The Royal Tree Problem, a Benchmark for Single and Multi-population Genetic Programming

appears in "Advances in Genetic Programming II", MIT Press, Pete Angeline and Kim Kinnear, editors

Bill Punch, Doug Zongker, and Erik Goodman

We report on work done to develop a benchmark problem for genetic programming, both as a difficult problem to test GP abilities and as a platform for tuning GP parameters. This benchmark, the *royal tree*, is a function that accounts for *tree shape* as part of its evaluation function, thus it controls for a parameter not often found in the GP literature. It also is a progressive function, allowing the user to set the difficulty of the problem attempted. We not only describe the function, but also report on results of using *island parallelism* for solving GP problems. The results obtained are somewhat surprising, as it appears that a single large population outperforms a group of smaller populations under all the conditions tested.

15.1 Introduction

Given the multiplicity of GP programs that could produce the correct solution for a particular problem, it is difficult to judge the effectiveness of various architectural changes or parameter settings on the performance of a GP system. We encountered these problems directly in the design of our genetic programming tool *lilgp*. When lilgp was completed, we wanted to test how well it solved a set of standard GP problems. In fact, for a new GP system it is difficult to judge whether it is performing as intended or not, since the programs it generates are not necessarily identical to those generated by other GP systems. This raised two questions: what constitutes a "standard" problem in GP, and how do we rate the performance of a system on such a problem.

One of the goals of this research was to create a benchmark problem to test how well a particular GP configuration would perform as compared to other configurations. Such benchmarks have existed for some time in the GA field, in particular the royal road problems of Holland [Jones 1994]. In creating the royal road, Holland addressed three issues. First, the royal road provides a 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. For example, 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. Third, Holland in fact created a *family* of royal road functions as benchmarks for GA systems, in order to offer a controllable level of difficulty, control over the amount of deception, clearly defined building blocks and control over linkage among adjacent building blocks.

We were interested in addressing the same issues, showing how GP could address difficult problems as well as providing a tunable benchmark for comparing GP configurations. Furthermore, we were interested in creating and using this benchmark to test the effectiveness of coarse-grain parallel GP's as compared to single population GP's. It is not obvious that the solutions typically found by a coarse-grain parallel GP architecture will resemble those produced by a single population GP architecture, especially given that the only measure of similarity typically used is whether the generated trees produced the proper answer. A suitable benchmark problem which has only a *single, unique tree* as an answer can remove this obstruction. Then, given a search for a unique, optimal solution, the question of parallel *speedup* provided by a coarse-grain architecture, in the sense of reduction of total function evaluations obtained through the use of multiple populations, can be more clearly addressed (See section 15.4.2 for details on speedup).

In contrast then to most GP problems, our proposed benchmark possess a unique solution, although that solution may be attained in different ways. Unlike the typical GA's chromosome, in which the structure and length are usually fixed, a GP's trees are usually subject only to limits on depth, or the number of nodes, and these are provided only as computational limits, i.e. limits to allow the GP to compute effectively on a particular computer, not as a problem limit. Therefore, in order to judge how parsimoniously a GP system can perform, it would be useful to have a family of benchmarks that contain unique solutions.

15.2 Benchmarks

Our goal was to devise a problem for GP that would share some characteristics of the royal road problem. 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 [Tackett 1994], who incorporated so-called "constructional problems" like the royal road into his research on 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`,

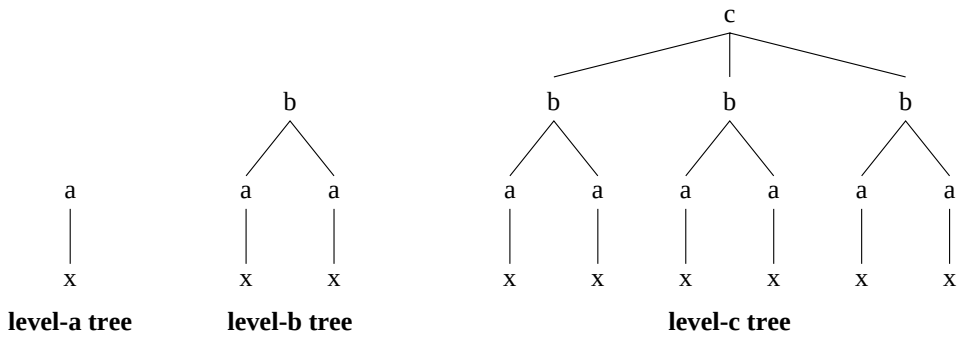

Figure 15.1:
A Perfect level-a, level-b and level-c Royal Tree.

`z`. For any depth, we define a "perfect" tree as shown in Figure 15.1. 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 (or any subtree) is the score of its root. Each function calculates its score by summing the weighted scores of its direct 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 score of that subtree, times a *FullBonus* weight, is added to the score of the root. 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*. After scoring the root, 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 $CompleteBonus = 2$. The score base case is a level-a tree, which has a score of 4 (the a—x connection is worth 1, times the FullBonus, times the CompleteBonus).

