Royal Trees as a Benchmark Problem for Genetic Programming: A Parallel Processing Example

Bill Punch, Doug Zongker, and Erik Goodman
MSU GARAGe (Genetic Algorithms Research and Applications Group)
punch@cps.msu.edu, zongker@cps.msu.edu, goodman@egr.msu.edu

submit for WORKSHOP and BOOK

1 Introduction

In the fall of 1994 we began work on a general purpose genetic programming tool called *lilgp*. We had multiple objectives in designing this system, including: making it as fast and memory efficient as possible, making it portable enough to run on a wide variety of machines, and providing support for a number of features not typically found in other GP systems, especially support for parallel processing. This support for parallel processing was focused not so much on the ability to run on multiple machines (though this is indeed possible) but instead to provide support for experiments using multiple, interacting subpopulations. We have had considerable experience in these kinds of experiments with GAs and wanted to transfer that expertise to GP[2,3].

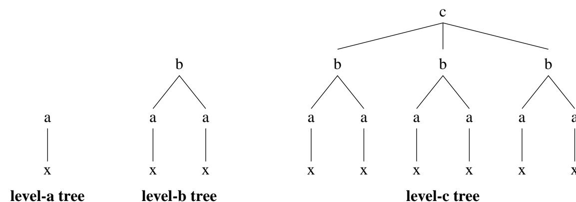
2 Benchmarks

We began to turn our attention to benchmark programs for GPs. We implemented a number of problems from GP I[1]: the Boolean 11-multiplexer, artificial ant, symbolic regression, pole-balancer, and hamstrung squad car and tried to compare the results but were looking for something more definitive. As GA researchers we began work on a royal road-type problem[4]. Holland's royal road serves two purposes. First, it provides proof-of-principle for the kind of difficult problems, exhibiting deception, that a genetic algorithm is capable of solving. Second, it serves as a benchmark of performance for tuning GA parameters. At ICGA93, Holland claimed a specialized, properly tuned GA could solve all but the last level of a 4-level royal road problem in 10,000 evaluations or less.

Our goal then was to try and devise a "royal tree" problem for GP that would share some characteristics of the royal road problem: it would be a difficult program that would show the capabilities of the GP and also provide a benchmark for tuning GP parameters. We saw very little evidence of such work in the literature. The only reference we have been able to find up to the time of this writing was the dissertation of W. A. Tackett[4], who incorporated so-called "constructional problems" like the royal road into his research on GP.

We also wanted to incorporate another feature into our function. One of the main difficulties in evaluating how well a GP worked was that problem performance was nearly the only measure of quality. If the GP solved the problem, it worked. Unfortunately, a GP could generate an infinity of programs to solve any particular problem. It seemed to us that tree *structure* should also be an issue. This seemed like the kind of realistic restriction that a GP solving a practical problem would have to face. Finally, we wanted our function to have a *progression* of "correct" answers, with the level of the problem dictating the level of complexity faced by the GP.

The *royal tree* consists of a single base function that is specialized into as many cases as necessary, depending on the desired complexity of the resulting problem. We define a series of functions, `a`, `b`, `c`, etc. with increasing arity. (An `a` function has arity 1, a `b` has arity 2, and so on.) We also define a number of terminals `x`, `y`, `z`. For any depth, we define a "perfect"



tree as shown in Figure 2. A level-a tree is an `a` with a single `x` child. A level-b tree is a `b` with two level-a trees as children. A level-c tree is a `c` with three level-b trees as children, and so on. A level-e tree has depth 5 and 326 nodes, while a level-f tree has depth 6 and 1927 nodes.

The raw fitness of the tree is the score of its root. Each function calculates its score by summing the weighted scores of its children. If the child is a perfect tree of the appropriate level (for instance, a complete level-c tree beneath a `d` node), then the weight is *FullBonus*. If the child has the correct root but is not a perfect tree, then the weight is *PartialBonus*. If the child's root is incorrect, then the weight is *Penalty*. In addition, if the function is itself the root of a perfect tree, the final sum is multiplied by *CompleteBonus*. Typical values used are: $FullBonus = 2$, $PartialBonus = 1$, $Penalty = \frac{1}{3}$, and $CompleteBonus = 2$.

3 Results

We compare runs between the artificial ant problem (Santa Fe trail, maximum 400 steps) and level-e royal tree. For the ant problem, the maximum fitness was 89; for the royal tree, the maximum fitness was 122,880. We did two major groups of runs—runs with single populations and run with multiple populations. All runs were done in replicates of 16, with a maximum of 500 generations.

The common parameters for the single population runs were as follows: population size of 3500, 90% internal-point crossover and 10% external-point crossover, maximum tree depth of 17, maximum tree size of 750 nodes, and initialization using the ramped half-and-half method with initial depths between 2 and 7 inclusive.

Parallel runs used the same parameters as the single population runs with the following differences. We conducted runs using two different parallel processing architectures. The first was a ring architecture of 7 subpopulations of 500 each (total 3500), exchanging the 5 best solutions to its single neighbor every 10 generations (i.e., $1 \rightarrow 2 \rightarrow \cdots \rightarrow 7 \rightarrow 1$.) The other was an injection architecture, a hierarchical arrangement where 6 subpopulations of 500 gave their 5 best solutions (total of 30) to a seventh subpopulation (also of 500), every 10 generations. There is no communication between the six, only a one-way injection of information ($[1, \ldots, 6] \rightarrow 7$). We have previously used injection architecture in GA applications and shown super-linear speedup in terms of number of evaluations needed to reach a particular quality of solution: the more subpopulations, the fewer total evaluations required[2].

