JID:ARTINT AID:2206 /FLA [m3SC+; v 1.60; Prn:26/05/2006; 14:33] P.1 (1-30)

Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Artificial Intelligence ••• (••••) •••–•••
Artificial Intelligence
www.elsevier.com/locate/artint

Backward-chaining evolutionary algorithms

Riccardo Poli *, William B. Langdon

Department of Computer Science, University of Essex, UK

Received 12 August 2005; received in revised form 11 April 2006; accepted 24 April 2006

Abstract

Starting from some simple observations on a popular selection method in Evolutionary Algorithms (EAs)—tournament selection—we highlight a previously-unknown source of inefficiency. This leads us to rethink the order in which operations are performed within EAs, and to suggest an algorithm—the EA with efficient macro-selection—that avoids the inefficiencies associated with tournament selection. This algorithm has the same expected behaviour as the standard EA but yields considerable savings in terms of fitness evaluations. Since fitness evaluations typically dominates the resources needed to solve any non-trivial problem, these savings translate into a reduction in computer time. Noting the connection between the algorithm and rule-based systems, we then further modify the order of operations in the EA, effectively turning the evolutionary search into an inference process operating in backward-chaining mode. The resulting backward-chaining EA, creates and evaluates individuals recursively, backward from the last generation to the first, using depth-first search and backtracking. It is even more powerful than the EA with efficient macro-selection in that it shares all its benefits, but it also provably finds fitter solutions sooner, i.e., it is a faster algorithm. These algorithms can be applied to any form of population based search, any representation, fitness function, crossover and mutation, provided they use tournament selection. We analyse their behaviour and benefits both theoretically, using Markov chain theory and space/time complexity analysis, and empirically, by performing a variety of experiments with standard and back-ward chaining versions of genetic algorithms and genetic programming.

1. Introduction

Evolutionary Algorithms (EAs) (see Algorithm 1) are a simple and, today, very popular form of search and optimisation technique [1,2,6,13,21,22]. Their invention dates back many decades [11,14,18,34,40] (and see also [10]). EAs share several ingredients with mainstream AI search techniques. For example, EAs can be seen as special kinds of generate-and-test algorithms, as parallel forms of beam search, etc. (see [26] for a discussion on similarities and differences between EAs and other search algorithms). However, their development has been largely in parallel and independent from AI search.

Despite the simplicity of EAs, sound theoretical models of EAs and precise mathematical results have been scarce and hard to obtain, often emerging many years after the proposal of the original algorithm [7,15,16,19,24,25,28–31, 36–39,42–44,47]. An important reason for this delay is that each algorithm, representation, set of genetic operators

* Corresponding author.
E-mail addresses: rpoli@essex.ac.uk (R. Poli), wlangdon@essex.ac.uk (W.B. Langdon).

0004-3702/$ – see front matter © 2006 Published by Elsevier B.V.
doi:10.1016/j.artint.2006.04.003

1: Initialise population
2: Evaluate population
3: **loop**
4: Select sub-population for reproduction
5: Recombine the genes of selected parents
6: Mutate the offspring stochastically
7: Evaluate the fitness of the new population
8: If stopping criterion is satisfied then exit loop
9: **end loop**

Algorithm 1. Generic evolutionary algorithm.

and, in some cases, fitness function requires a different theoretical model. In addition, the randomness, non-linearities and immense number of degrees of freedom present in a typical EA make life very hard for theoreticians.

One line of theoretical research where differences in representations have not been an obstacle is the analysis of selection algorithms (step 4 in Algorithm 1). This is because selection requires only knowledge of the fitness (or phenotype) of the individuals in the population, and so the same form of selection can be applied irrespective of the representation of an individual (or genotype).

Different selection methods have been analysed mathematically in depth in the last decade or so. The main emphasis of previous research has been the takeover time [12], i.e., the time required by selection to fill up the population with copies of the best individual in the initial generation, and the evaluation of the changes produced by selection on the fitness distribution of the population [4,5,23]. In this second line of research, the behaviour of selection algorithms is characterised using the loss of diversity, i.e., the proportion of individuals in a population that are not selected.

These theoretical studies are very comprehensive and appeared to have completely characterised selection, fundamentally making it a largely understood process. However, starting from some simple observations on the sampling behaviour of perhaps the most popular selection method, tournament selection, in this paper we show that this is a possible source of inefficiency in EAs. This phenomenon, which had not been analysed in previous research, has very deep implications, its analysis effectively leading to a completely new class of EAs—the backward-chaining EA—which is more powerful and closer in spirit to classical AI techniques than traditional EAs.

The paper is organised as follows. In Section 2 we describe tournament selection, we briefly review previous relevant theoretical results, and then go on to describe, in Section 3, the sampling inefficiency in this form of selection.

In order to remove the predicted sampling inefficiency of tournament selection, in Section 4, we rethink the order in which operations are performed within EAs. This reveals that, embedded in EAs, is a graph-structure induced by tournament selection which connects individual samples of the search space across time. (See Fig. 1 in Section 4.) We are then able to suggest an algorithm, the EA with efficient macro-selection, that exploits this graph to remove the inefficiencies associated with tournament selection. The algorithm has the same expected behaviour as the standard EA, while providing considerable savings in terms of fitness evaluations. Furthermore, it is totally general, i.e., it can be applied to any representation and fitness function, and can be used with any crossover and mutation.

In Section 5, we note an unexpected connection between the operations of the EA with efficient macro-selection and rule-based systems, which leads us to further modify the order of operations in the EA effectively turning the evolutionary search into an inference process operating in backward-chaining mode. The resulting algorithm, which we call a backward-chaining EA, creates and evaluates individuals recursively. It starts at the last generation and, using depth-first search and backtracking, works backwards to the first. This algorithm is even more powerful than the EA with efficient macro-selection in that it shares all its benefits, but it provably finds fitter solutions sooner, i.e., it is a faster algorithm.

We analyse theoretically the behaviour of the EA-EMS and BC-EA algorithms in Section 6. In particular, in Section 6.1 we start analysing the sampling behaviour of tournament selection, focusing on its effects over one time step (a generation) of an EA. We do this by noting and exploiting the similarity between sampling and the coupon collection problem. We extend the one-generation analysis to full runs in Section 6.2 by inventing, and then modelling mathematically using Markov chain theory, a more complex version of the problem—the iterated coupon collection problem—which exactly mimics tournament selection over multiple generations. This allows us to fully and exactly evaluate the effects of the sampling inefficiency of tournament selection over entire runs and indicates that extent of the savings that could be achieved. We discuss the details of the practical implementation of a backward-chaining EA

in Section 7 while we compare the time and space complexity of our implementation with those for a standard EA in Section 8. In Section 9 we provide experimental results with a Genetic Algorithm (GA) and a Genetic Programming (GP) implementation of backward chaining EA. We discuss our findings in Section 10 and provide our conclusions in Section 11.

2. Tournament selection

Tournament selection is one of the most popular forms of selection in EAs. In its simplest form, 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 [2]). The parameter n is called the *tournament size* and can be used to vary the selection pressure exerted by this method (the higher n the higher the pressure to select above average quality individuals).

In a population of size M, the *takeover time* is defined as the number of generations required for selection (when no other operator is present) to obtain a population containing $M - 1$ copies of the best individual in the initial generation [12]. In [12] the takeover time for tournament selection was estimated using the asymptotic expression

$$t^* = \frac{1}{\ln n}\big[\ln(M) + \ln\big(\ln(M)\big)\big]$$

where the approximation improves as the population size $M \to \infty$.

The *loss of* (*fitness*) *diversity* is the proportion of individuals of a population that is not selected during the selection phase. Assuming every member of the population has a unique fitness, the loss of diversity p_d for tournament selection was estimated in [4,5] as

$$p_d = n^{-\frac{1}{n-1}} - n^{-\frac{n}{n-1}},$$

and later calculated exactly in [23] as

$$p_d = \frac{1}{M}\sum_{k=1}^{M}\left(1 - \frac{k^n - (k-1)^n}{M^n}\right)^M.$$

The quantities t^* and p_d give an idea of the intensity with which a selection scheme acts on the population as a function of the tournament size n and population size M.

The only two other pieces of research we are aware of that are relevant in the context of our work are [41,45]. In [41] a particular version of tournament selection that guarantees that all individuals in a run are sampled is proposed and it is shown, in some cases, to improve the problem-solving ability of a GA. Similar results have been recently reported in [45] which, following an early version of our work [32], proposes a different tournament strategy, which 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.

3. Sampling behaviour of tournament selection

Let us denote with S the number of selection steps required in each generation (it will immediately become apparent what is meant by this). If one assumed that only selection is used or that we use selection to form a mating pool,[1] the creation of a new generation would require exactly $S = M$ selection steps. These are exactly the conditions assumed in [41,45]. However, we do not make this assumption. Instead, we consider the currently much more common case 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). So, there are situations where more than M selection steps are required to form a new generation. Consider a generational selecto-recombinative algorithm, where crossover is performed with probability p_c and reproduction is performed with probability $1 - p_c$. (Mutation can be included by random changes to children after they have been created by either crossover and/or reproduction. In all cases the number of selection

[1] The mating pool is an intermediate population which gets created by using only selection and from which other operations, such as reproduction and crossing over, draw individuals uniformly at random.

steps is unchanged.) The number S of selection steps required to form a new generation is a stochastic variable with mean

$$E[S] = M(1 - p_c) + \rho M p_c = M\big[1 + (\rho - 1)p_c\big],$$

where $\rho = 1$ for a crossover operator which returns two offspring after each application, and $\rho = 2$ if only one offspring is returned. The two-offspring version of crossover requires fewer tournaments, and, since $\rho = 1$, the number of selection steps required to form a new generation is not stochastic and we have simply $S = M$. For brevity in the following we will use the definition $\alpha = [1 + (\rho - 1)p_c]$.[2]

Because in each tournament we need n individuals and we perform $S = \alpha M$ selection steps, tournament selection requires drawing $n\alpha M$ individuals uniformly at random (with resampling) from the current population. An interesting side effect is, particularly for small tournaments, that not all individuals in a particular generation are necessarily sampled within the $n\alpha M$ draws. For example, let us imagine running an EA starting from a random population containing four individuals, which we will denote as 1, 2, 3 and 4. Let us assume that we are creating the next generation using tournament selection with tournament size $n = 2$ and mutation only. Then, the creation of the first individual will require randomly picking two individuals from the current population (say individuals 1 and 4) and selecting the best for mutation. We repeat the processes to create the second, third and fourth new individuals. Note it is entirely possible that individual 3 was never involved in any of the tournaments.

It is absolutely crucial, at this stage, to stress the difference between *not sampling* and *not selecting* an individual in a particular generation. Not selecting refers to an individual which was involved in one or more tournaments, but did not win any, and this is exactly what previous work on loss of diversity has concentrated on. Not sampling refers to an individual which did not participate in any tournament at all, simply because it was not sampled during the creation of the required $S = \alpha M$ tournament sets. It is individuals such as this that are the focus of this paper. Therefore, the results in this paper are orthogonal to those appeared in the work mentioned in Section 2 and are not limited by uniqueness assumptions.

Continuing with our argument, in general, how many individuals should we expect not to take part in any of $S = \alpha M$ tournaments? As will be shown in Section 6.1, an answer comes straight from the literature on the coupon collector problem. However, before we explain the connection in more detail, we may want to reflect briefly on why this effect is important.

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., memory, but also, and more importantly, CPU time for their creation and evaluation. For instance, individual 3 in the previous example was randomly generated and had its fitness evaluated in preparation for selection, but neither its fitness nor its genetic make up could have any influence on future generations. So, one might ask, why did we generate such an individual in the first place? And what about generations following the first two? It is entirely possible that an individual in generation two got created and evaluated, but was then neglected by tournament selection, so it had no effect whatsoever on generations 3, 4, etc. Did we really need to generate and evaluate such an individual? If not, what about the parents of such an individual: did we need them? What sort of saving could we obtain by not creating unnecessary individuals in a run?

In Section 6 we will provide theoretical answers to all the questions above and more. In particular, we will show that in some conditions, savings of 20% fitness evaluations or, in fact, even more are easily achievable. Before we do this, however, we want to reconsider the way EAs are run and see whether there are ways in which we could exploit the inefficiencies of tournament selection. Amazingly, we will find that not only there are efficient algorithms for achieving this, but also that this can be done *without altering in any way the expected behaviour of evolutionary algorithms*.

4. Running EAs efficiently

Normally, in each generation of an EA with tournament selection we iterate the following phases (see Algorithm 2):

[2] In the following we will ignore the (potential) stochasticity of S. This is justifiable for various reasons: a) it simplifies the analysis (but without significant loss in terms of accuracy of the results obtained, as empirically verified), b) when $\rho = 1$ (two-offspring crossover or mutation only algorithm) there is no stochasticity (and so the analysis is exact), c) even with $\rho = 2$ (one-offspring crossover) it is possible to slightly modify the EA in such a way that there is no stochasticity.

```
1: Randomly initialise individuals in population pop, calculate corresponding
   fitness values, and store them in vector fit
2: for gen from 1 to G do
3:   for ind from 1 to M do
4:     op = choose genetic operator
5:     for arg from 1 to arity(op) do
6:       pool = choose n random individuals drawing from pop
7:       w[arg] = select winner from pool based on fitnesses in fit
8:     end for
9:     newpop[ind] = result of running operator op with arguments w[1],...
10:    newfit[ind] = fitness of newpop[ind]
11:  end for
12:  pop = newpop
13:  fit = newfit
14: end for
```

Algorithm 2. Standard generational EA with tournament selection. M is the population size, n is the tournament size, and G is maximum number of generations.