The reasoning behind this "stair-step" approach to the function is based on the reasoning originally used by the royal road. Many combinations of solutions can be found through genetic combination, but each *proper* combination gives a big jump in evaluation credit. The FullBonus is provided to give a large credit to those trees that find the correct, complete royal tree child. Since a deeper royal tree, such as a level-f tree, has a number of complete royal trees as children, each complete subtree found gives a large credit to that particular solution. The PartialBonus is used to give credit for finding the proper, direct child for a node, even if that direct child is not the root of a royal tree. This pressure is not as great as the FullBonus, but

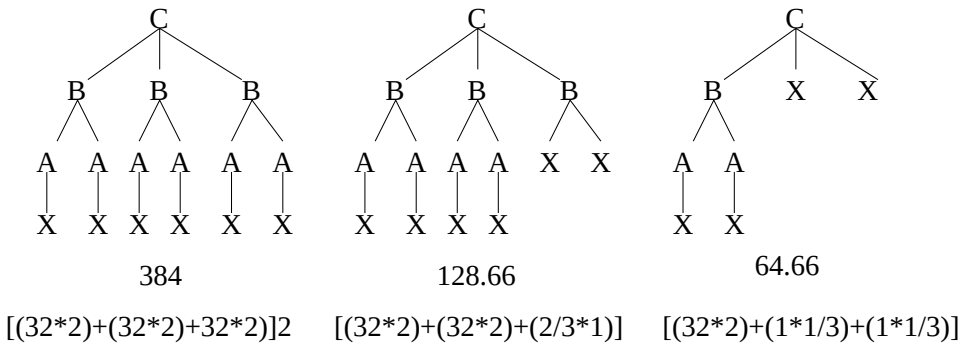

Figure 15.2:
Some example royal trees with scores, and the process used to derive those scores

it is an effective incentive since the score is determined recursively down the tree and each node receives some credit when if finds its proper, direct children. If a node does not have the correct, direct children, it is penalized by Penalty, making the FullBonus and PartialBonus even more effective. Finally, if the resulting tree itself is complete, then a very large credit is given. Some examples of complete and partial royal trees are shown in Figure 15.2. In this figure, the method of scoring is given. For example, in the leftmost tree, the score of a complete level-b tree is 32, so the score of a complete level-c tree is $[(32 * FullTree) + (32 * FullTree) + (32 * FullTree)] * CompleteTree$.

The reasoning behind the increase in arity required at each increased level of the royal tree is simple, we wanted to make a hard problem (and we demonstrably did, please see the result sections) for GP to solve. That is, we could simply have required that a proper ordering of say a b-function (multiple levels of a 2-arity function) for the tree, but requiring *increasing* arity as we climb to the next level dramatically increases the difficulty of the problem, and provides a measure of how well a GP can perform. For example, it is *extremely* difficult to climb to a level-f tree and we have never succeeded in climbing to level-g.

15.2.1 Differences Between Royal Road and Royal Tree

There are a number of differences worth noting between Holland's royal road functions and our royal tree function. The first is that we do not explicitly introduce deception into our functions. In fact, we did provide a method whereby we *could* introduce deception, namely the existence of 3 terminals: `x`, `y` and `z`. The `x` is the

only terminal allowable for a complete royal tree, but we could modify the fitness function so that connection of a `y` or `z` terminal gives deception, that is `y` and `z` would give a short-term high fitness to a tree using them as terminals, but only a `x` terminated tree would get the FullBonus and CompleteBonus boosts. We did not use this deception in the present problems since the royal tree in fact appeared quite difficult already without explicit deception.

The other interesting idea that Holland used and we did not was the concept of *introns*. An intron for a GP is essentially a group of non-functional nodes that simply "fill space" in the tree. When the tree is evaluated, the intron is ignored (essentially extracted from the tree). An intron can serve as "protection" for important subtrees from destructive crossovers. That is, crossover or mutation on the intron portion of a tree does not change the evaluation of the real function nodes of the tree. Thus subtrees can evolve intron "padding" to surround important subtrees, protecting them from disruption. Others have investigated the effect of such features on GA operation [Levenick 1995]. This is a very interesting area and we are presently experimenting with introns to address some of the difficulties we encountered in solving royal trees.

15.3 Single Population Results

We compare runs between the artificial ant problem using the Santa Fe trail (maximum 400 time steps)[Koza 1992, pg 147] and a level-e royal tree. We choose the ant because, of all standard problems listed by Koza[Koza 1992], it was the most similar in terms of the the structure of the function sets and terminals (no Ephemeral Random Constants for example) and it requires only a modest amount of time to solve (unlike, for example, the 11 multiplexer). For the ant problem, the maximum fitness was 89; for the royal tree, the maximum fitness was 122,880 given the weighting scheme described above. All runs were done in replicates of 16, with a maximum of 500 generations.

