Chapter 1

RUNNING GENETIC PROGRAMMING BACKWARDS

Riccardo Poli and William B. Langdon
Department of Computer Science, University of Essex, UK

Abstract Backward chaining evolutionary algorithms (BC-EA) offer the prospect of run-time efficiency savings by reducing the number of fitness evaluations without significantly changing the course of genetic algorithm (GA) or genetic programming (GP) runs. "Tournament selection, iterated coupon-collection problem, and backward-chaining evolutionary algorithm", Poli, *FOGA*, 2005, describes how BC-EA does this by avoiding the generation and evaluation of individuals which never appear in selection tournaments. It suggests the largest savings occur in very large populations, short runs and small tournament sizes, and shows some of the actual savings in fixed-length binary GAs. Here we provide a generational GP implementation, including mutation and two offspring crossover, of BC-EA and empirically investigate its efficiency, in terms of both fitness evaluations and effectiveness.

Keywords: Genetic programming, tournament selection, efficient algorithms, backward chaining

1. Introduction

Due to its simplicity and efficiency, particularly for large populations, tournament selection is currently the most popular form of fitness selection in GP. The average number of tournaments per generation depends upon whether crossover generates one or two children. With non-overlapping populations of size M and if crossover produces one child from two parents the expected number of tournaments needed to form a new generation is $M(1+p_c)$ (p_c is the crossover probability). However, if each crossover produces two offspring then only (and exactly) M tournaments are needed.

Here we focus on genetic programming with two-offspring crossover.[1] So, if n is the tournament size, creating a new generation requires drawing exactly nM individuals uniformly at random (with resampling) from the current population. As we highlighted in (Poli, 2005), an interesting side effect of this process is that not all individuals in a particular generation are necessarily sampled. This is particularly true where tournament groups are small. For example, $n = 2$.

Except in special cases (such as elitism), the individuals that do not get sampled by the selection process have no influence whatsoever on future generations. However, these individuals use up resources, especially CPU time. So, one might wonder whether it is possible to avoid generating such individuals and what sort of saving one could obtain.

In (Poli, 2005) we provided a theoretical analysis based on Markov chains of the sampling behaviour of tournament selection which started to show the savings. In addition, it described a general scheme, *Backward-Chaining Evolutionary Algorithms* (BC-EA), which exploits the sampling deficiencies of tournament selection to reduce (or make better use of) the fitness evaluations in an EA. (Poli, 2005) suggests the greatest benefits of backward chaining EAs come with very large populations, short runs and relatively small tournament sizes. These are the settings used frequently in genetic programming, particularly when attacking large real-world problems. So, a backward-chaining GP system would appear to have a great potential.

The next section provides a review of previous relevant work, including the main findings of (Poli, 2005). The third section describes the implementation and its time and space complexity of our backward chaining GP system. Section 4 experimentally compares its performance and behaviour with standard GP. We conclude with Section 5.

2. Background

One of the main lines of research on selection in EAs has been into *loss of diversity*, i.e. the proportion of individuals of a population that are not selected. In (Blickle and Thiele, 1997, Motoki, 2002) different selection methods, including tournament selection, were analysed in depth mathematically.

It is important to understand the difference between *not selecting* and *not sampling* an individual in a particular generation. *Not selecting* refers to an individual that did not win any tournaments. This is exactly what research on the loss of diversity has concentrated on. *Not sampling*, instead, refers to an individual that did not participate in any tournament at all, simply because it was not sampled during the creation of the required tournament sets. (Poli, 2005) and this paper focus on individuals which are not sampled.

[1] We have considered the one-offspring case in (Poli, 2005, Poli and Langdon, 2005).

(Sastry and Goldberg, 2001) show cases where the performance of a GA using a particular version of tournament selection (which guarantees that all individuals in a run are sampled) is better than a GA with standard tournament selection. Similar results have been recently reported in (Sokolov and Whitley, 2005) which proposes a different tournament strategy, that also guarantees that all individuals are sampled. While these two lines of work concentrate on modifying tournament selection, we focus on understanding and exploiting the sampling behaviour of standard tournament selection.

Tournament Selection and the Coupon Collector

In (Poli, 2005) a connection between tournament selection and the coupon collection problem was proposed and analysed. In the coupon collection problem (Feller, 1971) every time a collector buys a certain product, a coupon is given to him. The coupon is equally likely to be any one of N types. In order to win a prize, the collector must have at least one coupon of each type.

How is the process of tournament selection related to the coupon collection problem? We can imagine that the individuals in the current population are distinct coupons and that tournament selection will draw (with replacement) nM times from this pool of coupons. Results on the coupon collector problem tell us that if $n < \log M$ there may be a substantial number of individuals that selection did not sample. That is, for small tournament sizes or large populations, many individuals will not be sampled.

