Simple Incremental Testing

John R. Woodward

Birmingham University, Birmingham B15 2TT, UK Email:
J.R.Woodward@cs.bham.ac.uk

Abstract. Additional test cases are added one by one, as the population solves the current set of test cases, until some fixed final limit is reached. We examine the general nature of this approach.
Complexity is defined in a general sense. It is proved that adding a test case to a test set never reduces the complexity of a solution, and never increases the probability of finding a solution. The terms *representative* and *redundant*, are formally defined.
The variation in the number of test cases and the jumps in the number of test cases are observed. The size of the test set just before a general solution is found, indicates a threshold number of test cases required for generalization. We observe, how generalization varies with the size of the test set. Finally we observe the number of successes per evaluation required to produce a general solution.

1 Introduction

Standard Genetic Programming (SGP) [BNKF98] is a test and generate approach which uses a fixed size test set. Selection of this set is important as most of the time is spent testing. If the set is too small, we are unlikely to induce a general function. If the set is too large, time is wasted. It would be desirable to have a method which reduces the time spent testing.

Researchers have started to address this issue (sec. 2). The intuition behind these approaches is simple to understand and usually involves reducing the size of the test set. In all cases they show a speed up without losing generality in their solutions. We have a slightly different motive; to investigate the *nature* of the approach, rather than specifically looking for a speed up.

The philosophy behind this approach is simple: we must learn to walk before we can run. In the initial generations of SGP it is unlikely that we will find a solution which will pass all of the test cases. In effect we have bitten off more than we can chew. Any structured learning would present simpler problems initially and gradually increase the difficulty as the learner improves. Gradually increasing the number of test cases is a simple way to increase the difficulty (as opposed to exchanging 'easy' test cases for 'hard' test cases, keeping the test set a fixed size). Thus we start with a single test case and add only a single test case when the current set is solved.

The Incrementation Algorithm. We describe the schedule we use to increment the number of test cases. Initially we present to the population a test

set consisting of a single randomly selected test case. If and only if an individual solves the test set is another test case added to the test set. This process is continued until either, we have an individual which passes all of the test cases, or a maximum number of evaluations is exceeded. We may end up adding a number of test cases to the test set before an individual fails (this quantity is called the jump step size), but we only add one test case at a time. The fitness of the best individual in the population is therefore always one.

Outline of paper We review the current literature, lay down some basic definitions and theorems. Histograms and graphs are plotted to investigate various characteristics concerning the evolutionary process.

2 Literature Review

Hillis (reported in Mitchell[Mit96]) applied GA to find a sorting network. Candidate solutions were co-evolved along with test cases, consisting of unsorted examples. The fitness of the sorting network was the percentage of cases sorted correctly. The fitness of the problem was the percentage of test cases not sorted. Evolving a population of test cases provides a gradually increasingly difficult set of examples for the solutions to tackle. His work achieved an improvement over the approaches without co-evolution.

Gathercole et. al. [GR94a] introduce 3 methods; dynamic subset selection (DSS), historical subset selection (HSS), and random subset selection (RSS). In all 3 methods, the subset of test cases is available to the whole population. DSS; a subset of test cases is selected with a probability based on 'difficulty' and 'age'. HSS; during an initial set of runs using SGP, difficult test cases are identified and then used in a second set of runs using GP+HSS. Repeating the initial stage produces very similar subsets selected by this method, indicating this is a consistent method of identifying difficult test cases. RSS; a test case is selected for the subset with a fixed probability from the set of all test cases. Regarding results, DSS runs outperform RSS. It is inconclusive if GP is better than HSS.

Zhang et. al.[ZC99] give experimental evidence that linearly increasing the subset of test cases during the evolution can significantly reduce the number of fitness evaluations without sacrificing generalization. The test cases, associated with a specific individual, are exchanged between individuals using a crossover operator. A diversity measure ensures test cases belonging to an individual are dissimilar. A speed up factor of around 2 is achieved compared to standard GP, probably because of the linear increase in the number of test cases.