The common parameters for these 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. We ran the experiments under a number of conditions that used over-selection versus proportional selection, and with or without mutation. We define over-selection per Koza[Koza 1992, pg 98], that is we select 80% of the time from a subset of the population which constitutes the top z% of the population's fitness when the population is sorted in fitness order. The calculation of z is based on the formula $320/popsize$, or 32% for $popsize \leq 1000$,

Table 15.1:
Single Population Results

Problem	$P_c = 0.9, P_r = 0.1, P_m = 0.0$		$P_c = 0.875, P_r = 0.075, P_m = 0.05$	
	Over-selection	Prop. Selection	Over-selection	Prop. Selection
ant	$W : (7, 156)$	$W : (2, 265)$	$W : (10, 109)$	$W : (7, 112)$
	$L : (9, 78, 198)$	$L : (14, 68, 208)$	$L : (6, 73, 300)$	$L : (9, 67, 158)$
royal tree	$W : (1, 145)$	$W : (0, 0)$	$W : (8, 233)$	$W : (0, 0)$
	$L : (15, 6144, 47)$	$L : (16, 71, 85)$	$L : (8, 9046, 159)$	$L : (16, 71, 92)$

16% for $popsize = 2000$ and so on. Results are shown in Table15.1 below. In the Table, $W : (x, y)$ represents the number of wins (finding completely correct solutions *before* 500 generations), where x is the number of runs that were wins, and y is the average generation number in which the correct answer occurred. $L : (a, b, c)$ 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.

Finally, in the table P_c is the crossover rate, P_r the reproduction rate, and P_m is the mutation rate.

15.3.1 Discussion of Single Population Results

As expected, over-selection for a population size of 3500 out-performed proportional selection in all instances. This difference, however, was much more dramatic in the royal tree than in the ant problem, since in 32 runs the royal tree never found a solution using proportional selection. In fact, the royal tree averaged a fitness of only 71 (out of 122,880) in those 32 runs using proportional selection, and never improved that performance after around the 90th generation out of the 500 generations.

Furthermore, mutation improved the performance of both problems under both selections, though again the difference was most dramatic in the over-selection case for the royal tree. In over-selection without mutation, only one correct answer was found. In particular we note that the average best-of-run fitness for over-selection without mutation was 6144, the score for a level-d tree, suggesting that all but one population got stuck at a local optimum of a level-d tree. However, over-selection with mutation found 8 correct answers with an average best-of-fitness for the incorrect answers of 9046. This behavior is due to the royal tree's susceptibility to a kind of convergence not usually found in other problems. In solving the royal tree problem, the GP must first discover a level-b tree before it solves the level-c tree, a level-c tree before solving the level-d tree etc. This often means that higher level

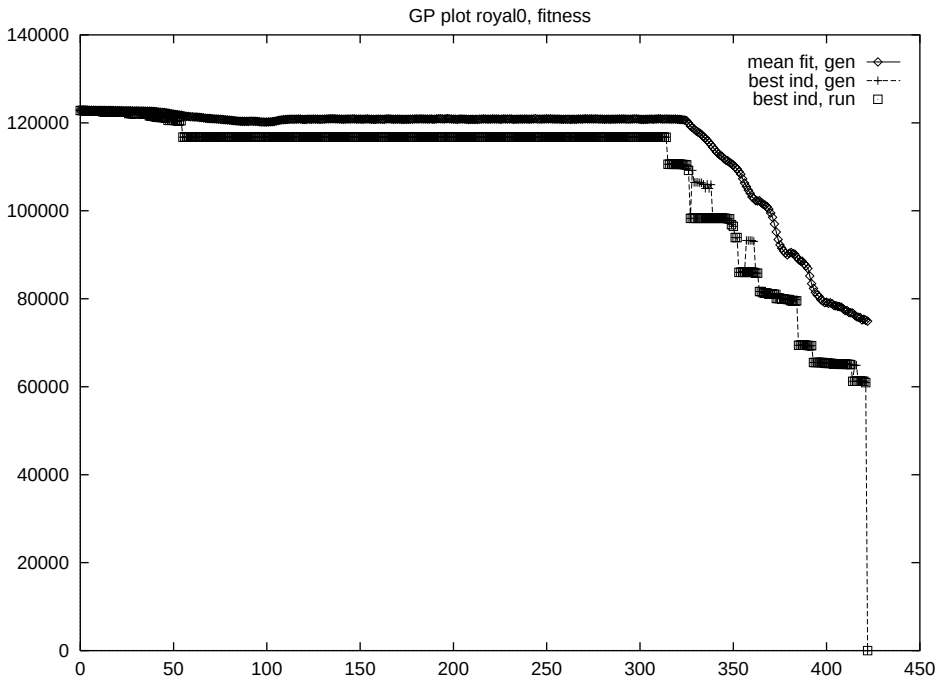

Figure 15.3:
Fitness (y-axis) vs. Generation Plot of a Successful Royal Tree Run

trees such as level-e or level-f cannot be formed since most of their `e` and `f` nodes are lost before they can be used. Mutation solves this dilemma by introducing these nodes back into the population.