In (Poli, 2005) we found that the expected number of distinct individuals sampled by tournament selection in one generation is approximately $M(1 - e^{-n})$. So, for $n = 2$ we should expect about 13.5% of the population not to be sampled. For $n = 3$ this drops to 5%, and becomes quickly negligible for larger tournament sizes. This suggests that saving computational resources by avoiding the creation and evaluation of individuals which will not be sampled by the tournament selection process may be possible only for small tournament sizes. However, low selection pressures are quite common in GP practice, particularly when attacking hard, multi-modal problems which require extensive exploration of the search space before zooming in on any particular region. Also, much greater savings in computation are possible if we exploit the transient behaviour of tournament selection *over multiple generations*.

To understand what happens over multiple generations, let us imagine we are interested in knowing the genetic makeup and fitness of m_0 individuals in a particular generation, G. Clearly, in order to create such individuals, we will need to know who their parent(s) were. On average, this will require running m_0 tournaments to select such parents. In each tournament we pick n individuals randomly from generation $G-1$. After, nm_0 such trials we will be in a position to determine which individuals in generation $G - 1$ will have an influence on

generation G.[2] Let m_1 be the number of individuals sampled. We can now perform nm_1 trials to determine which individuals in generation $G-2$ (the new coupon set) will have an influence on future generations. Let m_2 be their number. The process continues until we reach the initial random generation.

The quantities m_t for $t = 0, 1, ...$ are stochastic variables. Their probability distributions are necessary in order to evaluate the sampling behaviour of tournament selection over multiple generations. In (Poli, 2005) we analysed this process by defining and studying a new and more complex form of coupon collection problem: the iterated coupon collection problem. We modelled the iterated effects of tournament selection as a Markov chain and we showed that under very mild conditions the transition matrix for the chain is ergodic. Therefore, the probability distributions of m_t converge (roughly exponentially) towards a limit distribution which is independent from the initial conditions and, so, the expected value of m_t converges to a constant value.

In other words for long runs (i.e. large G) the number of individuals required in the final generation, m_0, makes almost no differences to the total number of individuals sampled by tournament selection. However for short runs the transient of the Markov chain is what one needs to focus on. Both are given by Markov chain theory but one needs to be able to numerically compute the eigenvalues and eigenvectors of the transition matrix.

Efficient tournament selection and backward chaining EAs

From a practical perspective, the question is: how can we modify an EA to achieve a computational saving from not evaluating and creating individuals not sampled by selection? The idea proposed in (Poli, 2005) is to reorder the different phases of an EA. These are: a) the choice of genetic operator to use to create a new individual, b) the formation of a random pool of individuals for the application of tournament selection, c) the identification of the winner of the tournament (parent) based on fitness, d) the execution of the chosen genetic operator, and e) the evaluation of the fitness of the resulting offspring.[3]

The genetic makeup of the individuals is required only in phases (c), (d) and (e), but not (a) and (b). So, it is possible to change the order in which we perform these phases without affecting the behaviour of our algorithm. For example, we can first iterate phases (a) and (b) as many times as needed to create a full new generation (of course, memorising all the decisions taken), and then iterate phases (c)–(e).[4]

[2] The other individuals in generation $G-1$ have not been sampled and so cannot contribute. Of course only the winners of tournaments pass their genetic material to generation G.

[3] Phases (b) and (c) are repeated once for mutation and twice for crossover. That is as many times as the arity of the genetic operator chosen in phase (a).

[4] (Teller and Andre, 1997) used a similar idea to speed up (but not reduce!) GP fitness evaluations.

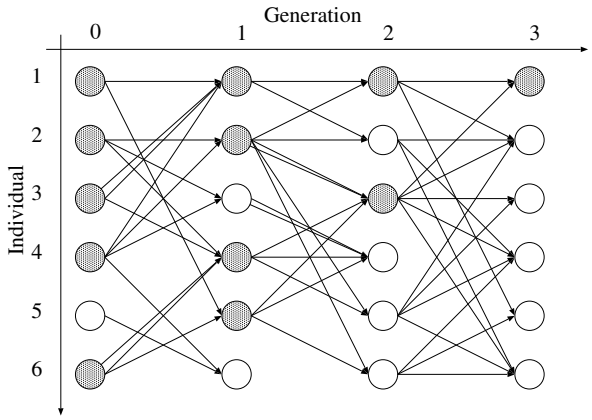

Figure 1-1. Example of graph structure induced by tournament selection in a population of $M = 6$ individuals, run for $G = 3$ generations, using binary tournaments ($n = 2$) and crossover rate $p_c = 1/3$. Nodes with four incoming links were created by crossover. The remaining nodes were created by either mutation or reproduction. Shaded nodes are the potential "ancestors" involved in the creation of the first individual in last generation.