(a) the choice of genetic operator to use to create a new individual (step 4 in Algorithm 2),
(b) the creation of a random pool of individuals for the application of tournament selection (step 6),
(c) the identification of the winner of the tournament (parent) based on fitness (step 7),
(d) the execution of the chosen genetic operator (step 9),
(e) the evaluation of the fitness of the resulting offspring (step 10).

Naturally, phases (b) and (c) are iterated as many times as the arity of the genetic operator chosen in phase (a), and the whole process needs to be repeated as many times as there are individuals in the new population.

The genetic makeup of the individuals involved in these operations is of interest only in phase (d) (we need to know the parents in order to produce offspring) and phases (c) and (e) (we must know the genetic makeup of individuals in order to evaluate their fitness). However, phases (a) and (b) (steps 4 and 6 in Algorithm 2) do not require any knowledge about the actual individuals involved in the creation of a new individual. In most implementations these phases are just performed by properly manipulating numbers drawn from a pseudo-random number generator.

So, there is really no reason why we could not 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). This idea was first used in [46] for the purposed on speeding up GP fitness evaluation.[3]

In fact, we could go even further. In many practical applications of EAs, people fix a maximum number of generations they are prepared to run their algorithm for.[4] Let this number be G. So, at the cost of some memory space, as shown in Algorithm 3, we could iterate phases (a) and (b) not just for one generation but for a whole run from the first generation to generation G (steps 2–9) and then iterate phases (c)–(e) (steps 10–18) as required (that is, either until generation G or until any other stopping criterion is satisfied). We call this algorithm an *EA with macro-selection*, for obvious reasons.

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 speaking this version of the algorithm is exactly the same as the original. In fact, if the same seed is used for the random number generator in both algorithms, *they are indistinguishable*! However, unlike the standard EA, the EA with macro-selection can easily be modified to avoid wasting the computation involved in generating and evaluating the individuals "neglected" by tournament selection.

[3] The main idea in [46] was to estimate the fitness of the individuals involved in the tournaments by evaluating them on a randomly chosen subset of the training set available. On the basis of this estimate, for most tournaments it was often possible to determine with a small error probability which individual would win if all the examples where used. These tournaments could therefore be decided quickly, while only in a subset of tournaments individuals ended up being evaluated using the whole training set. This is what produced their speed up.

[4] This is a limit that is virtually always present, even if another stopping criterion, e.g., based on fitness, is present.

```
1: Randomly initialise individuals in population pop[0], calculate corresponding fitness values,
   and store them in vector fit[0]
2: for gen from 1 to G do
3:    for ind from 1 to M do
4:       op[gen][ind] = choose genetic operator
5:       for arg from 1 to arity(op[gen][ind]) do
6:          pool[gen][ind][arg] = choose n random individuals drawing from pop[gen-1]
7:       end for
8:    end for
9: end for
10: for gen from 1 to G do
11:    for ind from 1 to M do
12:       for arg from 1 to arity(op[gen][ind]) do
13:          w[arg] = select winner from pool[gen][ind][arg] based on fitnesses in fit[gen-1]
14:       end for
15:       pop[gen][ind] = result of running operator op[gen][ind] with arguments w[1],...
16:       fit[gen][ind] = fitness of pop[gen][ind]
17:    end for
18: end for
```

Algorithm 3. EA with macro-selection.

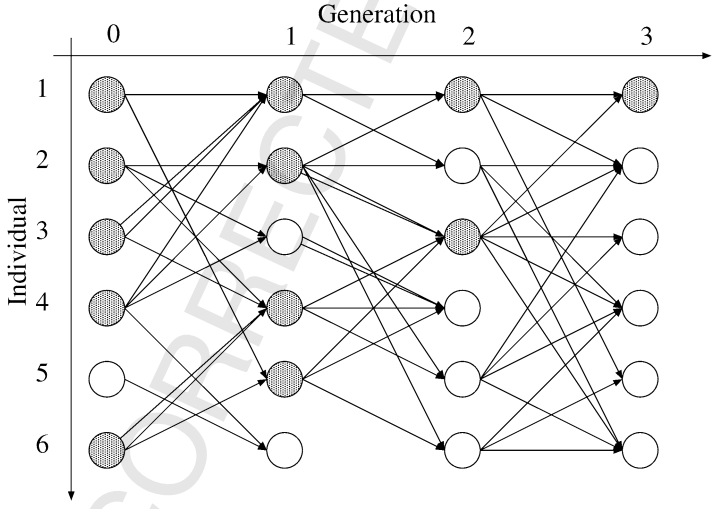

Fig. 1. Example of graph structure induced by tournament selection. Shaded nodes are the possible ancestors of the first individual in the last generation. Note how some nodes are not directly or indirectly connected to the nodes in the last generation.

To see how this is possible we should note that the iteration of phases (a) and (b) over multiple generations (steps 2–9 in Algorithm 3) induces a *graph structure* containing $(G+1)M$ nodes. Nodes represent all the individuals during a run (more precisely the elements of the pop array). Edges (which are stored in the pool array) connect each individual to the individuals which were involved in the tournaments necessary to select the parents of such an individual. We will call these individuals the *possible ancestors* of the individual (note that the possible ancestors of an individual are a superset of the *actual* ancestors, i.e., the parents, the parents of the parents, etc. of the individual in question).

Let us consider an example where we have a population of $M = 6$ individuals which we run for $G = 3$ generations using binary tournaments ($n = 2$), crossover rate $p_c = 1/3$ and each crossover produces one child. Mutation and reproduction are performed with a rate $(1 - p_c) = 2/3$. The graph induced by tournament selection might look like the one in Fig. 1.

```
1: for gen from 1 to G do
2:   for ind from 1 to M do
3:     op[gen][ind] = choose genetic operator
4:     for arg from 1 to arity(op[gen][ind]) do
5:       pool[gen][ind][arg] = choose n random individuals drawing from pop[gen-1]
6:     end for
7:   end for
8: end for
9: Analyse connected components in pool array and calculate neglected array
10: Randomly initialise individuals in population pop[0] except those
    marked in neglected[0], calculate fitness values, and store them in vector fit[0]
11: for gen from 1 to G do
12:   for ind from 1 to M do
13:     if not(neglected[gen][ind]) do
14:       for arg from 1 to arity(op[gen][ind]) do
15:         w[arg] = select winner from pool[gen][ind] [arg] based on fitnesses in fit[gen-1]
16:       end for
17:       pop[gen][ind] = result of running operator op[gen][ind] with arguments w[1],...
18:       fit[gen][ind] = fitness of pop[gen][ind]
19:     end if
20:   end for
21: end for
```

Algorithm 4. EA with efficient macro-selection.

We must emphasise that, although the graph-structure connecting the individuals in the population across time induced by tournament selection is particularly evident in (and is, thereby, revealed by) the EA with macro-selection, it is, nonetheless, present and deeply embedded in *every* EA using this form of selection.

So, how is this going to help us avoid generating and evaluating the individuals "neglected" by tournament selection? Simple. After macro-selection (the iteration of phases (a) and (b) up until generation G) is completed we analyse the information in the graph structure induced by tournament selection and identify which individuals are unnecessary, we mark them, and we avoid calculating and evaluating them when iterating phases (c)–(e). Clearly we want to mark those population members that were not involved in any tournament in each generation. However, if we are interested in calculating and evaluating all the individuals in the population at generation G, maximum efficiency is achieved by considering only the individuals which are directly or indirectly connected with the M individuals in generation G—a problem we can easily solve with a trivial connected-component algorithm. The modified algorithm is shown in Algorithm 4. We call this an *EA with efficient macro-selection* (*EA-EMS*). Note that to maximise efficiency, unusually, initialisation is not the first phase of the EA effectively coming *after* the macro-selection and connected-component detection phases.

Let us have a brief look at the differences between our new EA-EMS and a standard EA. Irrespective of the problem being solved and the parameter settings used, the behaviours of the standard algorithm and the efficient version proposed above will have to be on average identical. So, what are the differences between the two EAs?

Obviously, the standard algorithm requires more fitness evaluations and creations of individuals while the EA-EMS requires more bookkeeping and use of memory. Also, clearly, in any particular run, the plots of average fitness and maximum fitness in each generation may differ (since in EA-EMS not all individuals are considered in calculating these statistics). However, when averaged over multiple runs the average fitness plots would have to coincide.

A more important difference comes from the fact that most practitioners keep track of the best individual seen so far in a run of an EA and designate that as the result of the run. In EA-EMS we can either return the best individual in generation G or the best individual seen in a run, *out of those that have been sampled by tournament selection*. Because the EA-EMS algorithm does not create and evaluate individuals that did not get sampled, the end-of-run results may differ in the EA and EA-EMS algorithms. Of course, quite often the best individual seen in a run is actually a member of the population at the last generation. So, if one creates and evaluates all individuals in generation G (which leads to only a minor inefficiency in the EA with efficient macro-selection), most of the time the two algorithms will behave identically from this point of view too.

The EA-EMS offers us a way to fully exploit the sampling behaviour of tournament selection. This might appear to be the best we can get. However, the recursive nature of connected-component detection and the similarity between the mechanics of EAs and that of rule-based systems suggest a way to make further substantial improvements, as will be discussed in the next section.

5. Backward-chaining EAs and rule-based systems

At a sufficiently high level of abstraction, there are surprising similarities between EAs and Rule-Based Systems (RBSs) operating in forward-chaining mode (e.g., see [35,48]). In RBSs, we start with a working memory containing some premises, we apply a set of IF-THEN inference rules which modify the working memory by adding or removing facts and we iterate this process until a certain condition is satisfied (e.g., a fact which we consider to be a conclusion is asserted). The working memory in a RBS has a similar rôle to that of the population in an EA, and the facts in the working memory are effectively like the individuals in an population. Because the rules in the knowledge base of a RBS effectively manipulate the facts in the working memory, they share some similarity with the genetic operators in an EA which create new members of the population.

Running EAs from generation 0, to generation 1, to generation 2, and so on is the norm: the clock ticks forward in nature, and this is certainly what has been done for decades in the field of evolutionary computation. The loose analogy between RBSs and EAs mentioned above is not, in itself, terribly useful, except for one thing: it suggests the possibility of running an EA in backward chaining mode, like one can do with a RBS, thereby radically subverting the natural order of operations in the EA and the "time = generation number" EA canon.

Broadly speaking, when a RBS is run in backward chaining, the system focuses on one particular conclusion that it attempts to prove and operates as follows: a) it looks for all the rules which have such a conclusion as a consequent (i.e. a term following the "THEN" part of a rule), b) it analyses the antecedent (the "IF" part) of each such rule, c) if the antecedent is a fact (in other words, it is already in the working memory) then the original conclusion is proven and can be placed in the working memory. Otherwise the system saves the state of the inference and recursively restarts the process with the antecedent as a new conclusion to prove. If there is no rule which has the conclusion as a consequent, the recursion is stopped and another way of proving it is attempted. If a rule has more than one condition (which is quite common), the system attempts to prove the truth of all the conditions, one at a time. It will assert the conclusion of the rule only if all conditions are satisfied. When backward chaining, the RBS only considers rules that can contribute to determining the truth or falsity of the target conclusion. This can lead to major efficiency gains.

So, how would we run an EA in backward-chaining mode? Let us suppose we are interested in knowing the makeup of the population at generation G and let us start by focusing on the first individual in the population. Let r be such an individual. Effectively r plays the rôle of a conclusion we want to prove. 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.[5] In each such tournaments we will need to know the makeup of n (the tournament size) individuals from the previous generation (which, of course, at this stage we may still not know). Let us call $I = \{s_1, s_2, \ldots\}$ the set of the individuals that we need to know in generation $G - 1$ in order to determine r. Clearly, $s_1, s_2, \ldots$ are like the premises in a rule which, if applied, would allow us to work out r (this would require evaluating the fitness of each element of I, deciding the winners of the tournament(s) and applying the chosen genetic operator to generate r). Normally we will not know the makeup of these individuals. However, we can recursively consider each s_i 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 for the particular EA we are considering, or b) the individual for which we need to know the genetic makeup has already been constructed and evaluated. Clearly the individuals in generation 0 have a rôle similar to that of the initial contents of the working memory in a RBS. Once we have finished with r we repeat the process with all the other individuals of interest in the population at generation G, one by one. The process is summarised in Algorithm 5. We will call an EA running in this mode a *Backward-Chaining EA* (*BC-EA*).

[5] Decisions regarding operator choice and tournaments are trivial and can be made on the spot by drawing random numbers or can be all made in advance as in the EA-EMS.

1: Let r be an individual in the population at generation G
2: Choose an operator to apply to generate r
3: Do tournaments to select the parents:
$\{s_1, s_2, \ldots\}$ = individuals in generation $G-1$ involved in the tournaments
4: Do recursion using each unknown s_i as a subgoal. Recursion terminates
at generation 0 or when the individual is known (i.e. has been evaluated before).
5: Repeat for all individuals of interest in generation G.

Algorithm 5. Backward-chaining EA.

