators of estimation of distribution algorithms (EDAs) [8, 9, 11] to the domain of GP.

Finally, the results show that both GP and PIPE can deal with irrelevant terminals and unnecessary functions relatively well and their performance gets only slightly worse when adding these primitives.

Future work should study the scalability of GP, PIPE, and other similar approaches on the problems presented in this paper and other problems where problem size can be modified without affecting the inherent problem difficulty. The efforts to introducing linkage learning into GP (for example [1, 14]) should continue to succeed in the design of robust and scalable GP algorithms applicable to broad classes of difficult decomposable problems.

Acknowledgments

This work was partially supported by the Research Award and the Research Board at the University of Missouri. This work was also sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant F49620-03-1-0129, the National Science Foundation under ITR grant DMR-99-76550 (at Materials Computation Center), and ITR grant DMR-0121695 (at CPSD), and the Dept. of Energy under grant DEFG02-91ER45439 (at Fredrick Seitz MRL). The U.S. Government is authorized to reproduce and distribute reprints for government purposes notwithstanding any copyright notation thereon.

References

1. P. A. N. Bosman and E. D. de Jong. Learning probabilistic tree grammars for genetic programming. pages 190–199, 2004.
2. K. Deb and D. E. Goldberg. Analyzing deception in trap functions. *Foundations of Genetic Algorithms*, 2:9–108, 1993.
3. D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, Reading, MA, 1989.
4. D. E. Goldberg. *The design of innovation: Lessons from and for competent genetic algorithms*, volume 7 of *Genetic Algorithms and Evolutionary Computation*. Kluwer Academic Publishers, 2002.
5. D. E. Goldberg and U.-M. O'Reilly. Where does the good stuff go, and why? how contextual semantics influence program structure in simple genetic programming. *Proceedings of the First European Workshop on Genetic Programming*, 1391:16–36, 14-15 Apr. 1998.
6. G. R. Harik, E. Cantú-Paz, D. E. Goldberg, and B. L. Miller. The gambler's ruin problem, genetic algorithms, and the sizing of populations. *Proceedings of the International Conference on Evolutionary Computation (ICEC-97)*, pages 7–12, 1997. Also IlliGAL Report No. 96004.
7. J. R. Koza. *Genetic programming: On the programming of computers by means of natural selection*. MA: The MIT Press, Cambridge, 1992.
8. P. Larrañaga and J. A. Lozano, editors. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer, Boston, MA, 2002.
9. H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions I. Binary parameters. *Parallel Problem Solving from Nature*, pages 178–187, 1996.
10. H. Mühlenbein and D. Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm: I. Continuous parameter optimization. *Evolutionary Computation*, 1(1):25–49, 1993.
11. M. Pelikan, D. E. Goldberg, and F. Lobo. A survey of optimization by building and using probabilistic models. *Computational Optimization and Applications*, 21(1):5–20, 2002. Also IlliGAL Report No. 99018.
12. M. Pelikan, K. Sastry, and D. E. Goldberg. Scalability of the Bayesian optimization algorithm. *International Journal of Approximate Reasoning*, 31(3):221–258, 2002. Also IlliGAL Report No. 2001029.
13. R. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.
14. K. Sastry and D. E. Goldberg. Probabilistic model building and competent genetic programming. April 2003.
15. K. Sastry, U.-M. O'Reilly, and D. E. Goldberg. Convergence-time models for the simple genetic algorithm with finite population. IlliGAL Report No. 2001028, University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory, Urbana, IL, 2001.
16. D. Thierens. *Analysis and design of genetic algorithms*. PhD thesis, Katholieke Universiteit Leuven, Leuven, Belgium, 1995.