In fact, one can even go further. If we fix in advance the maximum number of generations G we are prepared to run our EA, then phases (a) and (b) (random choices of genetic operations and who will be in which tournament) can be done, not just for one generation, but for a whole run. Then we iterate phases (c)–(e) as required.

We can view the selection of genetic operations and tournament members (phases (a) and (b)) during the whole run, as producing a graph structure containing $(G + 1)M$ nodes. The nodes represent the individuals to be created during the run and the edges connect each individual to the individuals which were involved in the tournaments necessary to select its parents (see Figure 1-1). Nodes without outgoing nodes are not sampled by tournament selection.

If we are interested in calculating and evaluating m_0 individuals in the population at generation G, maximum efficiency can be achieved by considering (flagging for evaluation) only the individuals which are directly or indirectly connected with those m_0 individuals. For example, if in Figure 1-1 we were interested only in the first individual in the last generation, we would need to create and evaluate only that individual and its potential ancestors (shown with shaded nodes). The possible ancestors of our m_0 individuals can be found with a trivial connected-component graph algorithm. Once the relevant sub-graph is known, we evaluate the individuals in it from generation 0 to generation G.

The graph induced by tournament selection can be created without the need to know either what each individual (node) represents or its fitness. So, one might ask whether the construction and the evaluation of the individuals in the

sub-graph should simply be performed in the usual (forward) way, or whether it may be possible and useful to instantiate the nodes in some different order. In (Poli, 2005) it was proposed to recursively proceed backwards.

Here is the basic idea. Let us suppose we are interested in knowing the makeup of individual i in the population at generation G. In order to generate i we only need to know what operator to apply to produce it and which parents to use. In turn, in order to know which parents to use, we need to perform tournaments to select them. In each such tournaments we will need to know the relative fitness of n individuals from the previous generation (which of course, at this stage we may still not know). Let $S = \{s_1, s_2, \dots\}$ be the set of the individuals that we need to know in generation $G-1$ in order to determine i. If we don't know the makeup of these individuals, we can recursively consider each of them as a subgoal. So, we determine which operator should be used to compute s_1, we determine which set of individuals at generation $G-2$ is needed to do so, and we continue with the recursion. When we emerge from it, we repeat the process for s_2, etc. The recursion can terminate in one of two ways: a) we reach generation 0, in which case we can directly instantiate the individual in question by invoking the initialisation procedure, or b) the individual for which we need to know the genetic makeup has already been evaluated before. Once we have finished with i we repeat the same process for any other individuals of interest at generation G, one by one.

This algorithm is effectively a recursive depth-first traversal of the graph induced by tournament selection (c.f. Figure 1-1). While we traverse the graph, as soon as we are in a position to know the genetic makeup of a node encountered we invoke the fitness evaluation procedure. An EA running in this mode is a *Backward-Chaining Evolutionary Algorithm (BC-EA)*.

Irrespectively of the problem being solved and the parameter settings used, because the decisions as to which operator to adopt to create a new individual and which elements of the population to use for a tournament are random, statistically this version of the algorithm is almost identical to a standard EA (see (Poli, 2005)). However, there is an important difference: the *order* in which individuals in the population are evaluated. For example, let us consider the population depicted in Figure 1-1 and suppose we are interested in knowing the first individual in the last generation, i.e. individual $(3,1)$. In a standard EA, we evaluate individuals column by column from the left to the right in the following sequence: $(1,0)$, $(2,0)$, $(3,0)$, $(4,0)$, $(5,0)$, $(6,0)$, $(1,1)$, $(2,1)$, ... until, finally, we reach node $(1,3)$. A BC-EA would instead evaluate nodes in a different order, for example, according to the sequence: $(1,0)$, $(3,0)$, $(4,0)$, $(1,1)$, $(2,0)$, $(2,1)$, $(1,2)$, $(6,0)$, $(4,1)$, $(5,1)$, $(3,2)$, and finally $(1,3)$. So, the algorithm would move back and forth evaluating nodes at different generations.

Why is this important? Typically, in an EA the average fitness of the population and the maximum fitness in each generation grow as the generation number

grows. In the standard EA the first 3 individuals evaluated have an expected average fitness equal to the average fitness of the individuals at generation 0, and the same is true for the BC-EA. However, unlike for the standard EA, the fourth individual created and evaluated by BC-EA belongs to generation 1, so its fitness is expected to be higher than that of the previous individuals. Individual 5 has same expected fitness in the two algorithms. However, the 6th individual drawn by BC-EA is a generation 1 individual again, while the forward EA draws a generation 0 individual. So, again the BC-EA is expected to produce a higher fitness sample than the other EA. This applies also to the 7th individual drawn. Of course, this process cannot continue indefinitely, and at some point the individuals evaluated by BC-EA start being on average inferior.

