Backward-chaining Genetic Programming

Riccardo Poli and William B. Langdon
Department of Computer
University of Essex
UK
{rpoli,wlangdon}@essex.ac.uk

Technical Report CSM-425
ISSN: 1744-8050
March 2005

Abstract

Tournament selection is the most frequently used form of selection in genetic programming (GP). Tournament selection chooses individuals uniformly at random from the population. As noted in [7], even if this process is repeated many times in each generation, there is always a non-zero probability that some of the individuals in the population will not be involved in any tournament. In certain conditions, typical in GP, the number of individuals in this category can be large. Because these individuals have no influence on future generations, it is possible to avoid creating and evaluating them without altering in any significant way the course of a run. [7] proposed an algorithm, the backward chaining EA (BC-EA), to realised this, but provided limited empirical evidence of the actual savings and the experiments were restricted to fixed-length genetic algorithms. In contrast we provide a generational genetic programming implementation of BC-EA and empirically investigate the efficiency in terms of fitness evaluations and memory use and effectiveness in terms of ability to solve problems of BC-GP. Results indicate that large savings can be obtained with this approach.

1 Introduction

In tournament selection, a group of n individuals is chosen randomly uniformly from the current population, and the one with the best fitness is selected (e.g., see [1]). The tournament size n can be used to vary the selection pressure exerted by this method. For large populations, such as those used in genetic programming (GP), tournament selection is amongst the best methods as far as computation load is concerned. For example, it does not require sorting the whole population, like in rank selection. Also, it does not require repeatedly iterating over the elements of the population like roulette-wheel selection. Its efficiency and simplicity are probably the reasons why tournament selection is currently the most popular form of selection in GP.

In a generational GP system with a population of size M, where crossover is performed with probability p_c while mutation is performed with probability $1-p_c$, the average number of selection steps required to form a new generation is $M(1+p_c)$.[1] So, creating a new generation requires drawing $n(1+p_c)M$ individuals uniformly at random (with resampling) from the current population. As highlighted by Poli in [7], an interesting side effect of this process is that not all individuals in a particular generation are necessarily sampled, and this is particularly true for small values of the tournament size n.

In general those individuals that do not get sampled by the selection process have no influence whatsoever on future generations. However, these individuals use up resources, e.g., CPU time, for their creation and evaluation. How many individuals should we expect not to take part in any tournaments? Is it possible to avoid generating such individuals? If so, can we avoid generating some of the parents and ancestors of such individuals, too? What sort of saving could we obtain by not creating any unnecessary individuals in a run?

In [7], a sound theoretical analysis based on Markov chains of the sampling behaviour of tournament selection was provided. This has given some answers the previous questions. In addition, [7] provided a general scheme, the *Backward-Chaining Evolutionary Algorithm* (BC-EA), to exploit the sampling deficiencies of tournament selection and reduce the number of fitness evaluations in an EA. However, [7] provided limited empirical evidence of the actual savings one can obtain from a BC-EA. In particular, the experiments were restricted to fixed-length genetic algorithms and a very limited evaluation of the actual problem-solving ability of BC-EAs as compared to corresponding standard EAs. Finally [7] gave only limited implementation details.

The results in [7] indicate that the greatest benefits of a BC-EA would be obtained when using 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.

Here we empirically investigate a BC-EA implementation of genetic programming in terms of: efficiency of fitness evaluations, memory use and effectiveness in terms of ability to solve problems. Our results (cf. Section 4) indicate that the efficiency gains can be large.

In the next section we provide an extensive review of previous relevant work, describing in detail the main findings reported in [7], since this has inspired the present work. In Section 3 we present our system describing its implementation and evaluating its time and space complexity. In Section 4 we provide extensive experimental evidence to assess the performance and behaviour of this system in comparison to the standard form of GP. Finally, in Section 5 we provide our

[1]This is valid in the typical situation where each genetic operator directly invokes the selection procedure to provide a sufficient number of parents for its application (e.g., twice in case of crossover) and the crossover operator returns one offspring after each application. Extending the calculation to other cases is trivial.

conclusions.

