

Backward-chaining Genetic Programming

Riccardo Poli
Department of Computer Science
University of Essex, UK
rpoli@essex.ac.uk

William B. Langdon
Department of Computer Science
University of Essex, UK
wlangdon@essex.ac.uk

ABSTRACT

This paper presents a backward-chaining version of GP.

Categories and Subject Descriptors: I.2.2 [Artificial Intelligence]: Automatic Programming

General Terms: Algorithms, Performance

Keywords: GP, tournament selection, backward chaining.

1. INTRODUCTION

Tournament selection, the most frequently used form of selection in genetic programming (GP), chooses individuals uniformly at random from the population. As noted in [1], 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. Starting with a theoretical analysis of the sampling behaviour of tournament selection, in [1] we proposed an algorithm, the backward chaining EA (BC-EA), to do this, but provided limited empirical evidence of actual savings and experiments were restricted to fixed-length genetic algorithms.

We describe a GP implementation of BC-EA and empirically investigated the efficiency in terms of fitness evaluations and memory use and effectiveness in terms of ability to solve problems of BC-GP. A fuller description of the techniques and results reported here is available in [2], while further work with BC-GP is in [3].

2. BACKGROUND

Using results on the coupon collection problem, in [1] we found that the expected fraction of distinct individuals not sampled by tournament selection in one generation is approximately $e^{-n(1+p_c)}$, where n is the tournament size, p_c is the crossover rate and we assume crossover returns one offspring. This suggests that saving computational resources by avoiding the creation and evaluation of individuals neglected by selection may be possible only for low selection pressures. However, low selection pressures are quite common in GP practice. Also, much greater savings in compu-

tation 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 $m(0)$ individuals in a particular generation, G . Clearly, in order to create such individuals, we need to know who their parent(s) are. This requires running tournaments to select such parents from generation $G - 1$. Let $m(1)$ be their number. We can now perform tournaments to determine which individuals in generation $G - 2$ will contribute to generation $G - 1$, etc.

In [1] we showed that the probability distributions of the stochastic variables $m(t)$ converge towards a limit 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 or large 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.

The different phases of an EA 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. The genetic makeup of the individuals is not required in phases (a) and (b). So, as proposed in [1], phases (a) and (b) can be done for a whole run from generation 0 to generation G (followed by the iteration of phases (c)–(e) as required). Clearly this induces a graph structure (see Figure 1) where nodes represent all the individuals to be evolved during the run and edges connect each individual to the individuals in the tournaments performed to select the parents of such an individual.

If we are interested in calculating and evaluating $m(0)$ individuals in the population at generation G , maximum efficiency can be achieved by constructing and evaluating only the individuals which are directly or indirectly connected with those $m(0)$ individuals (e.g., see shaded nodes in Figure 1). In [1] it was proposed to proceed recursively, backwards in time from generation G . An EA running in this mode is a *Backward-Chaining EA (BC-EA)*. Irrespectively of the problem being solved, statistically a BC-EA is almost identical to a standard EA. However, a BC-EA is faster than an ordinary EA because it avoids evaluating individuals neglected by selection. Also, because of its fitness evaluation order, BC-EA is also a faster converging algorithm. (See [1] for a detailed discussion.)

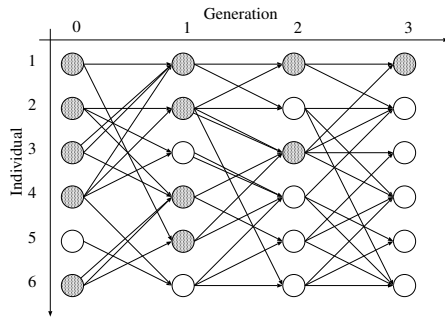


Figure 1: Example of graph structure induced by tournament selection. Shaded nodes are the potential ancestors of the first individual in generation 3.

Table 1: GP v. BC-GP on Poly 4 ($M = 100\,000$).

Gens	Forward		Backward		Saving
	Best Fit	Evals	Best Fit	Evals	
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: GP v. BC-GP on Poly-10 ($M = 10\,000$).

Gens	Forward		Backward		Saving
	Best Fit	Evals	Best Fit	Evals	
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%

3. BACKWARD-CHAINING GP

We implemented a *Backward-Chaining Genetic Programming* (BC-GP) system in Java. Figure 2 provides a pseudo-code description of the key components of our system. Note that we use a “lazy-evaluation” approach: we statically create the nodes in the graph (and store them using 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 selection procedure only when needed in order to construct an individual, rather than at the beginning of a run for all generations.

We ran BC-GP on three symbolic regression problems: the quartic polynomial $x^4 + x^2 + x^3 + x$, the multivariate polynomial $x_1x_2 + x_3x_4 + x_1x_4$ (Poly-4) and the multivariate polynomial $x_1x_2 + x_3x_4 + x_5x_6 + x_1x_7x_9 + x_3x_6x_{10}$ (Poly-10). The function set included $+$, $-$, \times and the protected division DIV. The terminal set included the independent variables in each problem. Fitness was the sum of the absolute errors between the output produced by a program and the desired output. We used binary tournaments ($n = 2$) for parent selection. The initial population was created using the “grow” method. We used 80% standard sub-tree crossover and 20% point mutation. 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 GP and BC-GP. (See [2] for more details.)

We computed statistics every M fitness evaluations (we treat this interval as a generation even if the fitness evaluations may spread over many generations). BC-GP computed only *one individual* in the last generation (i.e. $m(0) = 1$).

On the quartic polynomial, with populations of 1 000 or

```

define run(G,M)
  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
enddefine
define evolve_back(indiv,gen)
  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
enddefine
define tournament(gen)
  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] )
enddefine

```

Figure 2: Pseudo-code for BC-GP.

more individuals, BC-GP was always better than or equal to standard GP (see [2]). Poly-4 is much harder and requires large populations to be solvable in most runs. When run for 20 and 30 generations, BC-GP found more solutions faster than forward GP for generations 0..3 (see [2]). Table 1 shows that, by the end of the runs, BC-GP evolved solutions of similar fitness but took fewer fitness evaluations.

Poly-10 is very hard. Neither GP nor BC-GP found a solution in any of 9 000 runs. Table 2 shows that, by the end of the runs, BC-GP evolved solutions of similar fitness but took fewer fitness evaluations.

4. CONCLUSIONS

Thanks to its special way of recursively computing programs backward, 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.

5. REFERENCES

- [1] 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.
- [2] R. Poli and W. B. Langdon. Backward-chaining genetic programming. Technical Report CSM-425, Department of Computer Science, University of Essex, 2005.
- [3] R. Poli and W. B. Langdon. Running genetic programming backward. In R. L. Riolo, B. Worzel, and T. Yu, editors, *Genetic Programming Theory and Practice*. Kluwer, 2005.