This behaviour is typical: a BC-EA will find fitter individuals faster than an ordinary EA in the first part of a run and slower in the second part. So, if one restricts oneself to that first phase, the BC-EA is not just faster than an ordinary EA because it avoids evaluating individuals neglected by tournament selection, it is also a faster search algorithm!

3. Backward-chaining GP

Based on these ideas, we have designed and implemented a *Backward-Chaining Genetic Programming* (BC-GP) system in Java. The objective is to evaluate whether the BC-EA approach indeed brings significant efficiency gains in the case of large populations and short runs, and whether a BC-GP compares well with an equivalent standard (forward) version of GP in terms of ability to solve problems.

Backward-chaining GP Implementation

Figure 1-2 provides a pseudo-code description of the key components of our system. The main thing to notice is that we use a "lazy-evaluation" approach. We do not create the full graph structure induced by tournament selection: we statically create the nodes in the graph (and store them using two-dimensional arrays). However the edges are dynamically generated only when needed and stored in the stack as we do recursion. This is achieved by choosing genetic operator and invoking the tournament selection procedure only when needed in order to construct an individual, rather than at the beginning of a run and for all individuals and generations.

Also, note that our implementation is rather simplistic, in that it requires the pre-allocation of three $G \times M$ arrays:

`Population` is an array of pointers to the programs in the population at each generation. Programs are stored as strings of bytes, where each byte represents a primitive.

```
run(G,M):
begin
  Create G x M tables Known, Population and Fitness
  For each individual I of interest in generation G
    evolve_back(I,G)
  return all I of interest
end
```

```
evolve_back(indiv,gen):
begin
  if Known[indiv][gen] then
    return
  if gen == 0 then
    Population[gen][indiv] = random program
  else
    myrand = random_float()
    if  myrand < crossover_rate then
      if myrand < crossover_rate/2 or sibling_pool[gen] = empty then
         parent1 = tournament(gen-1)
         parent2 = tournament(gen-1)
         offsprings = crossover(parent1,parent2)
         Population[gen][indiv] = offspring[1]
         sibling_pool[gen].add(offspring[2])
      else
         Population[gen][indiv] = sibling_pool[gen].remove_random_indiv();
      endif
    else
      parent = tournament(gen-1)
      Population[gen][indiv] = mutation(parent)
    endif
  endif
  Fitness[gen][indiv] = fit_func(Population[gen][indiv])
  Known[gen][indiv] = true
end
```

```
tournament(gen)
begin
  fbest = 0; best = -1
  repeat tournament_size times
    candidate = random integer 1...M
    evolve_back( gen, candidate )
    if Fitness[gen][candidate] > fbest then
      fbest = Fitness[gen][candidate]
      best = candidate
    endif
  endrepeat
  return( Population[gen][best] )
end
```

Figure 1-2. Pseudo-code for backward-chaining GP. Note use of `sibling_pool` for second child produced by crossover.

`Fitness` is an array of single precision floating point numbers. This is used to store the fitness of the programs in `Population`.

`Known` is an array of bits. A bit set to 1 indicates that the corresponding individual in `Population` has been computed and evaluated.

Pre-allocating these arrays is wasteful since only the entries corresponding to individuals sampled by tournament selection are actually used. By using more efficient data structures one could save some memory. BC-GP also uses an expandable array `sibling_pool` to temporarily store the second offspring generated in each crossover.

Space and time complexity of BC-GP

Let us evaluate the space complexity of BC-GP and compare it to the space complexity of standard GP. We divide the calculation into two parts:

$$C = C_{\text{fixed}} + C_{\text{variable}},$$

where C_{fixed} represents the amount of memory (in bytes) required to store the data structures necessary to run GP excluding the GP programs themselves, while C_{variable} represents the memory used by the programs. This can vary as a function of the random seed used, the generation number and other parameters and details of a run.[5] As far as the fixed complexity is concerned, in a forward generational GP system

$$C^F_{\text{fixed}} = 2 \times M \times (4 + 4) = 16M$$

The factor of 2 arises since, in our generational approach, we store both the current and the new generation. This requires 2 vectors of pointers (4 byte each) to the population members and two vectors of fitness values (floats, 4 byte each), where the vectors are of size M. In BC-GP, instead, we need

$$C^B_{\text{fixed}} = G \times M \times (4 + 4 + \frac{1}{8}) \approx 8GM$$

since we need to store one array of pointers, one of floats, and one bit array, all of size $G \times M$.