2 Background

One of the main emphases 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 [2, 3, 6] 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. The work on tournament selection reported in [7] focuses on individuals such as this. Because we build on [7], in the next two sub-sections we review the main ideas and relevant results of [7]. Section 2.1 shows for long runs up to 20% of individuals may be neglected in each generation. While for short runs or huge populations this can be up to 40% or more. Section 2.2 gives an algorithm, based on reversing the traditional evaluation order of genetic algorithms, which realises these potential savings.

2.1 Tournament Selection and the Coupon Collector

In [7] a connection between tournament selection and the coupon collection problem was proposed and analysed. In the Coupon Collection problem, 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. It is well known that on average a collector will have to buy $E_N \approx N \log N$ products before he can expect to have a full set of coupons [4].

How is the process of tournament selection related to the Coupon Collection problem? We can imagine that the M individuals in the current population are $N = M$ distinct coupons and that tournament selection will draw (with replacement) $n(1 + p_c)M$ times from this pool of coupons. If $n(1 + p_c)M < M \log M$ there may be a substantial number of individuals that selection did not sample. I.e. for small tournament sizes or large populations, many individuals (regardless of their fitness) will not be sampled.

Using other results on the coupon collection problem, [7] found that the expected number of distinct individuals sampled by tournament selection in one generation is approximately $M(1 - e^{-n(1+p_c)})$. So, assuming $p_c = 0.5$, for $n = 2$ we should expect about 5% of the population to be neglected. For $n = 3$ this drops to 1%, and becomes quickly negligible for bigger values of n. (Bigger savings are possible for a mutation-only algorithm.)

This analysis suggests that saving substantial computational resources by avoiding the creation and evaluation of individuals which will be neglected by the tournament selection process may be possible only for low selection pressures. 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 the search onto any particular region. Also, *much greater* savings in computation are possible if we exploit the transient behaviour of tournament selection *over multiple generations*.

In [7] the sampling effects of tournament selection over multiple generations were analysed by defining and studying a more complex form of coupon collection problem: the iterated coupon collection problem. In the iterated coupon collection problem, the coupon set changes at regular intervals. Initially the collector is given a (possibly incomplete) set of $m(0)$ old coupons. Each old coupon allows the collector to draw μ new coupons. So, he can perform a total of $\mu m(0)$ trials, which will produce a set of $m(1)$ distinct coupons from the new set. The coupon set now changes, and the player performs $\mu m(1)$ trials to gather as many as possible new distinct coupons. And so on.

The iterated coupon collector problem is a model for what tournament selection will do over multiple generations in a generational system. The connection is simple. Suppose we were interested in knowing the genetic makeup and fitness of $m(0)$ individuals in a particular generation, G.[2] These are like the initial set of old coupons given to the player. Clearly, in order to create such individuals, we will need to know who their parent(s) were. On average, this will require running $m(0)(1+p_c)$ tournaments to select such parents. In each tournament we randomly pick n individuals from generation $G-1$ (each distinct individual in that generation is equivalent to a coupon in the new coupon set). After, $nm(0)(1+p_c)$ such trials we will be in a position to determine which individuals in generation $G-1$ will contribute to generation G. Let $m(1)$ be their number. These $m(1)$ individuals are the equivalent of the new set of coupons the collector has gathered. We can now perform $nm(1)(1+p_c)$ trials to determine which individuals in generation $G-2$ (the new coupon set) will contribute to 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. These can be obtained by modelling the iterated coupon collection problem as a Markov chain. This was done in [7] where it was shown that under very mild conditions ($\lfloor n(1+p_c)\rfloor > 1$) the transition matrix for the chain is ergodic, and therefore the probability distributions of $m(t)$ converges towards a limit distribution which is independent from the initial conditions. I.e., after a transient phase, the expected value of $m(t)$ converges to a constant value.

In other words, for large G it is almost irrelevant whether $m(0)$ is as small

[2]Interesting choices for $m(0)$ would include $m(0) = 1$ and $m(0) = M$, while we could think of G as the number of generations we are prepared to run our EA for.

as 1 or as large as M as far as the total number of individuals not sampled by tournament selection is concerned. For small values of G, however, the transient of the chain is what one needs to focus on. Generally, both are provided by the theory but one needs to be able to numerically compute the eigenvalues and eigenvectors of the transition matrix. Unfortunately, when the population is large (e.g. $M > 1000$ or so), this calculation becomes computationally very heavy and errors tend to accumulate significantly. So, in these cases, only empirical investigations can precisely assess behaviour.