In Table 1, $W : (x, y)$ represents the number of wins (finding completely correct solutions), where x is the number of runs that were wins, and y is the average generation number the win occurred in. $L : (a, b, c)$ is the number of losses (no correct solution after 500 generations), where a is the number of runs that were losses, b is the average best-of-run fitness, and c is the average generation the last best-of-run occurred in.

Problem	$P_c = 0.9, P_r = 0.1, P_m = 0.0$		$P_c = 0.875, P_r = 0.075, P_m = 0.05$	
	Overselection	Prop. Selection	Overselection	Prop.selection
Single Population Runs				
ant	$W : (7, 156)$ $L : (9, 78, 198)$	$W : (2, 265)$ $L : (14, 68, 208)$	$W : (10, 109)$ $L : (6, 73, 300)$	$W : (7, 112)$ $L : (9, 67, 158)$
royal tree	$W : (1, 145)$ $L : (15, 6144, 47)$	$W : (0, 0)$ $L : (16, 71, 85)$	$W : (8, 233)$ $L : (8, 9046, 159)$	$W : (0, 0)$ $L : (16, 71, 92)$
Ring Architecture Multipopulation Runs				
ant	$W : (4, 160)$ $L : (12, 68, 312)$	$W : (7, 286)$ $L : (9, 71, 257)$	$W : (6, 208)$ $L : (10, 74, 313)$	$W : (7, 240)$ $L : (9, 73, 244)$
royal tree	$W : (0, 0)$ $L : (16, 10005, 338)$	$W : (0, 0)$ $L : (16, 83, 62)$	$W : (0, 0)$ $L : (16, 16284, 373)$	$W : (0, 0)$ $L : (16, 76, 181)$
Injection Architecture Multipopulation Runs				
ant	$W : (2, 297)$ $L : (14, 70, 326)$	$W : (8, 270)$ $L : (8, 70, 272)$	$W : (2, 116)$ $L : (14, 70, 304)$	$W : (6, 309)$ $L : (10.74.256)$
royal tree	$W : (0, 0)$ $L : (16, 20764, 395)$	$W : (0, 0)$ $L : (16, 81, 152)$	$W : (0, 0)$ $L : (16, 18354, 405)$	$W : (0, 0)$ $L : (16, 83, 192)$

Table 1: Results

4 Discussion

Our previous work with GAs and parallel architectures showed dramatic improvement of performance, both in terms of better answers and in terms of fewer generations required to achieve such answers. This is clearly not the case for all GP problems. The ant problem showed dramatic improvement using fitness-proportionate selection for both rings and injection, but showed loss of performance using overselection. The royal tree did well with a single population, overselection, and mutation, but worse under all parallel conditions, though the average loss-scores in the parallel runs did improve. More interesting are the graphs of multiple population progress under injection (not shown here due to space). Here, the ant shows a dramatic improvement in the injected population *average* fitness, while the royal tree shows no such effect. Our hypothesis is that there is a "stairstep" action seen with the royal tree, that each subpopulation must solve level-c to get to level-d, level-d to get to level-e, etc. This synchronizes the subpopulations to an extent that prevents parallel architecture from increasing diversity and therefore performance. More work is required to confirm these findings. We feel that the royal tree provides a fresh perspective on "performance," one that practical GP applications will have to face.

5 Bibliography

1. J. R. Koza. *Genetic Programming.* Bradford/MIT Press, 1992.
2. S.-C. Lin, W. F. Punch, and E. D. Goodman. Coarse-grain parallel genetic algorithms: Categorization and new approach. *Sixth IEEE SPDP*, pages 28–37, October 1994.
3. W. F. Punch, R. C. Averill, E. D. Goodman, S.-C. Lin, and Y. Ding. Design using genetic algorithms—some results for composite material structures. *IEEE Expert*, 10(1), Feb 1995.
4. W. A. Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs.* PhD thesis, University of Southern California, April 1994.

Genetic programming is a branch of computer science studying heuristic algorithms based on neo-Darwinian principles for synthesizing programs, i.e., discrete symbolic compositional structures that process data. Summary of our rst glimpse at GPâ€" 27. Why should GP be considered a viable approach to program synthesis?Â GP Benchmarks. Problem Sextic Septic Nonic. R1 R2 R3.Â Running a program on multiple inputs can be expensive. Particularly for some types of data, e.g., images Solutions: Caching of outcomes of subprograms Parallel execution of programs on particular tness cases Bloat prevention methods. The Challenges for GPâ€" 59. Variants of GP. Parallel genetic algorithm is such an algorithm that uses multiple genetic algorithms to solve a single task [1]. All these algorithms try to solve the same task and after theyâ€™ve completed their job, the best individual of every algorithm is selected, then the best of them is selected, and this is the solution to a problem.Â These genetic algorithms do not depend on each other, as a result, they can run in parallel, taking advantage of a multicore CPU. Each algorithm has its own set of individual, as a result these individuals may differ from individuals of another algorithm, because they have different mutation/crossover history. Paper [2] describes a parallel genetic algorithm that uses two independent algorithms to improve its performance. Royal Trees as a Benchmark Problem for Genetic Programming: A Parallel Processing Example. August 1995. AuthorsÂ This support for parallel processing was focused not so much on the ability to run on multiple machines (though this is indeed possible) but instead to provide support for experiments using multiple, interacting subpopulations. We have had considerable experience in these kinds of experiments with GAs and wanted to transfer that expertise to GP[2,3]. 2 Benchmarks We began to turn our attention to benchmark programs for GPs.Â We describe in this paper a parallel implementation of Multi Niche Genetic Programming that we use to test the performance of a modified migration model.