Clearly, at its top-level, the BC-EA is a recursive depth-first traversal of the graph induced by tournament selection (see Fig. 1 and Section 4). While we traverse the graph (more precisely, when we re-emerge from each recursion), we are in a position to know the genetic makeup of the nodes encountered and so we can invoke the fitness evaluation procedure for them. Thus, we can label each node with the genetic makeup and fitness of the individual represented by such a node. Recursion stops when we reach a node without incoming links (a generation-0 individual, which gets immediately labelled randomly and evaluated) or when we reach a node that has been previously labelled.

Statistically a BC-EA is fully equivalent to the EA-EMS, and so it presents the same level of equivalence to an ordinary EA. In particular, if the same seed is used for the random number generators and all decisions regarding operators and tournaments are performed in a batch before the graph traversal, the Gth generations of a BC-EA and an EA are indistinguishable.

So, if there are no differences why bother with a BC-EA instead of using a simpler "forward-chaining" version of the algorithm? One important difference between the two modes of operation is the order in which individuals in the population are evaluated. To illustrate this let us re-consider the example in Fig. 1 and let us suppose that, in the first instance, we are interested in knowing the first individual in the last generation. (The possible ancestors of this individual are shown as shaded nodes in Fig. 1.) Furthermore, for brevity, let us denote the nodes in row i (for individual) and column g (for generation) in the graph with the notation r_{ig}. In a forward chaining EA, even if we knew which individuals are unnecessary to define our target individual r_{13} (individuals r_{50}, r_{31}, r_{61}, r_{22}, etc.), we would evaluate individuals column by column from the left to the right. (E.g. EA-EMS evaluates r_{10}, r_{20}, r_{30}, r_{40}, r_{60}, r_{11}, r_{21}, r_{41}, r_{51}, r_{12}, r_{32}, and finally r_{13}.) That is, generation 0 individuals are computed before generation 1 individuals, which in turn are computed before generation 2 individuals, and so on. A BC-EA would instead evaluate nodes in a different order. For example, it might do it according to the sequence: r_{10}, r_{30}, r_{40}, r_{11}, r_{20}, r_{21}, r_{12}, r_{60}, r_{41},r_{51}, r_{32}, and finally r_{13}. So, the algorithm would move back and forth evaluating nodes at different generations. That is "time $\neq$ generation number" in the BC-EA.

Why is this important? Typically, in an EA both the average fitness of the population and the maximum fitness in each generation grow as the generation number grows. In our forward chaining 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 forward-chaining 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. Individuals 5 and 6 have same expected fitness in the two algorithms. However, the seventh individual drawn by BC-EA is a generation 2 individual, while the forward EA draws a generation 1 individual. So, again the BC-EA is expected to produce a higher fitness sample than the other EA. Of course, this process is not going to continue indefinitely, and at some point the individuals evaluated by BC-EA start being on average inferior. This is unavoidable since the sets of individuals sampled by the two algorithms are identical.

This behaviour is general. In virtually all problems of practical interest, fitness tends to increase generation after generation. So a BC-EA will find fitter individuals faster than an EA-EMS 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 more efficient than an ordinary EA because it avoids evaluating individuals neglected by tournament selection but also because it tends to find better solutions faster. *I.e. BC-EA is also a more effective search algorithm*. How can we make sure we work in the region where the BC-EA is superior to the corresponding EA-EMS? Simple: like any ordinary EA, in a BC-EA one does not need to continue evolution until all the individuals in generation G are known and evaluated; we can stop the algorithm whenever the best fitness seen so far reaches a suitably high value. In this way we can avoid at least a part of the phase where BC-EA is slower than the EA-EMS.

It is worth noting that this "faster convergence" behaviour is present in a BC-EA irrespective of the value of the tournament size, although, of course, the benefits of using BC-EAs depend on it.

6. Theory

In this section we want to model mathematically the sampling behaviour of tournament selection and understand what savings we can achieve using the EA-EMS and the BC-EA. We will start by drawing an analogy between tournament selection and the coupon collection problem.

6.1. Coupon collection and tournament selection

In the *coupon collector problem*, every time a collector buys a certain product, a coupon is given to him or her. 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. The question is: how many products will the collector have to buy before he can expect to have a full set of coupons? The answer [9] can be derived by considering that the probability of obtaining a first coupon in one trial is 1 (so the expected waiting time is just 1 trial), the probability of obtaining a second coupon (distinct from the first one) is $\frac{N-1}{N}$ (so the expected waiting time is $\frac{N}{N-1}$), the probability of obtaining a third coupon (distinct from the first two) is $\frac{N-2}{N}$ (so the expected waiting time is $\frac{N}{N-2}$), and so on. So, the expected number of trials to obtain a full set of coupons is

$$E_N = 1 + \frac{N}{N-1} + \frac{N}{N-2} + \cdots + N = N \log N + \mathrm{O}(N).$$

It is well known that the $N \log N$ limit is sharp. If X is the number of purchases before one of each type of coupon is collected, for any constant c

$$\lim_{N\to\infty} \Pr\{X > N \log N + cN\} = 1 - \mathrm{e}^{-\mathrm{e}^{-c}}.$$

E.g., for $c = 3$, in the limit where there are many types of coupons, the probability it takes more than $N \log N + 3N$ trials to purchase at least one of each type is less than 5%.

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) $n\alpha M$ times from this pool of coupons. Because of the sharpness of the coupon-collector limit mentioned above, if $n\alpha > \log M + c$ for some suitable positive constant c, then we should expect tournament selection to sample all individuals in the population most of the time. However, for sufficiently small tournament sizes or for sufficiently large populations the probability that there will be individuals not sampled by selection becomes significant.

So, how many different coupons (individuals) should we expect to have sampled at the end of the $n\alpha M$ trials? In the coupon collection problem, the expected number of trials necessary to obtain a set of x distinct coupons is [9]

$$E_x = 1 + \frac{N}{N-1} + \frac{N}{N-2} + \cdots + \frac{N}{N-x+1} = N \log \frac{N}{N-x} + \mathrm{O}(N).$$

By setting $E_x = n\alpha M$, $N = M$ and ignoring terms of order $\mathrm{O}(N)$, from this we obtain an estimate for the number of distinct individuals sampled by selection

$$x \approx M(1 - \mathrm{e}^{-n\alpha}). \tag{1}$$

This indicates that the expected proportion of individuals *not* sampled in the current population varies approximately like a negative exponential of the tournament size.

This approximation is quite accurate. However, we can calculate the expected number of individuals neglected after performing $n\alpha M$ trials directly. We first calculate the probability that one individual is not involved in one trial as $1 - 1/M$. Then the expected number of individuals not involved in any tournaments is simply

$$M(1 - 1/M)^{n\alpha M} = M\left(\frac{M}{M-1}\right)^{-n\alpha M},$$

which also varies like a negative exponential of the tournament size.

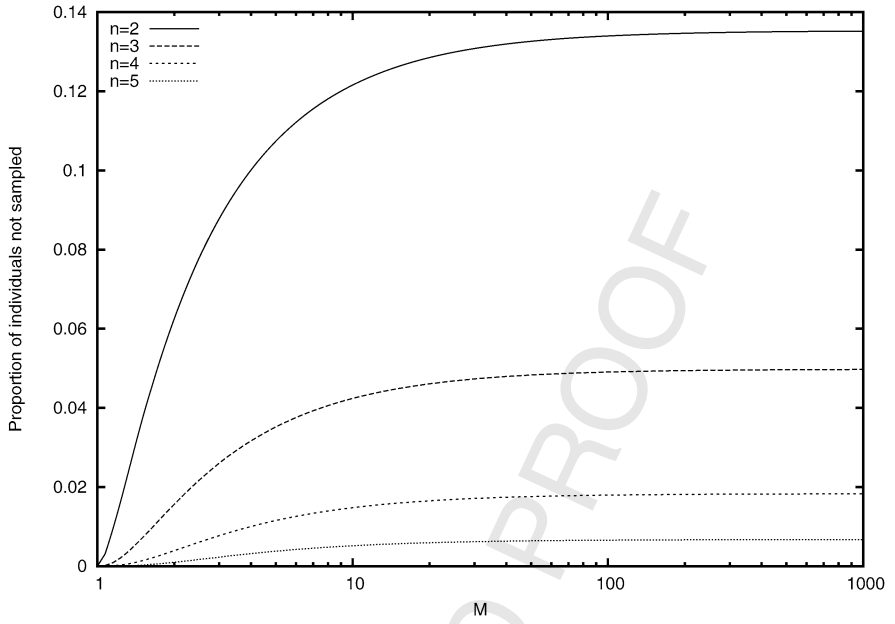

Fig. 2. Proportion of individuals not sampled in one generation by tournament selection for different tournament sizes, n, and population sizes M assuming only two-offspring crossover and/or mutation are used.

As shown in Fig. 2 for $\alpha = 1$ (two-offspring crossover or no crossover), typically for $n = 2$ over 13% of the population is neglected, for $n = 3$ this drops to 5%, for $n = 4$ this is 2%, and becomes negligible for bigger values of n.

This simple analysis suggests that saving computational resources by avoiding the creation and evaluation of individuals which will not be sampled by the tournament selection process is possible only for relatively low selection pressures. However, tournament sizes in the range 2–5 are quite common in practice, particularly when attacking hard, multi-modal problems which require extensive exploration of the search space before zooming the search onto any particular region. Furthermore, in the next section, where we look at the behaviour of tournament selection over multiple generations, we will show that much bigger savings than those suggested above can be achieved.

6.2. Iterated coupon collector problem

Let us consider a new game, that we will call the *iterated coupon collection problem*, where the coupon set changes at regular intervals, but the number of coupons available, N, remains constant. Initially the collector is given a (possibly incomplete) set of m_0 old coupons. Each old coupon allows the collector to draw n new coupons. So, he can perform a total of nm_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 nm_1 trials to gather as many new distinct coupons as possible. And so on. Interesting questions here are: what happens to m_t as t grows? Will it reach a limit? Will it oscillate? In which way will the values of n, m_0 and N influence its behaviour?

Before we answer these questions let us motivate our analysis a little. How is this new problem related to EAs and tournament selection? The connection is simple (we will assume $\alpha = 1$, e.g. two offspring crossover and mutation, for the sake of clarity). Suppose we are interested in computing and evaluating m_0 individuals in a particular generation, G, of a run. 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. This will require running m_0 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 such trials we will be in a position to determine which individuals in generation $G - 1$ will contribute to future generations, we can count them and denote this number

with m_1.[6] So, again, we can concentrate on these individuals only. They are equivalent of the new set of coupons the collector has gathered. We can now perform nm_1 trials to determine which individuals in generation $G-2$ (the new coupon set) will contribute to future generations, we can count them and denote this number with m_2 and so on until we reach the initial random generation. There the game stops.

The graph induced by tournament selection is a stochastic variable. Every time we run an EA, we instantiate such a variable. So, in terms of the graph structure associated to tournament selection, the process described above corresponds to the instantiation of one such structure and m_t corresponds to the number of possible ancestors (nodes) of the m_0 individuals of interest, in the $(G-t)$th vertical layer of the graph. So, effectively, *the iterated coupon collector problem is a model for the sampling behaviour of tournament selection over multiple generations in a generational EA*.

Knowing the sequence m_t for a particular EA would tell us how much we could save by not creating and evaluating individuals which will not be sampled by selection. Naturally, we will not have an oracle to help us choose G and to give us m_0. For now, while we concentrate on understanding more about the iterated coupon collector problem, we could think of G as the number of generations we are prepared to run our EA for, and we might imagine that $m_0 = M$ (the whole population).

In the classical coupon collection problem, the shopper will typically perform as many trials as necessary to gather a full collection of coupons. As we have seen before, however, it is quite easy to estimate how many distinct coupons one should expect at the end of any given *fixed* number of trials. Because the iterated coupon collection game starts with a known number of trials, we can calculate the *expected value* of m_1. However, we cannot directly apply the theory in the previous section to gather information about m_2. This is because m_1 is a stochastic variable, so in order to estimate m_2 we would need to know the probability distribution of m_1 not just its expected value.

Exact probabilistic modelling can be obtained by considering the coupon collection game as a Markov chain, where the state of the chain is the number of distinct coupons collected. The transition matrix for the chain can easily be constructed by noticing that the chain can be in state k (i.e., the collector has k distinct coupons) after the next coupon is purchased only if either it was already in state k and the new coupon is a duplicate (which happens with probability $\frac{k-1}{N}$) or it was in state $k-1$ and the next coupon is different from all those currently held (which, of course, happens with probability $\frac{N-k+1}{N}$). So, the number of distinct individuals in the previous generation sampled when randomly picking individuals for tournament selection can be described by applying the following Markov transition matrix a number of times:

$$A = \frac{1}{M}\begin{pmatrix} 0 & 0 & 0 & 0 & \dots & 0 \\ M & 1 & 0 & 0 & \dots & 0 \\ 0 & M-1 & 2 & 0 & \dots & 0 \\ 0 & 0 & M-2 & 3 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & M \end{pmatrix}.$$

The process always starts from state 0. This can be represented using the state probability vector $e_0 = (1 \quad 0 \quad 0 \quad 0 \quad \dots \; 0)^{\mathrm{T}}$.[7] So, the probability distribution over the states after t coupon purchases (or random samples from the population) is given by $A^t e_0$, which is simply the first column of the matrix A^t.