Variable complexity is harder to compute. In a standard GP system this is

$$C^F_{\text{variable}} \approx 2 \times M \times S^F_{\text{max}},$$

where S^F_{max} is the maximum value taken by the average program size during each generation of a run. In a BC-GP

$$C^B_{\text{variable}} = E^B \times S^B_{\text{avg}},$$

[5]The array *sibling_pool* typically includes only very few individuals and so we ignore it in our calculations.

where $S^B_{\rm avg}$ is the average program size during a BC-GP run (i.e., it is the program size averaged over all individuals created *in a run*) and E^B is the number of programs actually created and evaluated during the run ($E^B \leq E^F = GM$). So, the difference in memory required by the two algorithms is

$$\Delta C = C^B - C^F = M(8G - 16) + E^B \times S^B_{\rm avg} - 2 \times M \times S^F_{\rm max},$$

which indicates that in most conditions the use of BC-GP carries a significant memory overhead. However, this does not prevent the use of BC-GP.[6]

The memory overhead of BC-GP, ΔC, is a function of the average average-program-size $S^B_{\rm avg}$ and the maximum average-program-size $S^F_{\rm max}$. We know that statistically BC-GP and GP behave the same, so we expect $S^F_{\rm max} = S^B_{\rm max}$ and so $S^B_{\rm avg} < S^F_{\rm max}$. An additional complicating factor is that the size of programs often evolves. If *bloat* (Langdon et al., 1999) happens in a particular problem, then programs in both GP and BC-GP will increase in size towards the end of the run. However, since with BC-GP, in certain conditions ($m_0 \ll M$), we evaluate few individuals in the last generations of a run, where bloat is typically most marked, $S^B_{\rm avg}$ can be be a lot smaller than $S^F_{\rm avg}$. That is, with bloat the programs created in a BC-GP may be on average smaller than those created by forward GP. So, we may have $S^B_{\rm avg} \ll S^F_{\rm max}$.

These effects partly mitigate the memory overhead, ΔC, of BC-GP. Also because BC-GP tends to evaluates smaller programs than GP it has an impact on run time too. To see this we need to assess the computational complexity T required to run GP and BC-GP. T is effectively dominated by the cost of running the fitness function. The cost of fitness evaluation depends on various factors, but it is typically approximately proportional to the number of primitives in the program to be evaluated (i.e., executed) and the number of fitness cases N. So, if we express T in number of primitives executed, we have

$$T^F = G \times M \times N \times S^F_{\rm avg}$$

for standard GP, and

$$T^B = E^B \times N \times S^B_{\rm avg}$$

for BC-GP. So, the saving provided by BC-GP is

$$\Delta T = T^F - T^B = N \times (G \times M \times S^F_{\rm avg} - E^B \times S^B_{\rm avg}).$$

That is, for a bloating population the parsimony of BC-GP in terms of fitness evaluations is compounded with its parsimony in terms of program sizes. In some cases (Poli and Langdon, 2005) this leads to considerable savings.

[6]For example, in the worst possible case (where all programs are constructed and evaluated) a BC-GP with a population of 100,000 individuals run for 50 generations and with an average program size (throughout a run) of 100 nodes would require around 540MB of memory.

4. Experimental Results

Test problems and setup

We used BC-GP in a variety of experiments on three continuous symbolic regression problems where the objective was to induce a target function from examples. The target functions were a univariate quartic polynomial, a four variable quadratic polynomial and a 10 variable cubic polynomial. The quartic polynomial is $f(x) = x^4 + x^2 + x^3 + x$. For this problem we used 20 fitness cases of the form $(x, f(x))$ obtained by choosing x uniformly at random in the interval $[-1, +1]$. The first multivariate polynomial, Poly-4, is $f(x_1, x_2, x_3, x_4) = x_1x_2 + x_3x_4 + x_1x_4$. For Poly-4, 50 fitness cases of the form $(x_1, x_2, x_3, x_4, f(x_1, \cdots, x_4))$ were used. They were generated by randomly setting $x_i \in [-1, +1]$. The second multivariate polynomial, Poly-10, is $f(x_1, \cdots, x_{10}) = x_1x_2 + x_3x_4 + x_5x_6 + x_1x_7x_9 + x_3x_6x_{10}$. For Poly-10 we also used 50 fitness cases of the form $(x_1, \cdots, x_{10}, f(x_1, \cdots, x_{10}))$, again each of the ten variables are chosen at random from the range $[-1, +1]$. The function set for GP included the functions $+, -, \times$ and the protected division DIV (if $|y| <= 0.001$ DIV$(x, y) = x$ else DIV$(x, y) = x/y$). The terminal set included the independent variables in the problem (x for Quartic, x_1, x_2, x_3, x_4 for Poly-4 and x_1, x_2, ... x_{10} for Poly-10).

Fitness was calculated as the negation of the sum of the absolute errors between the output produced by a program and the desired output on each of the fitness cases. A problem was considered to be solved if a program with an error of less than 10^{-5} summed across all fitness cases was found. We used binary tournaments ($n = 2$) for parent selection. The initial population was created using the "grow" method with max depth of 6 levels (the root node being at level 0). We used 80% two-offspring sub-tree crossover (with uniform random selection of crossover points) and 20% point mutation with a 2% chance of mutation per tree node. The population size M was 100, 1 000, 10 000 and 100 000. The maximum number of fitness evaluations was was $30M$ (shorter runs were explored in (Poli and Langdon, 2005)). For different experiments, depending on statistical requirements, we performed 100, 1 000 or even 5 000 independent runs of both backward and forward GP.