Theory and empirical studies in [7] indicated that under some conditions, for long runs, up to 20% of individuals may be neglected in each generation. Furthermore, in runs where the duration of the transient is of the same order of magnitude as the maximum number of generations G, there are conditions when 40% or more individuals are unnecessary.

2.2 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 [7] 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. (Phases (b) and (c) are repeated once for mutation and twice for crossover. I.e. as many times as the arity of the genetic operator chosen in phase (a).)

The genetic makeup of the individuals is required only in phases (d), (c) 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). Indeed, this idea was used in [8] for the purposed on speeding up genetic programming fitness evaluation. The approach there was to estimate the fitness of the individuals involved in the tournaments by evaluating them on a subset of the fitness cases available. On the basis of this estimate, it was often possible to determine which individuals would win most tournaments with a small error probability. These tournaments could therefore be decided quickly, while only in a subset of tournaments individuals ended up being evaluated using all fitness cases.

In fact, one can even go further as proposed in [7]. If we fix in advance the maximum number of generations G we are prepared to run our EA for, phases (a) and (b) can be iterated not just for one generation but for a whole run from generation 0 to generation G. Then we iterate phases (c)–(e) as required.

Clearly the iteration of phases (a) and (b) over multiple generations induces

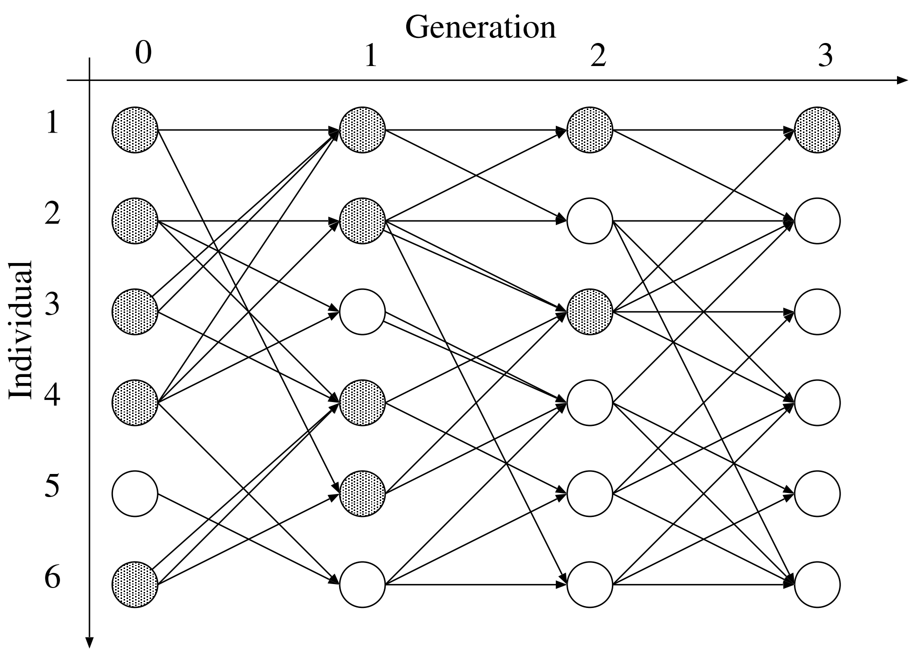

Figure 1: Example of graph structure induced by tournament selection in a population of $M = 6$ individuals run for $G = 3$ generations using a tournament size $n = 2$ and $p_c = 1/3$. Nodes with four incoming links were created by crossover. The remaining nodes were created by either mutation or reproduction.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

a graph structure containing $(G+1)M$ nodes representing all the individuals to be evolved during the run and where edges connect each individual to the individuals which were involved in the tournaments necessary to select the parents of such an individual. For example, Figure 1 shows a possible graph structure induced 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 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), which would produce a saving of 50% fitness evaluations. In general the problem of determining the possible ancestors of our $m(0)$ individuals of interest can be solved with a trivial connected-component algorithm. Once the relevant sub-graph is detected, we can then proceed evaluating only the individuals in the sub-graph, 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 to instantiate the nodes in some different order and there would be any benefits associated with that. In [7] it was proposed to recursively proceed backward.