This convergence behavior can be seen clearly in the following two figures. Figure 15.3 shows typical performance of a royal tree run that succeeded, Figure 15.4 shows typical performance of a royal tree run that did not find the solution (it found a local optimum from which it could not escape. Note the change of scale of the y coordinate for increased detail). Note also that the successful run made steady progress, followed by a "leap" at the end where it discovered the final solution. The failed run hit the local optimum and made no progress after that.

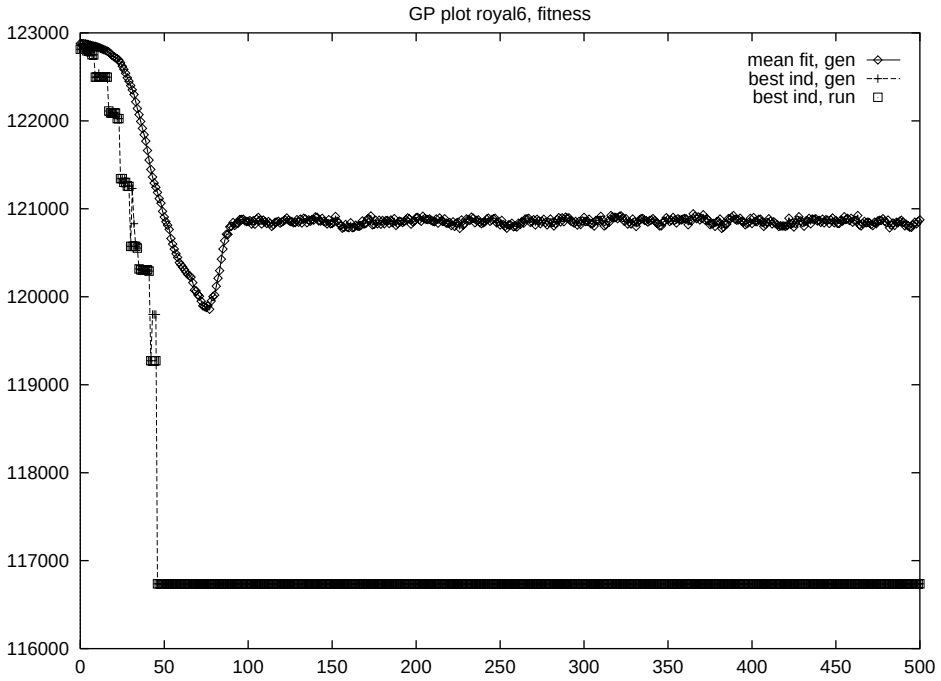

Figure 15.4:
Fitness (y-axis, note scale) vs. Generation Plot of an Unsuccessful Royal Tree Run

15.4 Coarse-Grain Parallel GPs

There are a number of approaches that can remedy the premature convergence seen in Figure 15.4. One of the most appealing is parallelization of GA's (PGA's) or GP's (PGP's), which produces a more realistic model of evolution than a single large population. Most of this kind of parallel processing work has been done in the area of PGA's. In GA literature [Lin et. al. 1994; Manderick and Spiessens 1989; Mulhenbein 1989; Starkweather et. at. 1991; Tanese 1989], it is has been shown that PGA's both decrease processing time and better explore the search space. Unlike sequential GA's which pay a high computational cost for maintaining subpopulations based on similarity comparisons, PGA's maintain multiple, separate subpopulations which may be allowed to evolve independently. This allows each subpopulation to explore different parts of the search space, each maintaining its own high-fitness individuals and controlling how mixing occurs with other subpopulations, if at all.

There are three parallel architectures reported in the literature: micro-grain, fine-grain and coarse grain (also called island-parallel) [Lin et. al. 1994]. We will focus here on coarse-grain work.

Coarse-grain GAs (cgGAs) maintain independent subpopulations (often referred to as "islands", giving rise to the term "island parallel GAs") which occasionally exchange solutions. The frequency of migration among subpopulations is typically small, and is selected so as to achieve a problem-specific balance between combining good schemata (building blocks) discovered in different subpopulations and allowing the subpopulations to search relatively independently (i.e., promoting diversity). Coarse-grain GAs have been shown to outperform "serial" GAs dramatically in many contexts. We have categorized cgGAs according to three characteristics: migration method (isolated island, synchronous island, or asynchronous island), connection scheme (static or dynamic), and node homogeneity (homogeneous or heterogeneous) [Lin et. al. 1994].

15.4.1 Injection Island GAs

We have begun work on a new PGA architecture called injection island GA's (iiGA's). iiGA's are a class of asynchronous, static- or dynamic-topology, heterogeneous node GA's. The two most interesting aspects of an iiGA are its migration rules and the heterogeneous nature of its nodes.

15.4.1.1 iiGA Heterogeneity

GA problems are typically encoded as an n-bit string which represents a complete solution to the problem. However, for many problems, the resolution of that bit string can be allowed to vary. That is, we can represent those n bits in n' bits, $n' < n$, by allowing one bit in the n'-long representation to represent r bits, $r > 1$, of the n-long bit representation. In such a translation, all r bits take the same value as the one bit from the n'-long representation and vice-versa. Thus the n'-long representation is an *abstraction* of the n-long representation. More formally, let:

$$n = p \times q \tag{15.1}$$

where p and q are integers, $p, q \geq 1$. Once p and q are determined, we can re-encode a *block* of bits $p' \times q'$ as 1 bit if and only if

$$p = l \times p', q = m \times q' \tag{15.2}$$

where l and m are integers, $l, m \geq 1$.