In symbolic regression problems the fitness of programs in the population, even after a prolonged period of evolution, can be extremely variable. Since the mean is a linear function, the mean population fitness can be seriously changed by individuals with outstandingly poor fitness. So while both algorithms draw, at each generation, individuals from the same distribution the measured means can be different. While observed means are similar in most generations, even averaging over many runs, the mean of means is still sometimes affected by noise injected by poor individuals. In contrast other statistics, e.g. the median

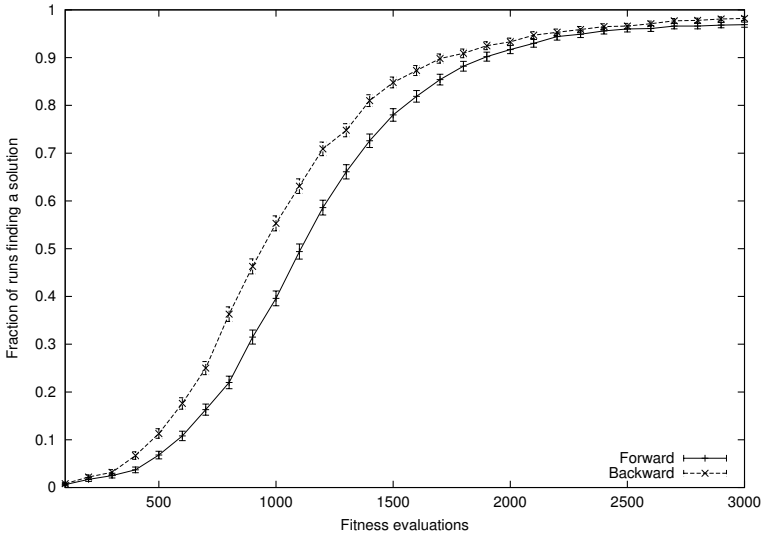

Figure 1-3. Quartic polynomial regression problem. Normal GP contrasted with chance of success with BC-GP (population size 100, average over 1000 runs).

and best, are non-linear and much less effected by the worst in the population. Therefore, we chose to plot the best and the proportion of successful runs.

To make a comparison between the algorithms possible, for BC-GP we computed statistics every M fitness evaluations. We treated this interval as a generation even though the fitness evaluations may be spread over several generations. In the BC-GP we computed 80% of the final generation (i.e. $m_0 = 0.8M$).[7]

Effectiveness and efficiency comparison

Figures 1-3 and 1-4 compare the success probabilities of BC-GP and GP for the quartic polynomial for population sizes 100 and 1 000. The error bars indicate standard error (based on the binomial distribution). As expected BC-GP does better and the difference is statistically significant except for the final generations. With a population of 1 000 (Figure 1-4) or bigger (data not reported), BC-GP is also always statistically better than or equal to standard GP. Naturally, with big populations both forward and backward GP almost always solve the quartic polynomial. Nevertheless BC-GP reaches 100% faster.

The four-variate polynomial, Poly-4, is much harder than Quartic. This is an interesting test case since it requires large populations to be solvable in most runs. Figure 1-5 shows the fraction of successful runs with a population

[7](Poli and Langdon, 2005) reports experiments where we calculated only one individual in the last generation (i.e. $m_0 = 1$).

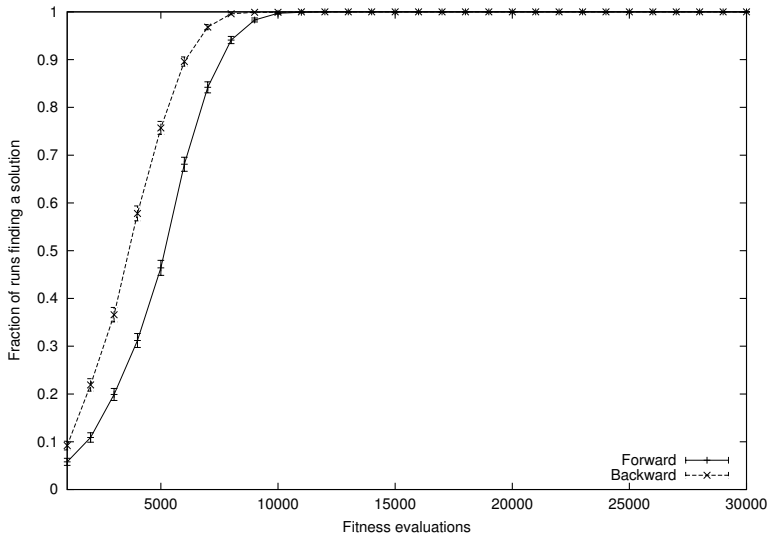

Figure 1-4. Quartic polynomial regression problem. As Fig.1-3 but with population of 1000.