Suppose we are only interested in m_0 individuals in the last generation G. The number of tournaments used to construct the last generation will be $n\alpha m_0$. Therefore the probability distribution of the number of distinct individuals we need to know from generation $G-1$, m_1, is given by $A^{n\alpha m_0} e_0$. Notice that this also gives us the probability distribution over the number of draws, $n\alpha m_1$, we will need to make from generation $G-2$ in order to fully determine the m_1 individuals we want to know at generation $G-1$.

[6] Note we work back from the last generation, index_0, generation $G-1$, index_1, generation $G-2$, index_2, etc. Also, because at this stage we are only interested in knowing the number of individuals playing an active rôle in generation $G-1$, there is no need to determine the winners of the tournaments. We just need to know who was involved in which tournament. So, we do not even need to evaluate fitness, and, therefore, we do not need to know the genetic makeup of any individual.

[7] Each element of a state probability vector represents the probability of a system being in the corresponding state. Since a system must always be in some state, the elements of the vector must add up to 1. In the coupon collection problem initially only state 0 is possible. So, only the first element of e_0 is non-zero.

For example, if the population size is $M = 3$, the number of states in the Markov chain is $M+1 = 4$, the tournament size is $n = 2$, we use a two-offspring version of crossover ($\alpha = 1$) and we are interested in $m_0 = 1$ individuals (so $n\alpha m_0 = 2$), then the probability distribution of m_1 is represented by the following probability vector

$$A^2 e_0 = \left(\frac{1}{3}\right)^2 \begin{pmatrix} 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 0 & 2 & 2 & 0 \\ 0 & 0 & 1 & 3 \end{pmatrix}^2 \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \left(\frac{1}{9}\right) \begin{pmatrix} 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 6 & 6 & 4 & 0 \\ 0 & 2 & 5 & 9 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{3} \begin{pmatrix} 0 \\ 1 \\ 2 \\ 0 \end{pmatrix}.$$

If we were interested in $m_0 = 2$ individuals at generation G, the probability distribution over the number m_1 of unique individuals sampled would be $A^4 e_0 = (0.0 \quad 0.0370 \quad 0.5185 \quad 0.4444)^{\mathrm{T}}$. Finally, if we were interested in the whole population ($m_0 = 3$) the distribution would be $A^6 e_0 = (0 \quad 0.0041 \quad 0.2551 \quad 0.7407)^{\mathrm{T}}$, which reveals that, in these conditions, even when building a whole generation there are still more than 1 in 4 chances of not sampling the whole population at the previous generation. Of course, if $m_0 = 0$, the probability vector for m_1 is e_0, i.e., $m_1 = 0$, and, more generally, $m_t = 0$ for $0 < t < G$.

Although this example is trivial, it reveals that for any given m_0 we can compute a distribution over m_1. That is, we can define a new *Markov chain* to model the *iterated coupon collector problem*. In this chain a state is exactly the same as in the coupon-collector chain (i.e., the number of distinct coupons collected), *except that now a time step corresponds to a complete set of draws from the new coupon set rather than just the draw of one coupon*. Since the number of states is unchanged, the transition matrix B for this new chain is the same size as A, i.e. $(M+1) \times (M+1)$. Column i of B is the probability distribution for $m_0 = i$, i.e. $A^{n\alpha i} e_0$. That is

$$B = \left(e_0 \middle| A^{n\alpha} e_0 \middle| A^{2n\alpha} e_0 \middle| \ldots \middle| A^{Mn\alpha} e_0\right).$$

For instance, we have just calculated these for the case $M = 3$, $n = 2$ and $\alpha = 1$ so

$$B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.3333 & 0.0370 & 0.0041 \\ 0 & 0.6667 & 0.5185 & 0.2551 \\ 0 & 0 & 0.4444 & 0.7407 \end{pmatrix}.$$

The important thing is that, now that the transition matrix is defined, the chain can be iterated to compute the probability distributions of m_2, m_3 and so on, as far back as necessary to reach generation 0.

In general B is block diagonal of the form

$$B = \left(\begin{array}{c|c} 1 & \mathbf{0}^{\mathrm{T}} \\ \hline \mathbf{0} & C \end{array}\right),$$

where $\mathbf{0}$ is a column vector containing M zeros and C is a $M \times M$ stochastic matrix. Clearly B is not ergodic (from state 0 we cannot reach any state other than 0 itself), so we cannot expect a unique limit distribution for m_t. However, because B is block diagonal, we have

$$B^x = \left(\begin{array}{c|c} 1 & \mathbf{0}^{\mathrm{T}} \\ \hline \mathbf{0} & C^x \end{array}\right).$$

So, if we ensured that the probability of the chain initially being in state 0 is 0 (that is $\Pr\{m_0 = 0\} = 0$), the chain could never visit such a state at any future time. Because of this property, and because, objectively, state 0 is totally uninteresting (of course we already know that if we are interested in no individual at generation G, we do not need to know any individual at previous generations!) we can declare such a state of the iterated coupon-collection chain as invalid, and reduce the state set to $\{1, 2, \ldots, M\}$. In this situation C is the state transition matrix for the chain, and to understand the sampling behaviour of tournament selection over multiple generations we just need to concentrate on the properties of C.

The transition matrix C is ergodic if $n\alpha > 1$ as can be easily seen by the following argument. If $n\alpha > 1$ then each old coupon gives us the right to draw more than one new coupon in the iterated coupon-collection problem. So, if the state of the chain is k ($k < M$), it is always possible to reach state $k+1$ in one stage of the game with non-zero probability. From there it is then, of course, possible to reach state $k+2$ and so on up to M. So, from any lower state it is always possible to reach any higher state in repeated iterations of the game. But, of course, the converse is always true: irrespective of the value of $n\alpha$ there is always a chance of getting fewer coupons than we had before in an

iteration of the game, due to resampling. So, from any higher state we can also reach any lower state (in fact, unlike the reverse, we can achieve this in just one iteration of the game).

Since, $\alpha \geqslant 1$ and $n > 1$ for any practical applications, the condition $n\alpha > 1$ is virtually always satisfied and C is ergodic. Then, the Perron–Frobenius theorem guarantees that the probability over the states of the chain converges towards a limit distribution which is independent from the initial conditions (see [7,8,25,33,36] for other applications of this result to EAs). This distribution is given by the (normalised) eigenvector corresponding to the largest eigenvalue of C ($\lambda_1 = 1$), while the speed at which the chain converges towards such a distribution is determined by the magnitude of the second largest eigenvalue λ_2 (the relaxation time of an ergodic Markov chain is $1/(1 - |\lambda_2|)$). Naturally, this infinite-time limit behaviour of the chain is particularly important if G is sufficiently big that m_t settles into the limit distribution well before we iterate back to generation 0. Otherwise the transient behaviour is what one needs to focus on. Both are provided by the theory.

Because the transition matrices we are talking about are relatively small (the matrix C is $M \times M$), they are amenable to numerical manipulation. We can, for example, find the eigenvalues and eigenvectors of C for quite respectable population sizes, certainly well in the range of those used in many applications of EAs, thereby determining the limit distribution and the speed at which this is approached.

If $p(t)$ is a probability vector representing the probability distribution over m_t, then the expected value of m_t is

$$E[m_t] = (1 \quad 2 \quad \ldots \quad M) \cdot p(t) = (1 \quad 2 \quad \ldots \quad M) \cdot C^t p(0). \tag{2}$$

Typically m_0 will be fixed by the user, i.e. the probability distribution $p(0)$ will be a delta function centred at this one value chosen by the user. Thus $p(0) = e_{m_0}$ (where e_l is a base vector containing all zeros except for element l which is 1).

If p^* denotes the limit distribution for $p(t)$, then for large enough G, the average number γ of individuals (in generations 0 through to $G - 1$) that have no effect whatsoever on a designated set of m_0 individuals of interest at generation G is approximately $\gamma = M - (1 \quad 2 \quad \ldots \quad M) \cdot p^*$. This is the average saving that could be achieved by not creating and evaluating unnecessary individuals using the EA-EMS and the BC-EA.

Notice that the ergodicity of the selection process means that over many generations the fraction of individuals we can avoid creating does not depend much on m_0. That is, for large enough G, whether m_0 is one or as large as M will make little difference to the saving. So we might want to know the entire makeup of generation G and still have a saving of approximately γG creations and evaluations of individuals.

6.3. Approximate model of transient behaviour

The model presented in the previous section is comprehensive and accurate, but for many practical purposes an approximate but simpler model would be desirable. We develop such a model in this section. In particular we will focus on modelling the transient behaviour of the number of individuals sampled by tournament selection over multiple generations, m_t.

When the number of individuals we want to know the fitness of at the end of our EA run, m_0, is sufficiently smaller than the population size M, we expect the number of individuals sampled by selection m_t to grow exponentially as we look back towards the start of the run. The reasons for this are quite simple: when only a few samples are drawn from a population, resampling is very unlikely, and, so, the ancestors of the individual of interest will tend to form a tree for at least some generations before the last. The branching factor of the tree is $n\alpha$. So, for small enough g, generation $G - g$ will include $(n\alpha)^g$ ancestors of each individual of interest in generation G. Naturally, this exponential growth continues only until resampling starts to become significant, i.e., until $m_0(n\alpha)^g$ becomes comparable with the expected number of individuals processed in the limit distribution p^*.

For small populations or high selective pressures the transient is short. However, there are cases where the transient lasts for many generations. For example, in a population of $M = 100\,000$ individuals and $\alpha = 1$ (i.e., in the case of only mutation and/or two-offspring crossover), the transient lasts for almost 20 generations (although it is exponential only for around 16 or 17). This population size may appear very big and these generation numbers may appear quite small. However, big populations and short runs are actually typical of at least one class of EAs, genetic programming [3, 18,19], where populations of several millions of individuals are not uncommon when solving complex important problems [17]. So, it is worth evaluating the impact of the transient on the total number of fitness evaluations.

Let us assume $G - g_e$ is the last generation in which the transient is effectively exponential. We can obtain an approximation of g_e by assuming $G - g_e$ is the generation at which the exponential $m_0(n\alpha)^{g_e}$ hits the population size limit M, whereby

$$g_e \approx \frac{\log(M/m_0)}{\log(n\alpha)}. \tag{3}$$

So, for example, if $n = 2$ and $\alpha = 1$, a population of size M and $m_0 = 1$, we have an exponential transient of $g_e \approx \log_2 M$ generations.

The number of individuals evaluated during the last g_e generations of a run is the sum of a geometric series, i.e.,

$$F^B = m_0 \frac{(n\alpha)^{g_e+1} - 1}{n\alpha - 1}.$$

By substituting Eq. (3) into this expression we obtain

$$F^B = m_0 \left(\frac{1}{n\alpha - 1}\right)\left(n\alpha\left(\frac{M}{m_0}\right) - 1\right). \tag{4}$$

This leads us to a simple upper bound for the number of individuals required in the exponential transient: $F^B < (\frac{n\alpha}{n\alpha-1})M$.

Eq. 4 allows us to compute the maximum efficiency gain obtainable. Let us assume we run our BC-EA for g_e generations and we are interested in computing m_0 individuals at generation g_e. To compute the same individuals, a standard EA would need to construct and evaluate of the order of

$$F^F = M \times g_e \approx \frac{M \log(M/m_0)}{\log(n\alpha)}$$

individuals. So, the speedup achievable exploiting the full transient is

$$\text{speedup} = \frac{F^F}{F^B} \approx \frac{\frac{M\log(M/m_0)}{\log(n\alpha)}}{m_0\left(\frac{1}{n\alpha-1}\right)\left(n\alpha\frac{M}{m_0} - 1\right)} \approx \frac{\log(M/m_0)}{\log(n\alpha)\left(\frac{n\alpha}{n\alpha-1}\right)}.$$

Clearly, maximum speedup can be obtained for $m_0 = 1$ and $n\alpha = 2$, in which case we have a speedup factor of approximately $\log_2 \sqrt{M}$. This can be big for very large populations. However, in practice it appears to be hard to achieve speed-ups of much more than around 10 or so, since the speedup factor is a logarithmic function of the population size.

7. BC-EA implementation

Encouraged by the theoretical evidence provided in the previous section which indicate that substantial savings can be achieved, following the ideas in Section 5, we have designed and implemented two backward-chaining EAs in Java. One is a simple GA, which we will refer to as BC-GA; the other is a GP implementation, which we will call BC-GP. The objective was to evaluate whether the BC-EA approach indeed brings significant efficiency gains, and whether BC-EAs compare well with equivalent standard (forward) versions in terms of ability to solve problems.

Algorithm 6 provides a pseudo-code description of the key components of our implementation. (All other components are exactly as in an ordinary EA and so are omitted for brevity.) 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. Instead we statically create the nodes in the graph (and store them using two-dimensional arrays) but graph edges are dynamically generated and stored on the stack as we do recursion. This is achieved by choosing the 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 $(G + 1) \times M$ arrays:

```
procedure run(G,M)
 1: Create arrays Known, Population and Fitness
 2: for all individuals I of interest in generation G do
 3:    evolve_back(I,G)
 4: end for
 5: return all I of interest

procedure evolve_back(indiv,gen)
 1: if Known[indiv][gen] then
 2:    return
 3: end if
 4: if gen = 0 then
 5:    Population[gen][indiv] = random individual
 6: else
 7:    if random_float() < crossover_rate then
 8:       parent1 = tournament(gen-1)
 9:       parent2 = tournament(gen-1)
10:       Population[gen][indiv] = crossover(parent1,parent2)
11:    else
12:       parent = tournament(gen-1)
13:       Population[gen][indiv] = mutation(parent)
14:    end if
15: end if
16: Fitness[gen][indiv] = fit_func(Population[gen][indiv])
17: Known[gen][indiv] = true
18: return

procedure tournament(gen)
 1: fbest = 0
 2: best = undefined
 3: for tournament_size times do
 4:    candidate = random integer 1...M
 5:    evolve_back( candidate, gen )
 6:    if Fitness[gen][candidate] > fbest then
 7:       fbest = Fitness[gen][candidate]
 8:       best = candidate
 9:    end if
10: end for
11: return Population[gen][best]
```

Algorithm 6. Backward-chaining EA with one-offspring crossover.