Such an encoding has the following basic properties:

1. The smallest block size is 1×1. The search space is 2^n.
2. The largest block size is $p \times q$. The search space is $2^1 = 2$.
3. The search space with a block size $p' \times q'$ is $2^{p/p'} \times 2^{q/q'}$.

An iiGA has multiple subpopulations that encode the same problem using different block sizes. Each generates its own "best" individual separately[1]

15.4.1.2 iiGA migration rules

An iiGA may have a number of different block sizes being used in its subpopulations. To allow interchange of individuals, we only allow a one-way exchange of information, where the direction is from a low resolution to a high resolution node. Solution exchange from one node type to another requires translation to the appropriate block size, which is done without loss of information from low to high resolution. One bit in an n'-long representation is translated into r bits with the same value in an n-long representation. Thus all nodes inject their best individual into a higher resolution node for "fine-grained" modification. This allows search to occur in multiple encodings, each focusing on different areas of the search space. More formally, we note that node x with block size $p1xq1$ can pass individuals to node y with block size $p2 \times q2$ if and only if

$$p_1 = j \times p_2, q_1 = k \times q_2 \tag{15.3}$$

where j, k are integers and $j, k \geq 1$. This establishes a hierarchy of exchange, where node x (lower resolution) is the child of node y (higher resolution) and node y is the parent of node x.

15.4.1.3 iiGA Advantages

iiGA's have the following advantages over other PGA's:

• Building blocks of lower resolution can be directly found by search at that resolution. After receiving lower resolution solutions from its parent node(s), a node of higher resolution can "fine-tune" these solutions.

[1] This is not the same as the dynamic parameter encoding (DPE) feature of GAucsd 1.4 [Schraudolph and Grefenstette 1992].

• The search space in nodes with lower resolution is proportionally smaller. This results in finding "fit" solutions more quickly, which are injected into higher resolution nodes for refinement.

• Nodes connected in the hierarchy (nodes with a parent-child relationship) share portions of the same search space, since the search space of parent is contained in the search space of child. Fast search at low resolution by the parent can potentially help the child find fitter individuals.

• iiGA's embody a divide-and-conquer and partitioning strategy which has been successfully applied to many problems. Homogeneous PGA's cannot guarantee such a division since crossover and mutation may produce individuals that belong to many subspaces –i.e., the divisions cannot be maintained. In iiGA's, the search space is fundamentally divided into hierarchical levels with well defined overlap (the search space of the parent is contained in the search space of the child). A node with block size $r = p' \times q'$ only searches for individuals separated by Hamming distance r.

• In iiGA's, nodes with smaller block size can find the solutions with higher resolution. Although DPE [Schraudolph and Grefenstette 1992] and ARGOT [Shaefer 1987] also deal with the resolution problem using zoom or inverse zoom operators, they are different from iiGA's. First, they are working on the phenotype level and only for real-valued parameters. iiGA's divide the string into small blocks regardless of the meaning of each bit. Second, the sampling error can fool them into prematurely converging on sub-optimal regions. Unlike PDE and ARGOT, iiGA's search different resolution levels in parallel and reduce the risk of searching the wrong target interval.

15.4.2 Speedup and Super Linear Speedup

We would like to take a brief aside concerning the term "speedup" and what it means to GA/GP researchers. Speedup is a term from parallel processing that indicates the amount of time gained by running an algorithm on many processor versus one processor[Amdahl 1967; Quinn 1987]. Thus the best speedup one can obtain is "linear speedup", that is the time n needed to perform the algorithm on a single processor can be reduced to n/p on p processors.

However, it is convenient to measure other forms of effort, especially in areas such as GP. We have often used the "number of evaluations" required, instead of time, as the measure of effort needed to determine the speedup ratio. This avoids

the problems of running an algorithm on different kinds of processors, making the measure hardware independent. Using such a measure we have often been able to find "super-linear" speedups, which on the surface appears impossible (how can we get more speedup than a ratio of the number of processors used).

We can explain super-linear speedup for coarse-grain parallel GPs by dividing the speedup into two parts. The first part is the speedup provided by running the GP on multiple processors. Running a GA or a GP on multiple processors is in fact rather simple, so simple in fact that it is often called "embarrassingly" parallel. For example we can divide the evaluation function evaluations of the population across multiple processors (micro-grain), or as in this chapter divide one large population into multiple smaller populations on multiple processors (coarse-grain). All of these approaches give linear (or very nearly linear depending on the particular application) speedup. The second speedup comes from the number of evaluations saved by running multiple subpopulations vs a single population. We and others have shown that the interaction of multiple, small subpopulations of GAs requires fewer evaluations to reach the same quality of answer than a single large population [Lin et. al. 1994]. These two pieces together, the speedup from multiprocessing and the speedup from small, interacting subpopulations, is where we get our super-linear speedups.