Teller et. al. [TA97] introduces The Rational Allocation of Trials (RAT). The key idea is that, after evaluating the population on some small set of test cases, if an individual is highly likely to win or lose a tournament, no further test cases need to be evaluated on that individual. The optimal number of fitness cases are allocated for each individual based on sampling statistics, rather the reducing the number of test cases. (It would be interesting to use this idea in conjunction with methods which actually reduce the size of the test set). The

results show a very interesting trend. With SGP, as the size of the training set size increases, the number of fitness cases increase exponentially. With RAT, the number of evaluations is *independent* of test set size. This is what one would expect; after evaluating a small initial number of test cases, we may have to make more evaluations in order to distinguish individuals, but this is independent of the number of *unused* test cases. RAT makes improvements in speed without significant degradation in the quality of solutions.

All the methods in sec. 2 report a speed up without affecting generalization. It would be interesting to see how much more improvement in speed could be squeezed out before generalization is affect. There are some interesting differences between these methods. For example, some use a global set of test cases where other methods associate a set of test cases with an individual. All of the methods above require some parameter setting. In our algorithm, no parameters are needed, so we feel we can gain a clearer picture of the nature of incremental testing than a method requiring parameters. Other work includes[GR94a,GR97,GR94b].

3 Complexity and Test Cases

We give a number of definitions, theorems and proofs. A single test case consists of a vector (input values) and a scalar (output value). A problem is represented as a set of test cases, and the two are considered equivalent. To learn or solve a problem, is to search for a program that produces the functionality defined by the set of test cases.

Definition 1. *The size $S(P)$ of a program P, is the number of nodes it contains.*

Definition 2. *The complexity $C(P)$ of a program P is the size of the smallest program functionally equivalent to P on the observed test cases.*

Definition 3. *The complexity $C(T)$ of a set of test cases T is the size of the smallest program functionally equivalent to T.*

Definitions 1 and 2 apply to all the representations used in GP; trees, lists (linear GP), graphs and forests (modular GP) [BNKF98]. If a program P solves a set of test cases T, then $C(P) = C(T)$. (i.e. set of test cases and the program are functionally equivalent on the *observed test cases*).

Theorem 1. *Given two sets of test cases T and V, where $T \subset V$, and program P_T solves T and program P_V solves V*

$$C(P_T) \leq C(P_V)$$

Proof. Assume $C(P_T) \leq C(P_V)$. The programs P_T and P_V can either be the same size or different sizes, giving 3 cases.

$C(P_T) = C(P_V)$, consistent with the assumption.

$C(P_T) < C(P_V)$, consistent with the assumption.

$C(P_T) > C(P_V)$, inconsistent with the assumption, as any program that performs correctly on V must performs correctly on T (as $T \subset V$), and if this were true, we now have a new lower value for the complexity of P_T, contradicting the definition of complexity. Hence case 3 cannot exist.

Consider incrementing a set of test cases by one case. Given a set of test cases T, we can find a program which satisfies these test cases. If we introduce a *single additional* test case making a set V either the program would perform correctly on the new test case, or it would not and we would have to find another program, either of the same complexity, or higher complexity, which satisfies the additional test case. If we could find a program with lower complexity that could solve V, that program would also solve T and we have a smaller complexity for T, which is contradictory. Hence the complexity of a program to solve a set of test cases will not decrease if a test case in added to the test set.

Theorem 2. *Given two sets of test cases T and V, if $T \subset V$, the probability of solving T, $prob(T)$, is greater than or equal to the probability of solving V, $prob(V)$.*

$$prob(T) \geq prob(V).$$

Proof. Given a set of programs, a proportion of which solve V, we are guaranteed that these solve T as $T \subset V$, hence $prob(T) \geq prob(V)$.

This is true for any given search algorithm and representation. It is tempting to state more complex problems are more difficult to solve (i.e. replacing the condition $T \subset V$ with $C(T) \subset C(V)$). This is true *on average*, and a proof is presented in a following paper.