`Population` is an array containing the individuals in the population at each generation.[8]

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

`Known` is an array of bits. This is initialised to 0. A bit set to 1 indicates that the corresponding individual in `Population` has been computed and its fitness has been calculated.

Using arrays as our main data structures is appropriate given the scientific objectives of the implementation. However it is also wasteful since, in BC-EAs, only those entries of the arrays corresponding to individuals sampled by tournament selection are used. In a "production" implementation one could use more sophisticated and efficient data structures (such as hash tables) and save some memory.

[8] If the representation is of fixed size, individuals are directly stored in `Population`. With variable-size representations, however, this is an array of pointers to other dynamically-allocated data structures representing the individuals.

```
procedure evolve_back(indiv,gen)
 1: if Known[indiv][gen] then
 2:    return
 3: end if
 4: if gen = 0 then
 5:    Population[gen][indiv] = random individual
 6: else
 7:    myrand = random_float()
 8:    if myrand < crossover_rate then
 9:       if myrand < crossover_rate/2 or sibling_pool[gen] = empty then
10:          parent1 = tournament(gen-1)
11:          parent2 = tournament(gen-1)
12:          offsprings = crossover(parent1,parent2)
13:          Population[gen][indiv] = offspring[1]
14:          sibling_pool[gen].add(offspring[2])
15:       else
16:          Population[gen][indiv] = sibling_pool[gen].remove_random_indiv()
17:       end if
18:    else
19:       parent = tournament(gen-1)
20:       Population[gen][indiv] = mutation(parent)
21:    end if
22: end if
23: Fitness[gen][indiv] = fit_func(Population[gen][indiv])
24: Known[gen][indiv] = true
25: return
```

Algorithm 7. Backward-chaining EA with two-offspring crossover.

As we mentioned in Section 3, crossover operators that return two offspring require on average half the number of selection steps than crossovers returning one offspring. Therefore, one child crossover operators are less efficient in a BC-EA. In order to take full advantage of two-offspring crossovers we need to modify the algorithm slightly. The major change is to add an expandable array, `sibling_pool`, which temporarily stores the second offspring generated by each crossover operation. Other minor changes are required to the `evolve_back` routine (see Algorithm 7).

8. Space and time complexity of BC-EA

BC-EAs are based on changing the order of various operations in an EA. This requires memorising choices and individuals over multiple generations. Let us evaluate the space complexity of our BC-EA and compare it to the space complexity of standard EAs.[9]

8.1. Fixed-size representations

We consider EAs where the representation of each individual requires a fixed amount of memory: b bytes. The space complexity of a forward generational EA is

$$C^F = 2 \times (b+4) \times M$$

where we assumed that we store both the current and the new generation, and that fitness values are stored in a vector of floats (4 byte each). So, for $b \gg 1$, $C^F \approx 2bM$. In BC-EA we need to store one array of individuals (each being of size b), one of floats, and one bit array, all of size $(G+1) \times M$. So the space complexity is

$$C^B = (G+1) \times M \times \left(b + 4 + \frac{1}{8}\right).$$

[9] Our calculations will ignore the small amount of memory required in the stack during recursion. Also, in the case of BC-EAs with two-offspring crossover we will ignore the memory required for the expandable array `sibling_pool` since this typically contains only a few individuals.

For $b \gg 1$, $C^B \approx (G+1)bM$. So, the difference in space complexity between the two algorithms is

$$\Delta C = C^B - C^F \approx (G-1) \times M \times b$$

which indicates that in most conditions the use of BC-EA carries a significant memory overhead. However, this does not prevent the use of BC-EAs. For example, if the representation of an individual requires $b = 100$ bytes, and we run a population of $M = 1000$ individuals for 100 generations, BC-EA requires only around 10 MB of memory to run.

Let us now consider the time complexity of the BC-EA. In fixed-size representations, often the time required by each fitness evaluation is approximately constant. Since the time spent doing fitness evaluation almost invariably dominates that required by all other phases of an EA, the time complexity of a standard EA is proportional to the number of fitness function calls

$$F^F = (G+1)M.$$

Likewise, for the BC-EA we have a time complexity proportional to

$$F^B = E^B,$$

where E^B is the number of individuals actually created and evaluated during the run. Naturally, with low selective pressure and large populations $F^B < F^F$ and so, the BC-EA runs faster than a corresponding EA.

8.2. Variable-size representations

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 the EA excluding the individuals themselves, while C_{variable} represents the memory used by the individuals. For variable-size representations, such as GP trees, 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 EA system

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

As before, 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

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

since we need to store one array of pointers (4 bytes each), one of floats, and one bit array, all of size $(G+1) \times M$.

Variable space complexity is harder to compute. For a forward variable-size representation EA this is

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

where $S^F_{\max}$ is the maximum (over all generations) of the average size of the individuals in each generation of a run. In a BC-EA

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

where S^B_{avg} is the average size of the individuals during a BC-EA run (i.e., it is the individual size averaged over all individuals created *in a run*). (Remember E^B is the number of individuals actually created and evaluated during the run.) So, the difference in space complexity between the two algorithms is

$$\Delta C = C^B - C^F \approx M\big(8(G+1) - 16\big) + E^B \times S^B_{\text{avg}} - 2 \times M \times S^F_{\max},$$

which, again, indicates that in most conditions the BC-EA carries a significant memory overhead. However, again, this does not prevent the use of BC-EA. Consider, for example, a BC-GP system with a population of 100 000 individuals run for 50 generations, where the average program size (throughout a run) is 100 bytes. In the worst possible case

(where all programs are constructed and evaluated and, so, $E^B = (G+1)M$) we need just over 500 MB of memory—which is readily available in most modern personal computers.

The memory overhead of BC-EA, ΔC, is a function of the average average-individual-size S^B_{avg} and the maximum average-individual-size S^F_{max}. We know that statistically BC-EA and EA behave the same, so we expect $S^F_{\text{max}} = S^B_{\text{max}}$ and so $S^B_{\text{avg}} < S^F_{\text{max}}$. We cannot, however, say much more about ΔC in general since the size of individuals often varies widely.

As in fixed-size representations, time complexity is dominated by fitness evaluation. Naturally, the number of fitness function calls is the same as in the fixed-size case (see Section 8.1). However, very often, in variable-size representations, the execution time of the fitness function varies with the size of the individual being evaluated. So, to say something more precise we need to know how size varies and how evaluation time depends on the size of the representation. To illustrate how this analysis can be done, in the next section we consider the case of GP.

8.2.1. Space and time complexity in BC-GP

The variability in individual size is particularly marked in GP due to a common phenomenon known as *bloat*. Bloat is a progressive growth in program size not accompanied by a corresponding improvement in fitness [20]. This can become very marked in the late phases of a run. If bloat happens in a particular problem, then programs in both standard GP and BC-GP will increase in size. However, since with BC-GP we may choose to evaluate only a few individuals in the last generations of a run (i.e. $m_0 \ll M$), and since bloat is typically most marked in these generations, S^B_{avg} can be much smaller than S^F_{avg}. That is, with bloat the programs created in a BC-GP system may be on average smaller than those created by forward GP. So, we may have $S^B_{\text{avg}} \ll S^F_{\text{max}}$.

These effects partly mitigate the memory overhead, ΔC, of BC-GP. Also because BC-GP tends to evaluates smaller programs than GP bloat has an interesting 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 examples in the training set (or fitness cases in GP's jargon), K. So, if we express T in number of primitives executed, for standard GP we have

$$T^F = F^F \times K \times S^F_{\text{avg}} = (G+1) \times M \times K \times S^F_{\text{avg}}$$

and for BC-GP

$$T^B = F^B \times K \times S^B_{\text{avg}} = E^B \times K \times S^B_{\text{avg}}.$$

So, the saving provided by BC-GP is

$$\Delta T = T^F - T^B = K \times \left((G+1) \times M \times S^F_{\text{avg}} - E^B \times S^B_{\text{avg}}\right).$$

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 to produce even more impressive savings.

8.3. Wall-clock execution-time

The number of fitness evaluations is the standard measure for efficiency for evolutionary algorithms. This is reasonable since, as already mentioned, for any non-trivial problem, the cost of fitness evaluation is the overwhelming component in the computation load of most EAs. This is particularly true for GP.

When comparing different algorithms it is often assumed that fitness evaluation will require approximately the same computation in all of the algorithms tested. However, as we observed above, this is not true when evaluation time depends on particular features (e.g., size) of the structures being evaluated, like in GP. In the case of GP, for example, we saw that it is more appropriate to use the number of primitives executed as a measure of efficiency. However, in practice there are situations where execution time may be effected by other factors, such as the total amount of memory used by an algorithm. In particular, if an algorithm uses an amount of memory that exceeds the physical memory of the computer and/or if there are other programs competing for memory running at the same time, paging and disk access for memory may become important factors in determining the performance of the algorithm.

Naturally, these factors are difficult to include in an analysis. However, because BC-EAs use more memory than the corresponding forward versions, it is clear there must be settings where paging and disk access will slow down

BC-EAs. This may make these algorithms actually less efficient than corresponding forward EAs, even though, in theory, they should be faster. Fortunately, as we will show in Section 9.4, even in very demanding conditions we never observed this happen, despite our using a fairly ordinary PC for our experiments.

9. Experimental results

We performed experiments with a BC-GA and a BC-GP with both one-offspring and two-offspring crossover, comparing them to standard GA and GP versions.

9.1. Test problems

In the case of the BC-GA we run experiments with the *counting ones problem*. This is a simple linear problem, widely used for benchmarking purposes, requiring a binary representation, where the fitness of an individual is the number of ones in the bit string representing such an individual.

In the case of BC-GP we considered a larger variety of problems with complexity ranging from very simple to very hard. In all problems the objective was to induce a continuous target function from examples. Problems of this type are known as *symbolic regression problems* in the GP literature (e.g., see [3,18]), since GP is asked to find a *function* which fits certain data-points, rather than finding *coefficients* for a pre-fixed function, as in a standard regression task. 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 easy 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 $f(x_1, x_2, x_3, x_4) = x_1x_2 + x_3x_4 + x_1x_4$. This is a much harder problem than the previous one, but it is still solvable. For this problem 50 fitness cases of the form $(x_1, x_2, x_3, x_4, f(x_1, \ldots, x_4))$ were used. They were generated by randomly setting $x_i \in [-1, +1]$. The second multivariate polynomial, Poly-10, is $f(x_1, \ldots, x_{10}) = x_1x_2 + x_3x_4 + x_5x_6 + x_1x_7x_9 + x_3x_6x_{10}$. Also for this problem we used 50 fitness cases of the form $(x_1, \ldots, x_{10}, f(x_1, \ldots, x_{10}))$, which, again, were obtained by randomly setting $x_i \in [-1, +1]$. This problem is extremely hard.

9.2. GA vs. BC-GA

Let us start by corroborating experimentally the equivalence between GA and BC-GA and the expected faster convergence behaviour of BC-EA. To assess this we performed 100 independent runs of both a forward GA and our BC-GA applied to a 100-bit counting ones problem. In these runs the maximum number of generations G was set to 99 (i.e., we did 100 generations). The population size M was 100. Only tournament selection and mutation (mutation rate $p_m = 0.01$) were used (so $\alpha = 1$). In the BC-GA we computed *all* the individuals in the last generation (G). That is, $m_0 = M$. To make a comparison between the algorithms possible, unless otherwise stated, 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.

Fig. 3 shows the fitness vs. generation plots for GA and BC-GA when the tournament size n is 2. It is clear from the figure that BC-GA performs about 20% fewer fitness evaluations than the standard EA, reaching, however, *the same average and maximum fitness values*. As predicted in the previous sections this significant computational saving comes without altering in any substantial way the behaviour of the EA.

Fig. 4 shows the fitness vs. generation plots for the two algorithms when the tournament size n is 3. With this tournament size, there is still a saving of about 6% which is definitely worth having, but clearly for even higher selection pressures the disadvantages of using a BC-GA in terms of memory use and bookkeeping become quickly preponderant.