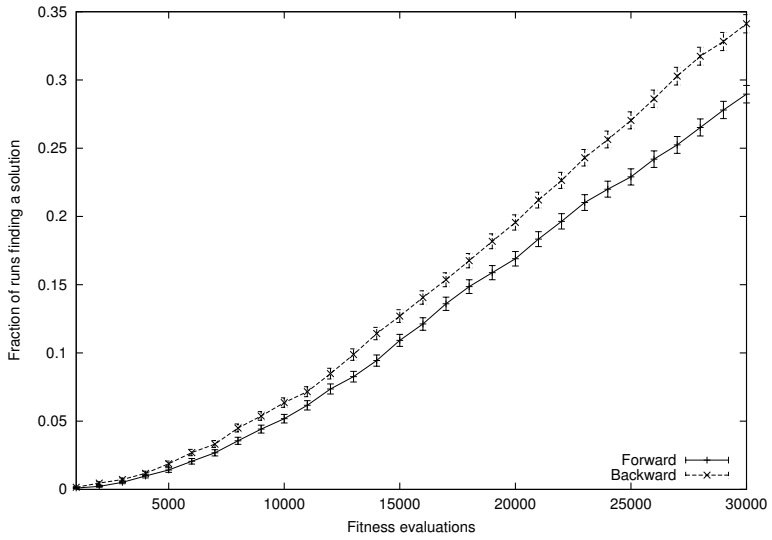

Figure 1-5. Fraction of successful runs (out of 5000 runs) on the Poly-4 problem for forward GP and BC-GP (30 generations) with populations of 1 000.

of 1 000. Figure 1-6 plots similar data but for a population of 10 000. The difference between BC-GP and forward GP is statistically significant for all population sizes used.

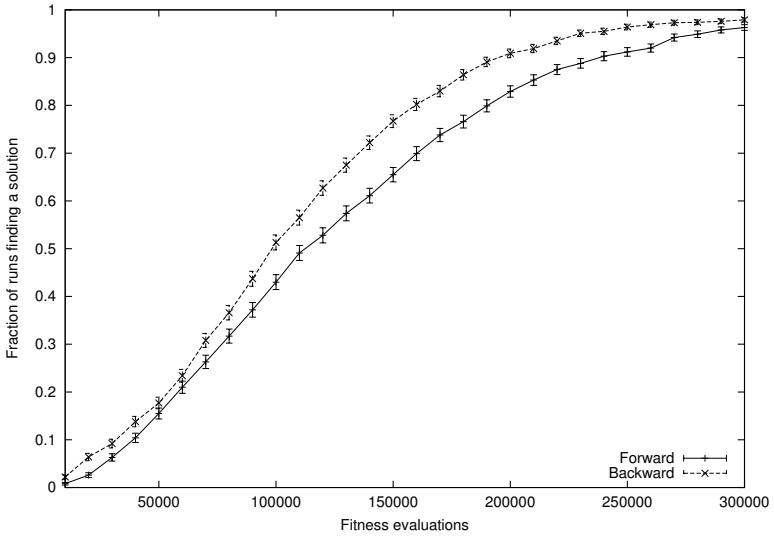

Figure 1-6. Fraction of successful runs (out of 1000 runs) on the Poly-4 problem for forward GP and BC-GP with populations of 10 000.

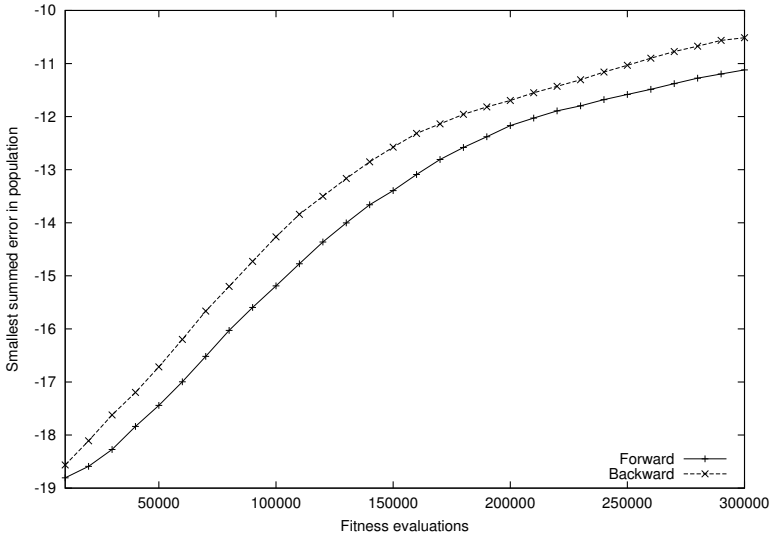

Figure 1-7. Error summed over 50 test cases for Poly-10 regression problem (means of 1000 runs, with populations of 10 000).