Given a set of test cases we may want to remove 'redundant' test cases but leave the remaining cases 'representative' of the original set. Some of the test cases maybe redundant in that the do not tell us anything new that can be obtained from other test cases. It may be desirable to not include such test cases in the test set. We formally define representative and redundant

Definition 4. *A set of test cases U_1 (where $U_1 \subseteq T$) is said to represent T, iff*

$$C(U_1) = C(T)$$

If we have a set of test cases T we want to learn, and there exists a set U_1, from which we can induce a program than is guaranteed to solve T, then U_1 is representative of T.

Definition 5. *A set of test cases U_2 (where $U_2 \subset T$) is said to be redundant from T iff U_1 is representative of T and $U_1 \cup U_2 = \varnothing$*

Given a test set T which we want to learn, if there is a proper subset U_1 from which we can induce a program which is guaranteed to solve T, then some of the test cases in T are redundant. Any test case not contained in U_1 is redundant. Definitions 4 5 formalize the concepts of representative and redundant, which are often informally talked about when putting together a set of test cases. The definition 3 relate to the *quantity* of test cases. The definitions 4 5 relate to the *quality* of test cases.

4 Experiments and Commentary

We investigate the nature of the method of incrementing the number of test cases. We conduct a number of GP runs, presenting graphs in pairs using two different representations. In the graphs on the left we use a tree based representation with a primitive set P0. In the graphs on the right we use a modular based representation with a primitive set P1. For details of these search algorithms see [Woo04]. The idea of using two different representations is to make conclusions independent of the representation. We use the boolean even 4 parity problem. The parameter setting are listed in table 1. The fitness of an individual is the number of test cases passed (score ranging from 0 to 16). The best individual *always* has a fitness of 1 (unless we find a solution). The set of test cases does not include duplicates. For each graph or histogram presented we describe our motivation for plotting it, then comment on it.

Table 1. Table summarizing parameter settings for experiments

Population size	3
Maximum number of evaluations	10000000
Initial number of test cases	1
Problem	even parity 4
P0	XOR NAND
P1	OR AND NAND NOR

Number of test cases vs. number of evaluations. Fig. 1 shows how the number of test cases varies. In the initial population an individual is found which solves typically only a few test cases. Later, an individual is found which solves a few more test cases. Finally, after finding an individual which can solve between 6 and 8 test cases, we suddenly find an individual which can solve all 16 test cases.

All graphs we looked at have the same typical shape shown in fig. 1. We never observe a linear or smooth increase in the number of test cases, but typically small initial jumps of 1 or 2 in the number of test cases we can solve, eventually followed by a typically large final jump to all test cases.

Frequency of final number of test cases just before generalization. How many test cases do we need to generalize? Typically, as in the graphs in fig. 1, we manage to induce a general solution after between 6 and 8 test cases. We would be lucky to generalize after seeing only one test case, and unlucky not to generalize after seeing 15 test cases. These are the two extremes, but we are more interested in the general distribution. We generate 1000 successful runs, and record the number of test cases solved just before a general solution is found which generalizes to all 16 test cases.

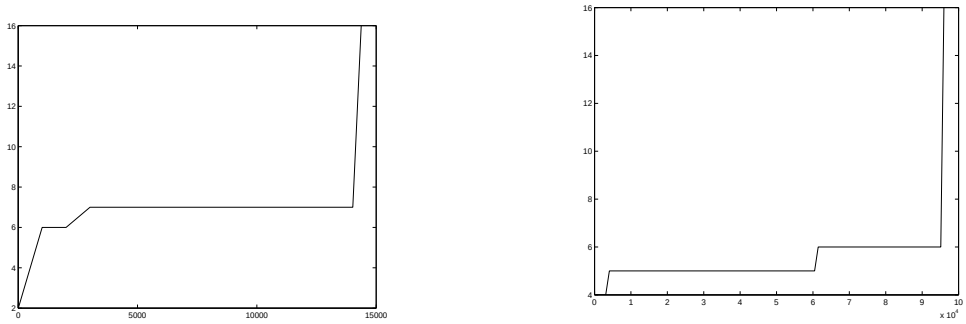

Fig. 1. The horizontal axis is the number of evaluations. The vertical axis is the number of test cases in the test set. Initially only a few test cases are solved. This climbs in incremental jumps. Eventually after solving only a faction of the complete set of test cases, we generate a solution which generalizes to all of the test cases. Note that we do not see a smooth increase in the number of test cases but erratic jumps.