An important question about BC-EA is how the expected number of fitness evaluations changes as a function of M, n, m_0 and G. In order to assess the impact of both the transient and the limit-distribution behaviour of BC-EA, we performed a series of experiments where the task was to evaluate just one individual at generation G (that is $m_0 = 1$). In these experiments, we set $G = 49$ (i.e., we performed exactly 50 generations, 0 through to 49). This was big enough that all transients had finished well before generation 0, thereby revealing also the limit-distribution sampling behaviour. In the experiments we used populations of size $M = 10$, $M = 1000$ and $M = 100\,000$. So, forward runs

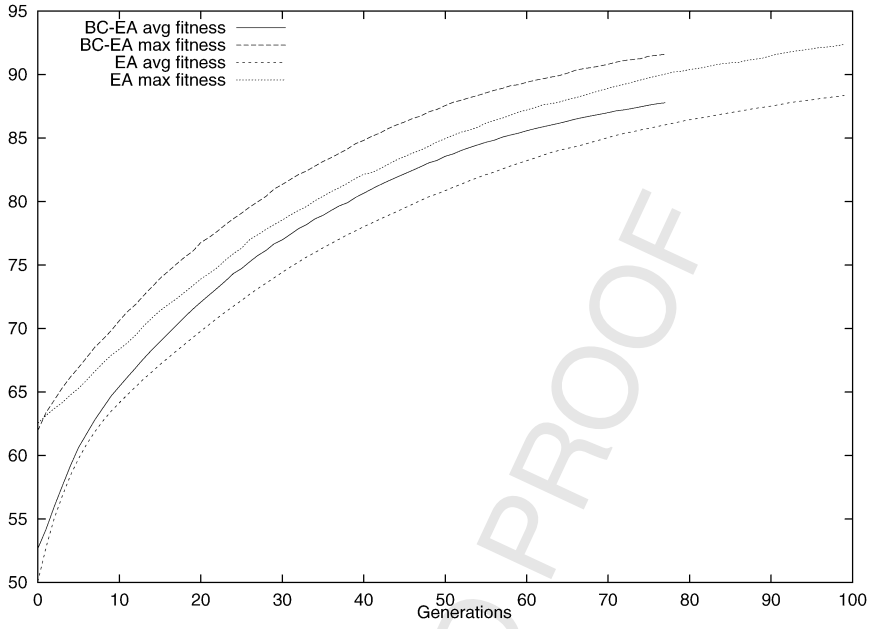

Fig. 3. Comparison between BC-GA and standard GA when a tournament size of 2 is used. Means over 100 independent runs. Note, to make the comparison possible, in the case of the BC-GA each unit on the abscissa axis ("Generations") corresponds $M = 100$ fitness evaluations.

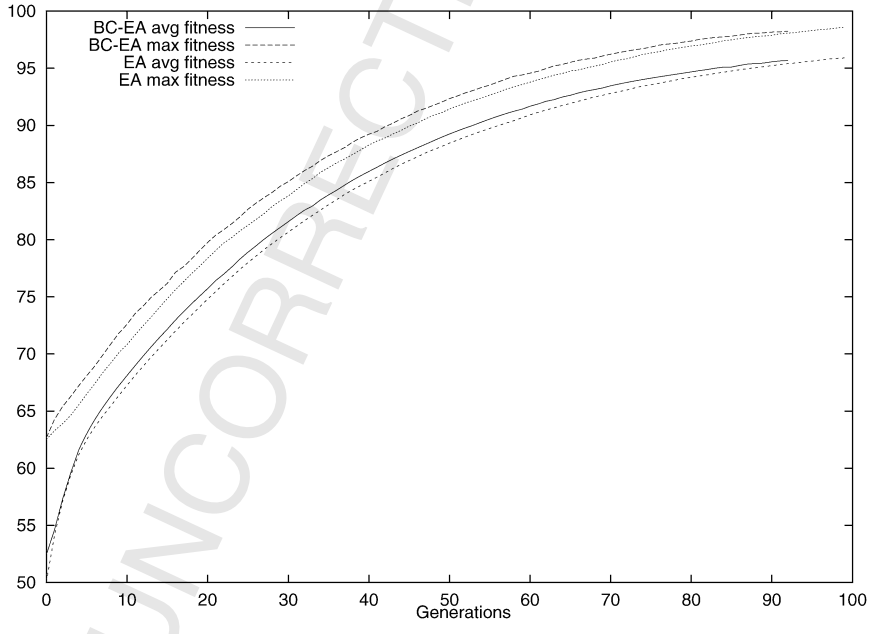

Fig. 4. Comparison between BC-GA and standard GA when a tournament size of 3 is used. Means over 100 independent runs.

required exactly $F^F = 500$, $F^F = 50\,000$ and $F^F = 5\,000\,000$ fitness evaluations to complete. For each setting we did 100 independent runs.

Fig. 5 shows the average proportion of individuals evaluated by BC-EA in each generation when *mutation only* is used ($\alpha = 1$) for tournaments sizes $n = 2$ and $n = 3$ as a function of the population size M, while Fig. 6 shows the average proportion of individuals evaluated by a BC-EA with *one-offspring crossover* with $p_c = 0.5$ ($\alpha = 1.5$) for the same tournaments sizes.

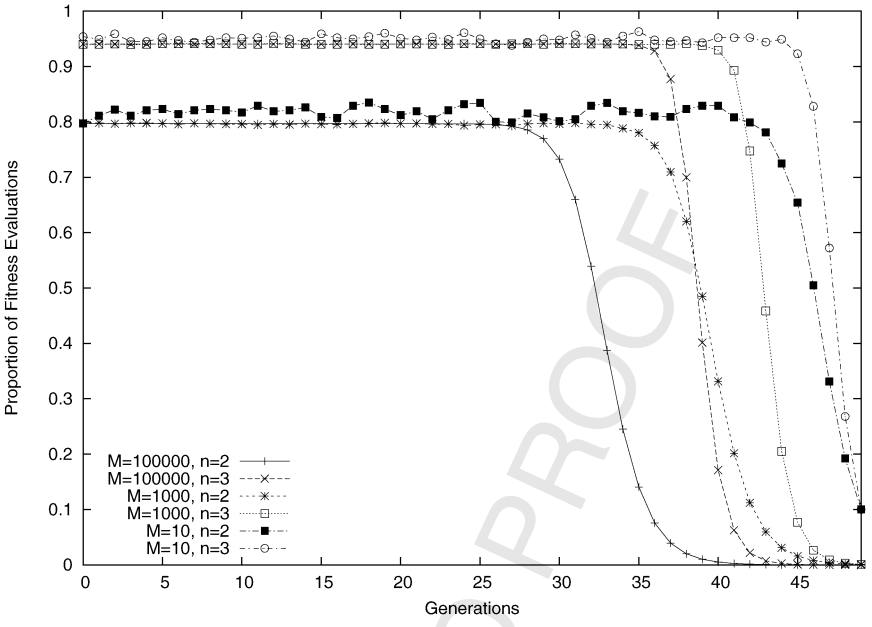

Fig. 5. Average proportion of individuals evaluated by BC-EA when mutation only is used ($\alpha = 1$) for tournaments sizes $n = 2$ and $n = 3$. Means over 100 independent runs.

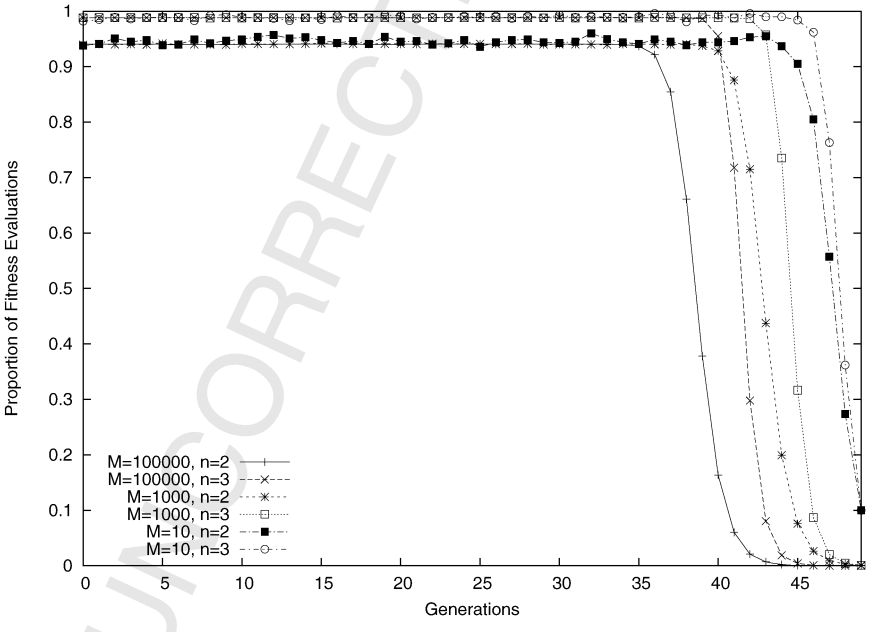

Fig. 6. Average proportion of individuals evaluated by BC-EA when one-offspring crossover with $p_c = 0.5$ is used ($\alpha = 1.5$) for tournaments sizes $n = 2$ and $n = 3$. Means over 100 independent runs.

From these figures we can see that, as expected, the limit-distribution saving is largely independent from the size of the population. E.g., for $n = 2$, after the transient about 80% of the population is an "ancestor" of the individual of interest in generation G if mutation only is used, while this goes up to 94% when $\alpha = 1.5$. For EAs where long runs are used, these percentages provide an approximate estimation of the total proportion of fitness evaluations required by a backward chaining version of the algorithm w.r.t. the standard algorithm.

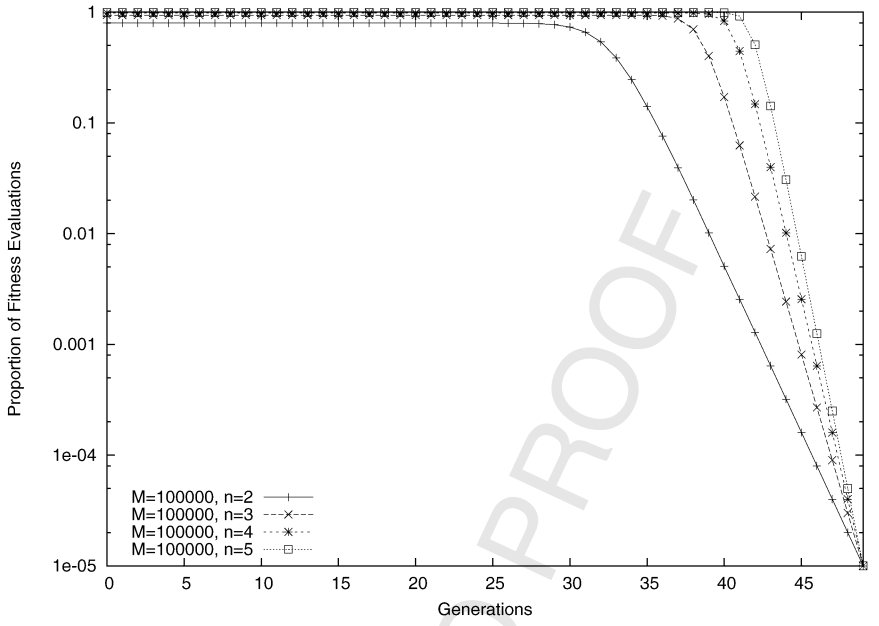

Fig. 7. Logarithmic plot of the average proportion of individuals evaluated by BC-EA when mutation only is used ($\alpha = 1$) for tournaments sizes $n = 2$–5 and a population size $M = 100\,000$.

Table 1
Mean number of fitness evaluations recorded during 50 generations in the experiments shown in Figs. 5 and 6 as a percentage of the fitness evaluations required by a forward EA (reported in the last column for reference)

M	BC-EA + mutation		BC-EA + one-offspring crossover		Forward EA
	$n = 2$	$n = 3$	$n = 2$	$n = 3$	
10	76.8%	90.8%	90.4%	95.2%	500
1 000	64.0%	81.9%	81.8%	89.2%	50 000
100 000	53.4%	74.1%	73.8%	83.2%	5 000 000

Figs. 5 and 6 also show that during most of the transient the number of individuals sampled by tournament selection, m_t, grows very quickly (backward from generation 49). As clearly shown in Fig. 7, the growth is exponential as predicted in Section 6.3. The rapid growth lasts for $g_e \approx 18$ generations for $n = 2$, for $g_e \approx 12$ generations for $n = 3$, for $g_e \approx 9$ generations for $n = 4$, and for $g_e \approx 8$ generations for $n = 5$ which confirms the accuracy of the approximation in Eq. (3) (that predicts g_e values of approximately 17, 10, 8 and 7, respectively).

Even when runs last for more than g_e generations, the effects of the exponential transient are marked. To illustrate this, Table 1 reports the mean *total* number of fitness evaluations recorded in the experiments shown in Figs. 5 and 6 as a percentage of the standard EA fitness evaluations, $F^F = (G + 1) \times M$. Taking, for example, the case of $n = 2$ and no mutation, where the limit distribution effort would be around 80%, we can see that efforts of as low as 53.4% of those required by a forward EA are achieved.[10]

9.3. GP vs. BC-GP

The function set for GP included the functions $+$, $-$, $\times$ and the "protected" division DIV where $\mathrm{DIV}(x, y) = x/y$ unless $|y| <= 0.001$, in which case $\mathrm{DIV}(x, y) = x$ to avoid run-time errors. 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

[10] The similarity between the numbers in the third and fourth columns of the table are not a mistake: both mutation with tournament size $n = 3$ and one-offspring crossover with tournament size $n = 2$ and $p_c = 0.5$ require the same average number of selection steps, $3M$.