It is for this reason that we often conduct experiments on a single processor with multiple subpopulations, to see if we indeed get the multi-population speedup since we know it is relatively simple to get the multiprocessor speedup.

15.5 Parallel GPs

Our goal is to see if the coarse-grain parallel processing techniques described in Section 15.4 apply in the realm of PGP's. In particular, we wished to investigate the effects various topologies, such as a ring topology and our injection architecture topology would have on a PGP as compared to a similarly sized single population on both the ant and the royal tree problem.

The parallel runs used the same parameters as the single population runs with the following differences. The ring architecture consisted of 7 subpopulations of 500 each (total 3500, same as the single population runs), exchanging the 5 best solutions to its single neighbor every 10 generations (i.e., $1 \rightarrow 2 \rightarrow \cdots \rightarrow 7 \rightarrow 1$.) The injection architecture was set up as follows. We created a hierarchical arrangement of 7 subpopulations of 500, where 6 of the subpopulations were the leaves of a tree to the root 7th subpopulation. The 6 leaf subpopulations "injected" their 5 best solutions (total of 30) to the seventh root subpopulation every 10 generations. There

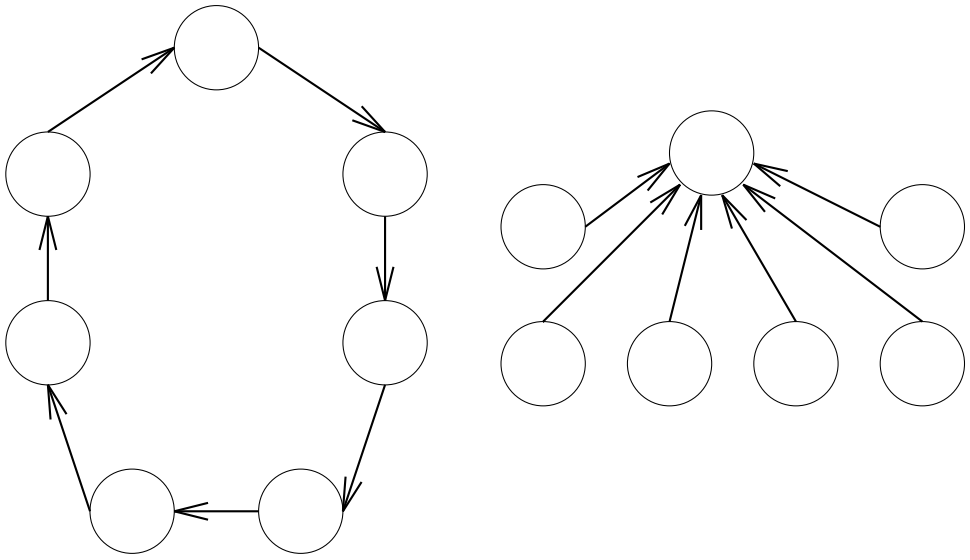

Figure 15.5:
The Parallel Topologies Used for the Parallel Processing Experiments

is no communication between the six leaf subpopulations, only a one-way injection of information ($[1, \ldots, 6] \rightarrow 7$). In all cases the new solutions introduced into a subpopulation replace the *worst* individuals in that subpopulation. Thus if 5 new solutions are introduced, we chose to remove the 5 worst solutions, though lilgp does allow us to use any selection method we choose. The two configurations are shown in Figure 15.5.

Strictly speaking, this is not an injection architecture in the sense we have just previously described, as we are not using a different representation for each of the 6 leaf subpopulations. However, each of the leaves is starting with a different random seed and therefore beginning in different parts of the search space.

Table 15.2shows the results of the parallel runs under the same variation in conditions used in the single population runs.

15.5.1 Discussion of Multi-population Results

Clearly the most surprising aspect of these results was the fact that *no* multi-population approach found a *single* correct answer for the royal tree problem out

Table 15.2:
Multi-population Results. The first two columns use mutation, the second two do not

Ring Architecture Multi-population Runs

Problem	$P_c = 0.9, P_r = 0.1, P_m = 0.0$		$P_c = 0.875, P_r = 0.075, P_m = 0.05$	
	Over-selection	Prop. Selection	Over-selection	Prop. Selection
ant	$W : (4, 160)$	$W : (7, 286)$	$W : (6, 208)$	$W : (7, 240)$
	$L : (12, 68, 312)$	$L : (9, 71, 257)$	$L : (10, 74, 313)$	$L : (9, 73, 244)$
royal tree	$W : (0, 0)$	$W : (0, 0)$	$W : (0, 0)$	$W : (0, 0)$
	$L : (16, 10005, 338)$	$L : (16, 83, 62)$	$L : (16, 16284, 373)$	$L : (16, 76, 181)$