Symbolic regression of Poly-10 is very hard. We tried 1000 runs with populations of 100, 1000 and 10 000, and 100 runs with 100 000 individuals. Neither standard GP nor BC-GP found a solution in any of their runs. As illustrated in

Table 1-1. Normal GP v. Backward chaining on Quartic, Poly 4 and Poly 10. Population 10 000. Generations 30. Means of 1 000 runs.

	Forward			Backward			
Problem	Best Fit	Evals	Succ Prob	Best Fit	Evals	Succ Prob	Saving
Quartic	0.00	300 000	100.0%	0.00	240 321	100.0%	19.9%
Poly-4	0.12	300 000	96.3%	0.16	240 315	96.0%	19.9%
Poly-10	11.12	300 000	0.0%	11.29	240 299	0.0 %	19.9%

Figure 1-7 for the case $M = 10\,000$, BC-GP on average finds better programs for the same number of fitness evaluations.

So far we have compared forward GP and BC-GP when both algorithms are given the same number of fitness evaluations. In Table 1-1, we show a comparison when they are run for the same number of generations ($G = 30$). Thanks to the savings obtained by avoiding to create and evaluate individuals not sampled by selection (and any of their unnecessary ancestors), by the end of the runs, BC-GP evolved solutions of similar fitness but took around 20% fewer fitness evaluations. Similar savings are obtained at all population sizes.

The tests mentioned above have been performed also for the case of tournament size $n = 3$. In all cases BC-GP was superior, but by a smaller margin.

5. Conclusions

We exploited a recent theoretical analysis (Poli, 2005) of the sampling behaviour of tournament selection over multiple generations to build a new, highly efficient realisation of GP: backward chaining genetic programming (BC-GP). Thanks to its special way of recursively computing programs and fitnesses backward from the last generation to the first, BC-GP offers a combination of simplicity, fast convergence, increased efficiency in terms of fitness evaluations and primitive evaluations, statistical equivalence to a standard GP, reduced bloat and broad applicability. This comes at the cost of an increased memory use.

The BC-GP algorithm is not hard to implement (see pseudo-code in Figure 1-2). Also, BC-GP tends to find better individuals faster irrespective of the value of the tournament sizes n. However, if one wants use tournaments with more than three individuals and to compute a large proportion of the final generation, the computational saving provided by BC-GP may be too limited to be worth the implementation effort and the memory overhead. In applications which require computing only a small number of individuals in a given generation of interest and where a very large population is used, then BC-GP can be fruitfully applied even for large tournament size. For example, with BC-GP, tournament size 7, and a population of a million individuals one could calculate

1 individual at generation 7, 7 individuals at generation 6, 49 individuals at generation 5, etc. Note that this costs less than initialising the population in a forward GP. The information gained by BG-GP in this way could prove very important, for example, in deciding whether to continue a run or not.

In future research we intend to test the new algorithm on other problems and explore possible ways of further improving the allocation of trials and decision making in BC-GP and GP.

Acknowledgements

The authors would like to thank Chris Stephens, Darrell Whitley, Kumara Sastry and Bob McKay for their useful comments.

References

Blickle, Tobias and Thiele, Lothar (1997). A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394.

Feller, William (1971). *An Introduction to Probability Theory and Its Applications*, volume 2. John Wiley.

Langdon, William B., Soule, Terry, Poli, Riccardo, and Foster, James A. (1999). The evolution of size and shape. In Spector, Lee, Langdon, William B., O'Reilly, Una-May, and Angeline, Peter J., editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press.

Motoki, Tatsuya (2002). Calculating the expected loss of diversity of selection schemes. *Evolutionary Computation*, 10(4):397–422.

Poli, Riccardo (2005). Tournament selection, iterated coupon-collection problem, and backward-chaining evolutionary algorithms. In *Proceedings of the Foundations of Genetic Algorithms Workshop (FOGA 8)*.

Poli, Riccardo and Langdon, William B. (2005). Backward-chaining genetic programming. Technical Report CSM 425, Department of Computer Science, University of Essex.

Sastry, K. and Goldberg, D. E. (2001). Modeling tournament selection with replacement using apparent added noise. In *Proceedings of ANNIE 2001*, volume 11, pages 129–134.

Sokolov, Artem and Whitley, Darrell (2005). Unbiased tournament selection. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005)*. ACM.

Teller, Astro and Andre, David (1997). Automatically choosing the number of fitness cases: The rational allocation of trials. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 321–328, Stanford University, CA, USA. Morgan Kaufmann.