Here is the basic idea. Let us suppose we are interested in knowing the makeup of individual r in the population at generation G. In order to generate r we only need to know what operator to apply to produce it and what 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 makeup of n individuals from the previous generation (which of course, at this stage we may still not know). Let $S = \{s_1, s_2, \ldots\}$ be the set of the individuals that we need to know in generation $G - 1$ in order to determine r. 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 r 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). 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 EA (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 [7] for a more detailed discussion).

One important difference between the two modes of operation is the *order* in which individuals in the population are evaluated. For example, let us consider the population depicted in Figure 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 backward chaining 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 converge 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 converging algorithm.

3 Backward-chaining GP

Based on the ideas summarised in the previous section, 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.

3.1 Backward-chaining GP Implementation

Figure 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" type of 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) but then edges are only dynamically generated 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
    if random_float() < crossover_rate then
      parent1 = tournament(gen-1)
      parent2 = tournament(gen-1)
      Population[gen][indiv] = crossover(parent1,parent2)
    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 2: Pseudo-code for backward-chaining GP.

`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 its fitness has been calculated.

This is wasteful since, in BC-GP, not all the entries of the arrays are used (only those corresponding to individuals sampled by tournament selection are). So, by using more efficient (albeit complex) data structures one could save some memory.

3.2 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.

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

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

where S^B_{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 space complexity between the two algorithms is

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

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. 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 — an amount of memory readily available in most modern ordinary personal computers.

The memory overhead of BC-GP, ΔC, is a function of the average average-program-size S^B_{avg} and the maximum average-program-size S^F_{max}. We know that statistically BC-GP and GP must behave the same, so we should expect $S^F_{\text{max}} = S^B_{\text{max}}$, and, consequently, also $S^B_{\text{avg}} < S^F_{\text{max}}$. However, because of size changes during the run, we expect $S^B_{\text{avg}} \neq S^F_{\text{avg}}$. If we require BC-GP to compute only very few individuals in the last generation (e.g., just one), then BC-GP will construct many individuals in the first generations of a run, and, typically, very few in the last generations, while a forward GP will always compute equal numbers of individuals at all generations. If *bloat* [5] happens in a particular problem, then both GP and BC-GP will show bloat. However, since with BC-GP we only evaluate very few individuals in the last generations of a run — those where bloat is typically most marked — we should expect to see that S^B_{avg} is significantly smaller than S^F_{avg}. That is, in these conditions the programs created in a BC-GP are on average smaller than those created by GP. So, we should expect $S^B_{\text{avg}} \ll S^F_{\text{max}}$.

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 effort T required to run GP and BC-GP. As is usual in GP, T is effectively determined by the cost of running the fitness function (the costs of recursive calls and array updates in BC-GP are negligible compared to fitness evaluation). 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

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

for standard GP, and

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

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

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

That is, for a bloating population the parsimony of BC-GP in terms of fitness evaluations is compounded with its parsimony in terms of average program sizes. As will be shown in the next section, in some conditions this can lead to considerable computational savings.

4 Experimental Results

4.1 Test problems and setup

We used backward chaining 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 multivariate quadratic polynomial and a multivariate 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 $[-1, +1]$. One multivariate polynomial, Poly-4, is $f(x_1, x_2, x_3, x_4) = x_1x_2 + x_3x_4 + x_1x_4$. For this problem 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 $f(x_1, \cdots, x_{10}) = x_1x_2 + x_3x_4 + x_5x_6 + x_1x_7x_9 + x_3x_6x_{10}$. Also for this problem we used 50 fitness cases of the form $(x_1, \cdots, x_{10}, f(x_1, \cdots, x_{10}))$, which, again, were obtained by randomly setting $x_i \in [-1, +1]$. The function set for GP included the functions $+, -, \times$ and the protected division DIV. I.e. if $|y| <= 0.001$ $\text{DIV}(x, y) = x$ else $\text{DIV}(x, y) = x/y$. 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.