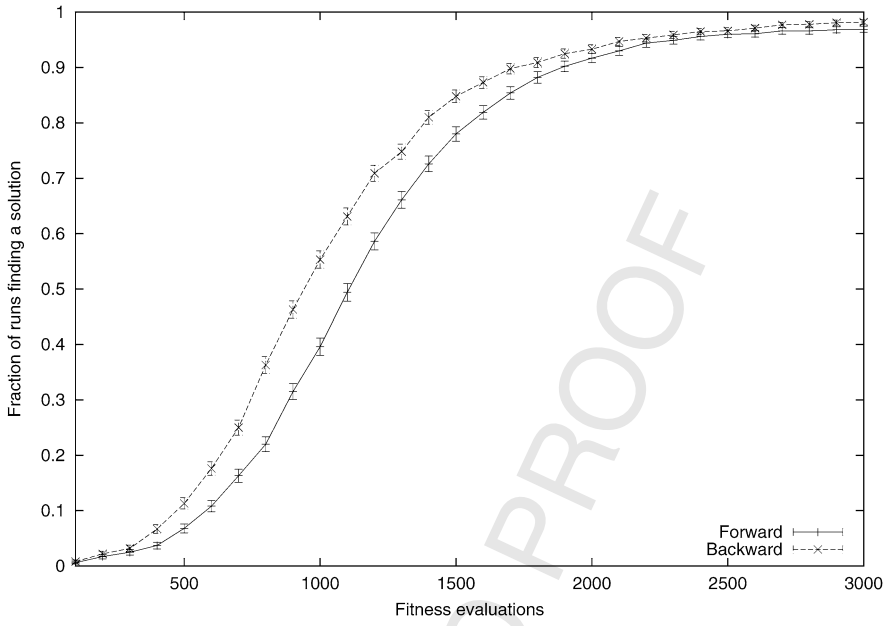

Fig. 8. Quartic polynomial regression problem. Normal GP contrasted with chance of success with BC-GP (population size 100, average over 1000 runs).

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 [18] with maximum 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.

For the purpose of comparing the problem solving ability of GP and BC-GP, we gave both algorithms the same number of fitness evaluations. The maximum number of fitness evaluations was $30M$, which for standard GP corresponds to 30 generations. 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 the BC-GP we computed 80% of the final generation (i.e. $m_0 = 0.8M$) since this is approximately the steady state value of m_t for $n = 2$ and $t \to \infty$.

Figs. 8 and 9 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 (Fig. 9) 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 and requires large populations to be solvable in most runs. Fig. 10 shows the fraction of successful runs with a population of 1 000. Fig. 11 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.

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 Fig. 12 for the case $M = 10\,000$, BC-GP on average finds better programs for the same number of fitness evaluations.

In Figs. 8–12 we have compared forward GP and BC-GP when both algorithms are given the same number of fitness evaluations. Instead, in Table 2, we show a comparison when they are run for the same number of generations ($G = 30$). Like for the BC-GA, thanks to the savings obtained by avoiding the creation and evaluation of individuals not sampled by selection (and their unnecessary ancestors), by the end of the runs, BC-GP evolved solutions of similar fitness (which again confirms the statistical equivalence of EAs and BC-EAs), but took around 20% fewer fitness evaluations. Similar savings are obtained at all population sizes.

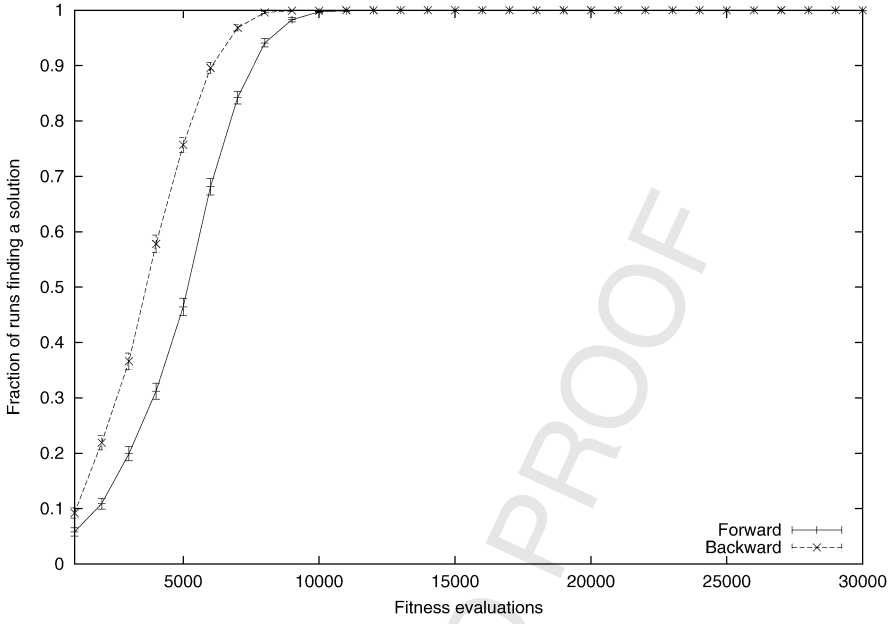

Fig. 9. Quartic polynomial regression problem. As Fig. 8 but with population of 1000.

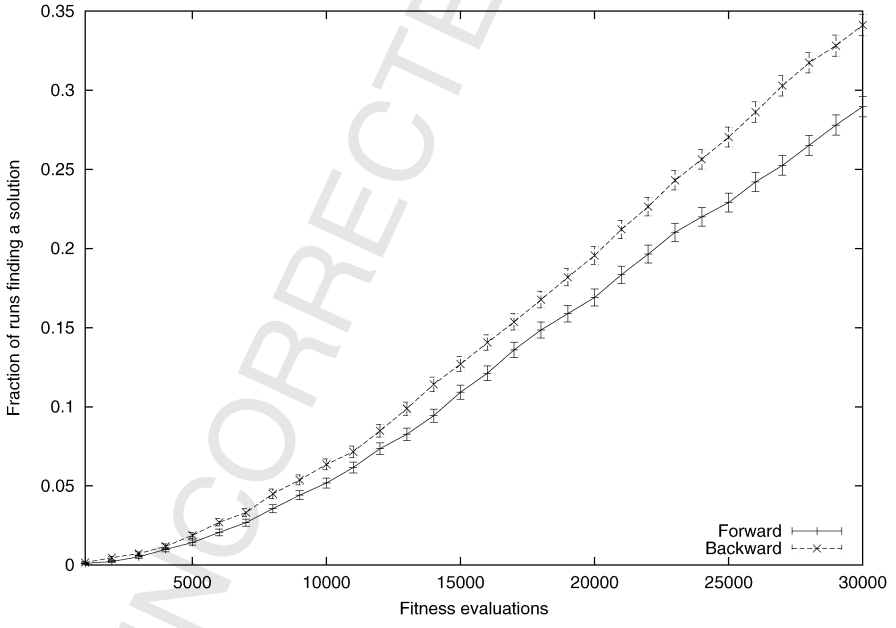

Fig. 10. Fraction of successful runs (out of 5000 runs) on the Poly-4 problem for forward GP and BC-GP (30 generations) with populations of 1 000.

All the tests reported in this section have been performed also for the case of tournament size $n = 3$. We don't report on these for brevity. In all cases BC-GP was superior to GP, but, naturally, by a smaller margin.

9.4. Wall-clock execution-time comparison

To evaluate whether paging and disk access for memory had an impact in our experiments, we considered the most demanding of our representations—variable-size GP trees—and ran a series of experiments measuring wall-clock

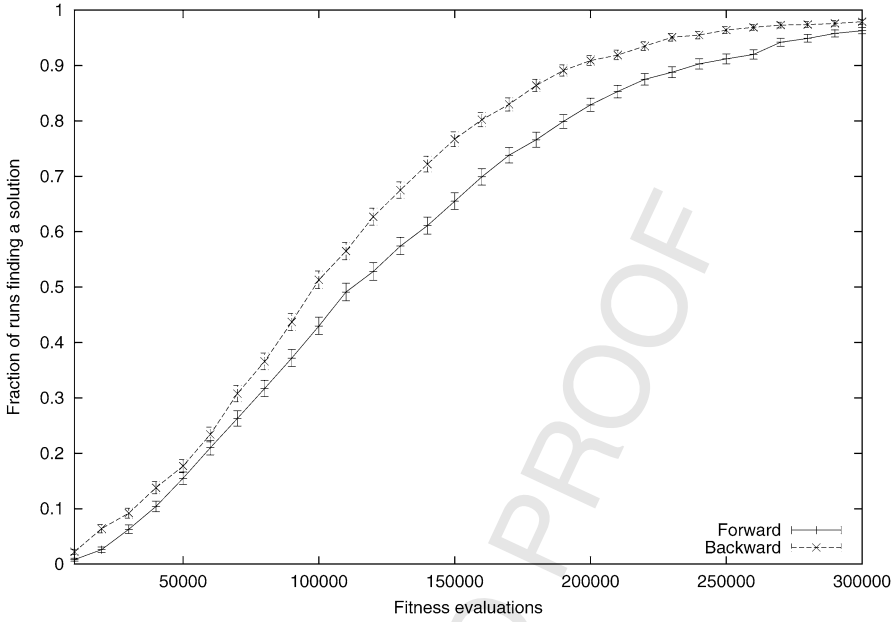

Fig. 11. Fraction of successful runs (out of 1000 runs) on the Poly-4 problem for forward GP and BC-GP with populations of 10 000.

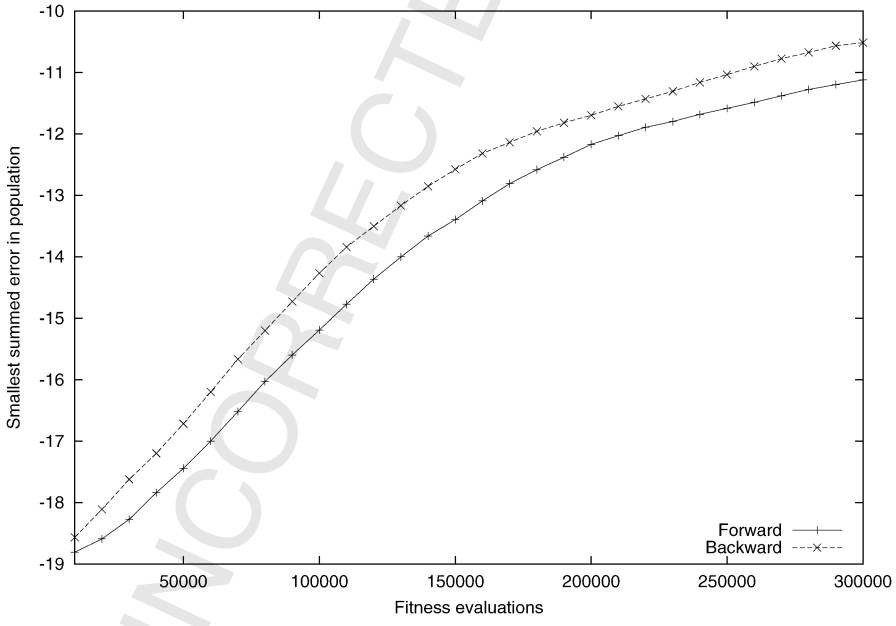

Fig. 12. Error summed over 50 test cases for Poly-10 regression problem (means of 1000 runs, with populations of 10 000).

Table 2
Normal GP v. Backward chaining on Quartic, Poly 4 and Poly 10. Population 10 000. Generations 30. Means of 1 000 runs

Problem	Forward			Backward			Saving
	Best Fit	Evals	Succ Prob	Best Fit	Evals	Succ Prob	
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%

Table 3
Wall-clock per-primitive execution-time comparison between GP and BC-GP

	Population size			
	100	1 000	10 000	100 000
GP	0.782 µs	0.596 µs	0.592 µs	0.599 µs
BC-GP	0.819 µs	0.598 µs	0.591 µs	0.595 µs

execution times for both forward and backward GP and different population sizes applied to the Quartic polynomial problem. Each run lasted for 50 generations. All other GP parameters were as in Section 9.3. The results are shown in Table 3. Results are the average execution time per primitive (in microseconds). Averages were computed over 10 independent runs by dividing the *total* execution time of the runs by the number of primitives executed in the runs. For a fairer comparison we report execution time per primitive instead of total execution time since BC-GP runs used around 20% fewer fitness evaluations that GP runs. Runs were performed on a 3 GHz Linux PC with 2 GB of memory.

As one can clearly see in Table 3, even with very large populations, there were no significant differences in per-primitive execution times between GP and BC-GP. Also, with the exception of populations of 100 individuals, there were no significant differences in execution times as the population size was varied. The higher execution time in populations of 100 individuals is an artifact due to the code for collecting statistics requiring a non-negligible proportion of the computation time at such small population sizes.

10. Discussion

In this paper we have focused on a source of inefficiency in the sampling behaviour of tournament selection: the creation and evaluation of individuals that cannot influence future generations. We have proposed general methods to remove this source of inefficiency and speed up EAs based on tournament selection. One of these methods, the backward chaining EA, provides the additional benefit of converging faster than a standard (forward) algorithm due to its constructing and evaluating individuals belonging to later generations sooner. We have analysed these algorithms both theoretically (Section 6) and experimentally (Section 9), strongly corroborating the feasibility of this approach.

The implementation of a backward chaining EA is not very complex and the added book keeping required is quite limited. However, there is no doubt that BC-EAs require more memory than their forward counterparts. If one, however, is prepared to accept this overhead and adopt the ideas behind BC-EA, the computational savings can be very big. These are achievable not only when we exploit the transient behaviour of the algorithm, but also in the limit-distribution behaviour, as will be illustrated below.