Injection Architecture Multi-population Runs

Problem	$P_c = 0.9, P_r = 0.1, P_m = 0.0$		$P_c = 0.875, P_r = 0.075, P_m = 0.05$	
	Over-selection	Prop. Selection	Over-selection	Prop. Selection
ant	$W : (2, 297)$	$W : (8, 270)$	$W : (2, 116)$	$W : (6, 309)$
	$L : (14, 70, 326)$	$L : (8, 70, 272)$	$L : (14, 70, 304)$	$L : (10.74.256)$
royal tree	$W : (0, 0)$	$W : (0, 0)$	$W : (0, 0)$	$W : (0, 0)$
	$L : (16, 20764, 395)$	$L : (16, 81, 152)$	$L : (16, 18354, 405)$	$L : (16, 83, 192)$

of 128 runs. There was however a large difference in the performance of these runs. The over-selected royal trees average best-of-run fitness was very high relative to the proportional cases, though mutation did not appear to play as significant a role as it did in the single population situations. For the ant problem, parallel processing did not improve performance *except* for the case of proportional selection, where without mutation there was an increase in performance over the single population.

We were surprised by these results since in our GA applications we had always seen an increase in performance with coarse-grain parallelism. However, not only did we *not* observe a performance gain, but also the royal tree coarse-grain parallelism was *apparently detrimental* to performance. When we look at an example ant problem result under injection, we see in Figure 15.5 the kind of performance we expected and the kind we had seen in previous GA applications. Subpopulation 7, the injected subpopulation, shows and increase in performance, presumably due to the injection of improved building blocks from the other subpopulations. However, we did not observe the kind of performance of Figure 15.5 more that 50% of the time in the ant problem, and we *never* saw such performance on the royal tree problem.

15.5.2 Variations on Exchange Rate Experiments

In an unpublished tech report, Koza and Andre [Koza and Andre 1995] reported on parallel GP's implemented on a 64 node transputer using a kind of "fine grain"

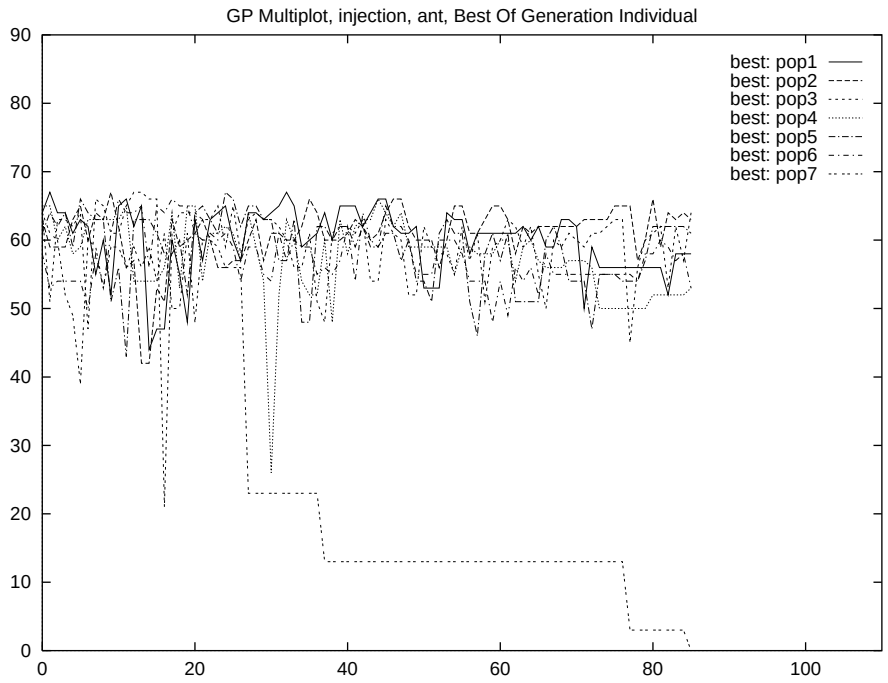

Figure 15.6:
Fitness (y-axis) vs. Generations of a Successful Ant Problem using the Injection Architecture

parallelism. Here, the nodes were laid out in an 8×8 array, and exchange was done between a node and its local 4 neighbors. Koza and Andre also did a high level of exchange, exchanging 20% or more of the populations on each exchange. We had previously avoided such high exchange rates in our GA work since high rates approach a panmictic population, preventing us from investigating the effect of coarse-grain parallelism. However, Koza and Andre achieved "super-linear" speedup (see Section 15.4.2) with 64 (in an 8x8 grid configuration, with exchange between the 4 neighbors) transputer processors. This means that they are apparently getting multi-population speedup. We therefore conducted a series of experiments using a variety of exchange rate to see if we could also find this multi-population speedup component. We used a base configuration of the best previous multi-population results (over-selection, mutation) and exchange every 10 generations. For the ring architecture we compared exchanges of 10, 50 and 100 individuals per exchange.