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% standard sub-tree crossover (with uniform random selection of crossover points) and 20% point mutation with a 2% chance of mutation per tree node. The maximum number of generations G was 10, 20 and 30. The population size M, depending on target polynomial, was 100, 1 000, 10 000 and 100 000.

For each setting we performed 1,000 independent runs of both the backward and the standard version of the algorithm applied to the three problems.

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

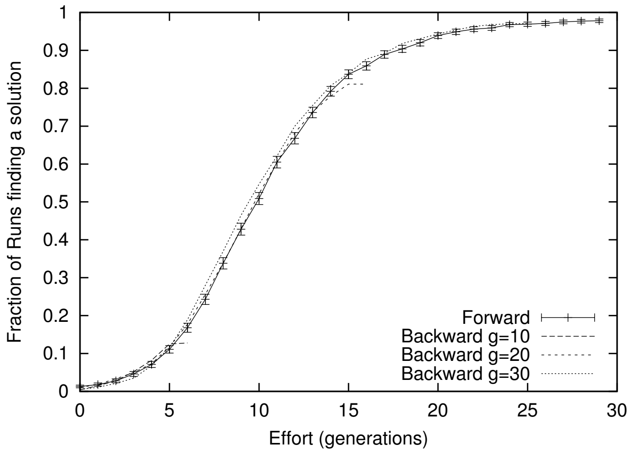

Figure 3: Quartic polynomial regression problem. Normal (forward GP, solid line) contrasted with effort and chance of success with backward chaining GP (generation 10, 20 and 30, population size 100, average over 1000 runs).

noise injected by poor individuals. In contrast other statistics, e.g. the median 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, and we treated this interval as a generation even though the fitness evaluations may be spread over several generations. In the BC-GP we computed only one individual in the last generation (i.e. $m(0) = 1$).

4.2 Effectiveness and efficiency comparison

Figures 3, 4 and 5 show the cumulative number of fitness evaluations required to solve the quartic polynomial. (I.e. to find a program with an error of less than 10^{-5} summed across all fitness cases.) The error bars indicate standard error (based on the binomial distribution). With a small population of 100, as expected, with all three run lengths (10, 20 and 30 generations) backward chaining initially does slightly better. However the difference may not be statistically significant. With a bigger population (1 000, c.f. Figure 4, or 10 000, c.f. Figure 5), however, BC-GP is always better than or equal to standard GP. In all cases, with a population of $M = 10\,000$, for almost all generations, both forward and backward GP almost always solve the quartic polynomial. Nevertheless backward chaining 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 6 shows the fraction of successful runs (out of 1000 runs) on the Poly-4 problem contrasting forward GP (run for 30 generations) against BC-

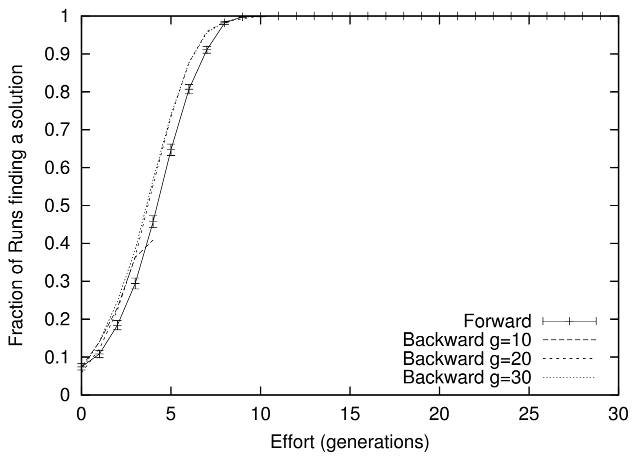

Figure 4: Quartic polynomial regression problem. As Fig.3 but with population of 1000. Note, except near the end of runs backward chaining (dotted lines) consistently gives a higher chance of success for the same number of fitness evaluations.