Maximum savings are achieved when $n\alpha$ is minimum. The smallest value α can take is 1 and with standard tournament selection the minimum for n is 2. So, we already know that the best we can do is saving around 20% fitness evaluations. However, a form of tournament selection exists (e.g., see [12,22]) that we can modify to obtain even more spectacular savings.

In this form of tournament selection, one picks up two individuals at random and then chooses the one with the higher fitness with probability p, the other with probability $1 - p$. For $p = 1$ this form of selection is equivalent to standard tournament selection with $n = 2$, while it is a form of random selection for $p = 0.5$. By acting on p it is possible to vary the selection pressure of the method continuously between these two extremes. An alternative description of the method is that we choose the higher fitness individual with probability q and randomly between the two with probability $1 - q$ (naturally $p = q + (1 - q)/2 = (1 + q)/2$). In this case q can be varied in the interval $[0, 1]$.

This second version of the algorithm can be modified for our purposes. Instead of first choosing a pair of individuals and then deciding whether we select the best or we pick one at random, we, equivalently, first decide which selection strategy we are going to use, and then, based on this, we randomly draw individuals from the population. If we decide to go for the best in the tournament, then we must draw two individuals from the population. However, if we decide to choose randomly between the two members of the tournament, then we can just draw one random individual from the population (instead of drawing two individuals and then randomly discarding one).

With this method, the expected number of individuals drawn in each tournament is $n = 2 \times q + 1 \times (1 - q) = q + 1 \leqslant 2$. So, clearly the smaller q the bigger the saving we should expect in a BC-EA. Just to get a feel for the order of magnitude of these savings, let us assume $\alpha = 1$ and let us use Eq. (1) (Section 6.1) to estimate the expected

proportion of individuals not sampled. This is approximately $e^{-(1+q)}$. So, for very low selection pressures saving of over 35% fitness evaluations are possible.

Naturally, much more substantial savings can be obtained when exploiting the transient behaviour of BC-EAs. In Section 9.2 we showed that when running a BC-EA with $m_0 = 1$, the effort is of the order of the population size. However, the reader will probably wonder about the usefulness of evaluating just one individual in the last generation. Normally we would want to have the whole generation. However, we need to remember that the individual provided by a BC-EA (with $m_0 = 1$) at generation G is effectively a random sample drawn from the population at that generation. Although we expect one individual to be insufficient, one important question is whether we really need to have the whole of generation G in order to solve a problem, particularly considering that in many EAs there is substantial loss of genotypic diversity in the population in the late phases of a run. In [27] we have experimented with an implementation of BC-GP with one-offspring crossover showing that even when run with $m_0 = 1$ BC-GP can solve problems. So, in at least some cases, we do not need the whole population. To get a more complete and satisfactory answer, future work on BC-EAs will need to include a thorough investigation of the best way to choose m_0 and G.

11. Conclusions

In this paper, we have analysed the sampling behaviour of tournament selection over multiple generations and used this analysis to come up with and demonstrate more efficient implementations of evolutionary algorithms (EAs) that are much more rooted in classical AI than any other previous class of EAs. In particular we have proposed a new way of running EAs, the backward chaining-EA (BC-EA), which offers a combination of fast convergence, increased efficiency in terms of fitness evaluations, complete statistical equivalence to a standard EA and broad applicability. Because of these interesting properties we think the class of BC-EAs is an area worthy of further investigation.

To reiterate, the BC-EA algorithm is not hard to implement, as we have discussed in Section 7. Also, BC-EA tends to find better individuals faster irrespective of the tournament sizes. However, if one wants to use tournaments with more than three individuals and to compute a large proportion of the final generation, the computational saving provided by BC-EA 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-EA can be fruitfully applied even for large tournament size. For example, with BC-EA, tournament size 7, and a population of a million individuals—which is not unusual in some EAs such as GP—one could calculate 1 individual at generation 7, 7 individuals at generation 6, 49 individuals at generation 5, etc. at a cost inferior to that required to initialise the population in a forward EA. The information gained in this way about future generations could prove very important, for example, in deciding whether to continue a run or not. This information is certainly not available in a traditional EA.

In future research we intend to test the new algorithm on real-world problems, and explore possible ways of further improving the allocation of trials and decision making in BC-EAs, for example, by replacing our current depth-first-search strategy with an informed search algorithm, such as, perhaps, A^*. Applying a backward chaining approach to other forms of local selection, beyond tournament selection, is another promising area for future research.

Acknowledgements

We would like to thank Chris Stephens, Darrell Whitley, Kumara Sastry and Bob McKay for their useful comments. The reviewers and the co-editor-in-chief in charge of this manuscript are also warmly thanked for their help in improving it.

References

[1] T. Baeck, D.B. Fogel, Z. Michalewicz (Eds.), Oxford Univ. Press, Oxford, 1997.
[2] T. Bäck, D.B. Fogel, T. Michalewicz (Eds.), Evolutionary Computation 1: Basic Algorithms and Operators, Institute of Physics Publishing, 2000.
[3] W. Banzhaf, P. Nordin, R.E. Keller, F.D. Francone, Genetic Programming—An Introduction; On the Automatic Evolution of Computer Programs and its Applications, Morgan Kaufmann, January 1998, dpunkt.verlag.
[4] T. Blickle, L. Thiele, A mathematical analysis of tournament selection, in: L.J. Eshelman (Ed.), Proceedings of the Sixth International Conference on Genetic Algorithms (ICGA'95), San Francisco, CA, Morgan Kaufmann, 1995, pp. 9–16.

[5] T. Blickle, L. Thiele, A comparison of selection schemes used in evolutionary algorithms, Evolutionary Computation 4 (4) (1997) 361–394.
[6] L. Davis (Ed.), Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, 1991.
[7] T.E. Davis, J.C. Principe, A Markov chain framework for the simple genetic algorithm, Evolutionary Computation 1 (3) (1993) 269–288.
[8] K.A. De Jong, W.M. Spears, D.F. Gordon, Using Markov chains to analyze GAFOs, in: L.D. Whitley, M.D. Vose (Eds.), Proceedings of the Third Workshop on Foundations of Genetic Algorithms, San Francisco, CA, July 31–August 2 1995, Morgan Kaufmann, 1995, pp. 115–138.
[9] W. Feller, An Introduction to Probability Theory and Its Applications, vol. 2, John Wiley, 1971.
[10] D.B. Fogel (Ed.), Evolutionary Computation. The Fossil Record. Selected Readings on the History of Evolutionary Computation, IEEE Press, 1998.
[11] L.J. Fogel, A.J. Owens, M.J. Walsh, Artificial Intelligence through Simulated Evolution, Wiley, New York, 1966.
[12] D.E. Goldberg, K. Deb, A comparative analysis of selection schemes used in genetic algorithms, in: G.J.E. Rawlins (Ed.), Foundations of Genetic Algorithms, Morgan Kaufmann, 1991.
[13] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading MA, 1989.
[14] J. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, MI, 1975.
[15] J. He, X. Yao, Drift analysis and average time complexity of evolutionary algorithms, Artificial Intelligence 127 (1) (2001) 57–85.
[16] J. He, X. Yao, Towards an analytic framework for analysing the computation time of evolutionary algorithms, Artificial Intelligence 145 (1–2) (2003) 59–97.
[17] J.R. Koza, M.A. Keane, M.J. Streeter, W. Mydlowec, J. Yu, G. Lanza, Genetic Programming IV: Routine Human-Competitive Machine Intelligence, Kluwer Academic, 2003.
[18] J.R. Koza, A genetic approach to the truck backer upper problem and the inter-twined spiral problem, in: Proceedings of IJCNN International Joint Conference on Neural Networks, vol. IV, IEEE Press, 1992, pp. 310–318.
[19] W.B. Langdon, R. Poli, Foundations of Genetic Programming, Springer-Verlag, Berlin, 2002.
[20] W.B. Langdon, T. Soule, R. Poli, J.A. Foster, The evolution of size and shape, in: L. Spector, W.B. Langdon, U.-M. O'Reilly, P.J. Angeline (Eds.), Advances in Genetic Programming 3, MIT Press, Cambridge, MA, June 1999, pp. 163–190. Chapter 8.
[21] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, second ed., Springer-Verlag, Berlin, 1994.
[22] M. Mitchell, An Introduction to Genetic Algorithms, MIT Press, Cambridge, MA, 1996.
[23] T. Motoki, Calculating the expected loss of diversity of selection schemes, Evolutionary Computation 10 (4) (2002) 397–422.
[24] N.F. McPhee, R. Poli, J.E. Rowe, A schema theory analysis of mutation size biases in genetic programming with linear representations, in: Proceedings of the 2001 Congress on Evolutionary Computation CEC2001, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27–30 May 2001, IEEE Press, 2001, pp. 1078–1085.
[25] A.E. Nix, M.D. Vose, Modeling genetic algorithms with Markov chains, Annals of Mathematics and Artificial Intelligence 5 (1992) 79–88.
[26] R. Poli, B. Logan, The evolutionary computation cookbook: Recipes for designing new algorithms, in: Proceedings of the Second Online Workshop on Evolutionary Computation, Nagoya, Japan, March 1996.
[27] R. Poli, W.B. Langdon, Backward-chaining genetic programming, in: H.-G. Beyer, U.-M. O'Reilly, D.V. Arnold, W. Banzhaf, C. Blum, E.W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J.A. Foster, E.D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G.R. Raidl, T. Soule, A.M. Tyrrell, J.-P. Watson, E. Zitzler (Eds.), GECCO 2005: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, vol. 2, Washington, DC, 25–29 June 2005, ACM Press, 2005, pp. 1777–1778.
[28] R. Poli, N.F. McPhee, General schema theory for genetic programming with subtree-swapping crossover: Part I, Evolutionary Computation 11 (1) (March 2003) 53–66.
[29] R. Poli, N.F. McPhee, General schema theory for genetic programming with subtree-swapping crossover: Part II, Evolutionary Computation 11 (2) (June 2003) 169–206.
[30] R. Poli, N.F. McPhee, J.E. Rowe, Exact schema theory and Markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover, Genetic Programming and Evolvable Machines 5 (1) (March 2004) 31–70.
[31] R. Poli, Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover, Genetic Programming and Evolvable Machines 2 (2) (June 2001) 123–163.
[32] 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.
[33] R. Poli, J.E. Rowe, N.F. McPhee, Markov chain models for GP and variable-length GAs with homologous crossover, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), San Francisco, CA, 7–11 July 2001, Morgan Kaufmann, 2001.
[34] I. Rechenberg, Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution, Frommann–Holzboog, Stuttgart, 1973.
[35] S.J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, second ed., Prentice Hall, Englewood Cliffs, NJ, 2003.
[36] G. Rudolph, Convergence analysis of canonical genetic algorithm, IEEE Transactions on Neural Networks 5 (1) (1994) 96–101.
[37] G. Rudolph, Genetic algorithms, in: T. Baeck, D.B. Fogel, Z. Michalewicz (Eds.), Handbook of Evolutionary Computation, Oxford University Press, Oxford, 1997, pp. B2.4-20–27.
[38] G. Rudolph, Models of stochastic convergence, in: T. Baeck, D.B. Fogel, Z. Michalewicz (Eds.), Handbook of Evolutionary Computation, Oxford University Press, Oxford, 1997, pp. B2.3-1–3.
[39] G. Rudolph, Stochastic processes, in: T. Baeck, D.B. Fogel, Z. Michalewicz (Eds.), Handbook of Evolutionary Computation, Oxford University Press, Oxford, 1997, pp. B2.2-1–8.
[40] H.-P. Schwefel, Numerical Optimization of Computer Models, Wiley, Chichester, 1981.
[41] K. Sastry, D.E. Goldberg, Modeling tournament selection with replacement using apparent added noise, in: Proceedings of ANNIE 2001, vol. 11, 2001, pp. 129–134.

[42] C.R. Stephens, Some exact results from a coarse grained formulation of genetic dynamics, in: L. Spector, E.D. Goodman, A. Wu, W.B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M.H. Garzon, E. Burke (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), San Francisco, CA, 7–11 July 2001, Morgan Kaufmann, 2001, pp. 631–638.
[43] C.R. Stephens, H. Waelbroeck, Effective degrees of freedom in genetic algorithms and the block hypothesis, in: T. Bäck (Ed.), Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97), East Lansing, Morgan Kaufmann, 1997, pp. 34–40.
[44] C.R. Stephens, H. Waelbroeck, Schemata evolution and building blocks, Evolutionary Computation 7 (2) (1999) 109–124.
[45] A. Sokolov, D. Whitley, Unbiased tournament selection, in: H.-G. Beyer, U.-M. O'Reilly, D.V. Arnold, W. Banzhaf, C. Blum, E.W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J.A. Foster, E.D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G.R. Raidl, T. Soule, A.M. Tyrrell, J.-P. Watson, E. Zitzler (Eds.), GECCO 2005 Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, vol. 2, Washington, DC, 25–29 June, ACM Press, 2005, pp. 1131–1138.
[46] A. Teller, 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, R.L. Riolo (Eds.), Genetic Programming 1997: Proceedings of the Second Annual Conference, Stanford University, CA, 13–16 July 1997, Morgan Kaufmann, 1997, pp. 321–328.
[47] M.D. Vose, The Simple Genetic Algorithm: Foundations and Theory, MIT Press, Cambridge, MA, 1999.
[48] P.H. Winston, Artificial Intelligence, third ed., Addison-Wesley, Reading, MA, 1992.