Table 15.3:
Increased Exchange Results using Over-selection and Mutation

Ring Architecture

Problem	Over-selection, $P_c = 0.875$, $P_r = 0.075$, $P_m = 0.05$		
	Exchange 10	Exchange 50	Exchange 100
ant	$W : (6, 208)$	$W : (4, 207)$	$W : (5, 192)$
	$L : (10, 74, 313)$	$L : (12, 75, 328)$	$L : (11, 72, 304)$
royal tree	$W : (0, 0)$	$W : (2, 437)$	$W : (2, 477)$
	$L : (16, 16284, 373)$	$L : (14, 17100, 316)$	$L : (14, 15842, 399)$

Injection Architecture

Problem	Over-selection, $P_c = 0.875$, $P_r = 0.075$, $P_m = 0.05$		
	Exchange 5	Exchange 20	Exchange 40
ant	$W : (2, 116)$	$W : (3, 171)$	$W : (4, 157)$
	$L : (14, 70, 304)$	$L : (13, 69, 287)$	$L : (12, 68, 333)$
royal tree	$W : (0, 0)$	$W : (0, 0)$	$W : (0, 0)$
	$L : (16 : 18354, 405)$	$L : (16, 20157, 390)$	$L : (16, 29045, 427)$

For the injection architecture, we compared 5, 20 and 40 individuals injected from *each* leaf subpopulation into the root subpopulation. These results are shown in Table 15.3.

15.5.3 Discussion of Exchange Experiments

Though some aspects of the performance improved by increasing the exchange rate, especially for the ant problem, the gain was not nearly as much as was needed to equal the single population performance. The ant under a ring architecture typically had about 5 wins, but did show some slow improvement under increasing exchange rates for an injection architecture. The royal tree finally did find some solutions using increased exchange in a ring architecture, but again no solutions were found using an injection architecture. However, note that the "failure" value in the losses of the injected royal tree again steadily improved under exchange rate increase. In fact, examination of the injected royal-tree solutions shows that indeed the "failed" solutions are rather close. Many have the `e` root as required, as well as two or three full level-d tree subtrees hanging off the `e`.

15.6 Overall 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 over-selection. The royal tree did well with a single population, over-selection, and mutation, but worse under all parallel conditions, though the average loss-scores in the parallel runs did improve. However, it is not clear what we can deduce from changes of the losses since this change does not in any way indicate that the average score for ALL the runs increases as you effectively removed the winning runs that would have helped this average score. More interesting are the graphs of multiple population progress under injection. Here, the ant can show (but does not always show) a dramatic improvement in the injected population's *average* fitness, while the royal tree does not clearly show such an effect. Our hypothesis are twofold: first, there is a "stair-step" 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. Second, the fact that the royal tree is looking for a *single* correct answer belies the fact that the parallel operations did increase the average best-fitness value for failed runs. Again, upon examination of those missed answers it was clear that the subpopulations were indeed making progress and getting close to the answers (for example, an e root with 3 of the 5 level-d children). However, they did not perform *better* than the single populations.

Moreover, further research on increased exchange rates following the work of Koza and Andre showed some improvement of performance, but not up to the level of a single large population. 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.

References

Amdahl, G. (1967), "Validity of the Single processor Approach to Achieving Large Scale Computing Capabilities," *Proceedings AFIPS Conference*, Vol. 30, pp.483-485, Thompson Books, Washington, D.C.

Jones, T (1994). "A Description of Holland's Royal Road Function," *Evolutionary Computation*, 2(4), pp. 409-415

Koza, J.R. and D. Andre (1995) "Parallel Genetic Programming on a Network of Transputers." Stanford Tech Report STAN-CS-TR-95-1542, January.

Koza, J.R.(1992) *Genetic Programming*. Bradford/MIT Press.

Levenick, J.R. (1995) "Metabits: Generic endogenous crossover control," *Sixth International Conference on Genetic Algorithms*, pp 88-95, Morgan Kaufmann

Lin, S.-C., W. F. Punch, and E. D. Goodman (1994) "Coarse-grain parallel genetic algorithms: Categorization and new approach." *Sixth IEEE SPDP*, pp 28–37, October.

Manderick B., and P. Spiessens (1989), "Fine-Grained Parallel Genetic Algorithms," *Third International Conference on Genetic Algorithms*, pp. 428-433, June.

Muhlenbein, H. (1989) "Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization," *Third International Conference on Genetic Algorithms*, pp. 416-421, June.

Punch, W. F., R. C. Averill, E. D. Goodman, S.-C. Lin, and Y. Ding (1995) "Design using genetic algorithms—some results for composite material structures." *IEEE Expert*, 10(1), pp 42-49, February.

Quinn, M.J. (1987) *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, NY.

Shaefer, C. G. (1987), "The ARGOT Strategy: Adaptive Representation Genetic Optimizer Technique," *Proceedings. Second ICGA*, pp. 50-55, July.

Schraudolph, N. and J. Grefenstette (1992), "A User's Guide to GAucsd 1.4," July.

Starkweather, T., D. Whitley and K. Mathias (1991), "Optimization Using Distributed Genetic Algorithms," *PPSN*, pp. 176-185.

Tackett, W. A. (1994) "Recombination, Selection, and the Genetic Construction of Computer Programs." PhD thesis, University of Southern California, April.

Tanese, R (1989) "Distributed Genetic Algorithms," *Third International Conference on Genetic Algorithms*, pp. 434-440, June.