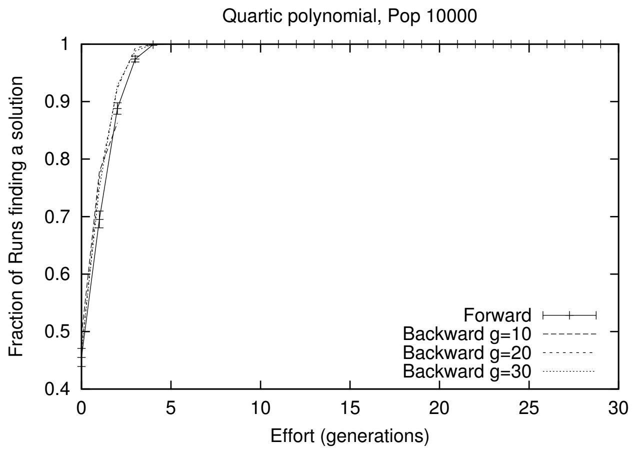

Figure 5: Quartic polynomial regression problem. As Fig.3 but with population of 10 000. Again backward chaining consistently gives a higher chance of success for the same number of fitness evaluations.

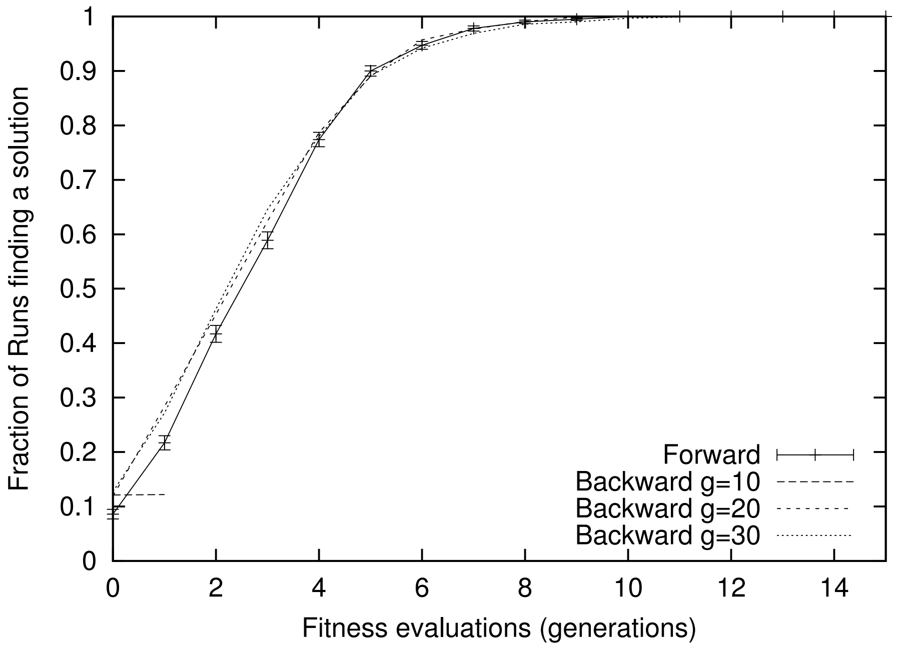

Figure 6: Fraction of successful runs (out of 1000 runs) on the Poly-4 problem for forward GP (30 generations) and BC-GP (10, 20 and 30 generations) with populations of 100 000. Only the first 15 generations are shown for ease of comparison. For $G = 20$ and $G = 30$ backward chaining GP (dotted lines) consistently finds better programs for the same number of fitness evaluations.

GP (run for 10, 20 and 30 generations) when the population includes 100 000 individuals. In 122 (out of 1000) 10-generation runs BC-GP solved the problem almost instantly but other runs made no progress after that. However, when run for 20 and 30 generations, BC-GP found more solutions faster than forward GP for generations 0...3. The difference between the two algorithms is statistically significant. Table 1 shows that, by the end of the runs, the backward chaining GP evolved solutions of similar fitness but took, depending on number of generations, between 33% and 91% fewer fitness evaluations.

Symbolic regression of Poly-10 is very hard. We tried 1000 runs with populations of 100, 1000 and 10 000 and 10, 20 or 30 generations. Neither the standard nor the backward chaining GP found a solution in any of their 9000 runs. Table 2 shows that, by the end of the runs, the backward chaining GP evolved solutions of similar fitness but took, depending on number of generations, between 27% and 75% fewer fitness evaluations. Figure 7 shows in all three cases (10, 20 and 30 generations) except right at the end of runs, the backward chaining GP on average finds better programs for the same effort.

Finally, to assess the behaviour of BC-GP with respect to bloat and to

Table 1: Normal GP v. Backward chaining on Poly 4 problem. Population 100 000. Means of 1000 runs.

	Forward		Backward		
Gens	Best fitness	Evals	Best fitness	Evals	Saving
10	0.030	1 000 000	0.122	89600	91%
20	0	2 000 000	0.001	1 046 000	48%
30	0	3 000 000	0	2 015 000	33%

Table 2: Best-of-run results for normal GP and backward chaining GP on Poly-10 problem. Population 10 000. Means of 1000 runs.

	Forward		Backward		
Gens	Best Fitness	Evals	Best Fitness	Evals	Saving
10	14.5	100 000	15.6	25 000	75%
20	13.6	200 000	14.2	122 000	39%
30	13.0	300 000	13.4	219 000	27%

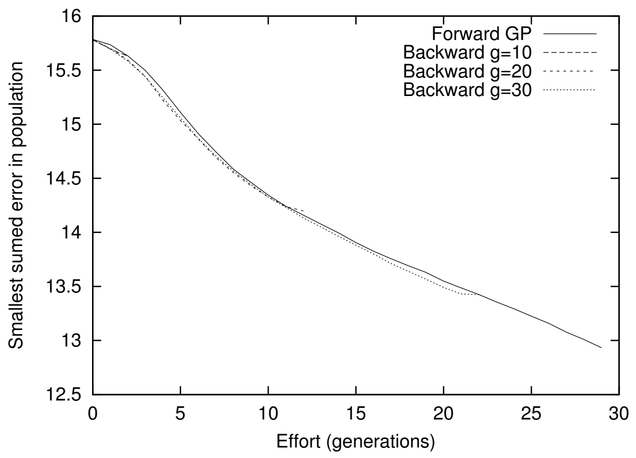

Figure 7: Error summed over 50 test cases for Poly-10 regression problem (means of 1000 runs, with populations of 10 000). Except near the end of runs, backward chaining (dotted lines) consistently finds better programs for the same number of fitness evaluations.

Table 3: Time complexity comparison between normal GP and backward chaining GP on Poly-4 (30 generations). Means of 100 runs. (See Section 3.2 for the meaning of symbols.)

	Forward		Backward	
	M=100	M=10 000	M=100	M=10 000
E	3 000	300 000	2 540	219 000
S_{avg}	16.43	24.12	11.64	10.55
T	2 460 000	362 000 000	1 480 000	116 000 000
Saving			**40%**	**68%**
Solved	2%	94%	3%	93 %

evaluate its effective time complexity we re-run Poly-4 with populations of 100 and 10 000 for 30 generations (performing 100 independent runs in each case) while collecting program size statistics. Table 3 illustrates that GP and BC-GP had essentially the same performance in terms of ability to solve the problem (bottom row), but that average program sizes in BC-GP were substantially smaller. Compounding this with the saving in terms of fitness evaluations (15% for $M = 100$ and 27% for $M = 10\,000$), BC-GP had an average time complexity of between 40% and 68% fewer primitive evaluations.

5 Conclusions

We exploited a recent theoretical analysis [7] 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 only at the cost of an increased memory use. 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.

Acknowledgements

The authors would like to thank Chris Stephens and Darrell Whitley for helpful comments and suggestions.

References

[1] T. Bäck, D. B. Fogel, and T. Michalewicz, editors. *Evolutionary Computation 1: Basic Algorithms and Operators.* Institute of Physics Publishing, 2000.

[2] T. Blickle and L. Thiele. A mathematical analysis of tournament selection. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms (ICGA'95)*, pages 9–16, San Francisco, California, 1995. Morgan Kaufmann Publishers.

[3] T. Blickle and L. Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, 1997.

[4] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 2. John Wiley, 1971.

[5] W. B. Langdon, T. Soule, R. Poli, and J. A. Foster. The evolution of size and shape. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA, June 1999.

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

[7] R. Poli. Tournament selection, iterated coupon-collection problem, and backward-chaining evolutionary algorithms. In *Proceedings of the Foundations of Genetic Algorithms Workshop (FOGA 8)*, 4th January 2005.

[8] A. Teller and D. Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 321–328, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.