Both histograms (fig 2) have a non symmetric bell shaped distribution. As predicted, very few runs which have observed one test case generalize. Similarly, very few runs reach a test set size of 15, because most runs find an individual which generalizes well before this. Most of the runs managed to generalize after seeing around 8 test cases, indicating a threshold number of test cases needed to be learned before generalization.

Generalization vs final number of test cases. Here (fig. 3) we fix the final number of test case, and plot the number of programs that generalize to all 16 test cases. For example, few programs are likely to generalize after seeing one test case, but all programs generalize after seeing 16 test cases. Generally, the more test cases we test on, the more likely we are to generalize. (Note we are testing generalization ability on all of the test cases, an alternative would be to test on the *remaining* test cases not used in training).

The graphs in fig 3 bear out our expectations. These curves show a monotonic function (sigmoid in fashion), demonstrating that the more test cases we observe, the more likely we are to generalize. It would be interesting to investigate shape of these curves as these may be indications of when to stop adding further test cases. For example, at around 8 test cases, the gradient is at its maximum, so adding a new test case maximally increases the chance of generalization. At around 10 test cases, the gradient begins to flatten out, so we are not getting the same rate of improvements we were observing at 8 test cases, and we receive diminishing returns for further testing.

Frequency of jumps in test set size. In the graphs in fig. 1 we typically observed that the number of test cases increases by only one or two, and we are less likely to observe large jumps. We collected data on these jump sizes in order to study the frequency. Unlike the previous graphs, we also included data

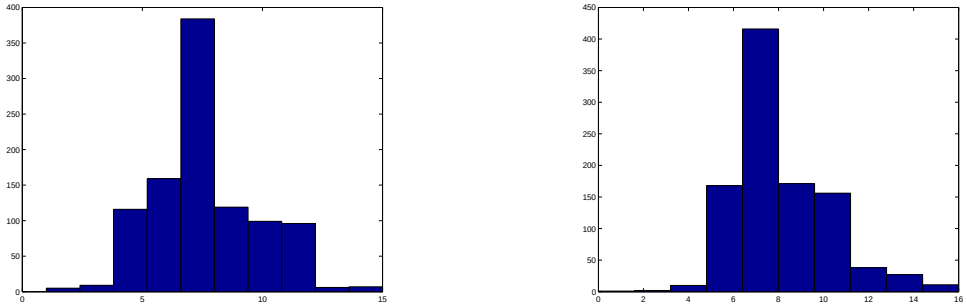

Fig. 2. A histogram showing the frequency of the number of test cases in the test set just before a solution which generalizes is found.

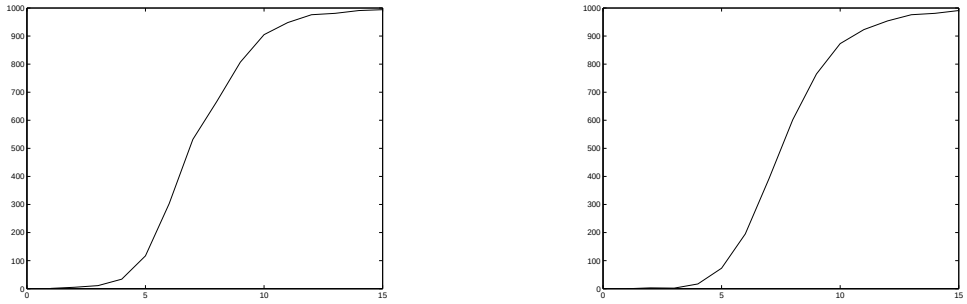

Fig. 3. We fix the final number of test cases (horizontal axis), and then test for generalization on all test cases. The vertical axes show the number of runs which produced programs which generalized. This was done for 1000 runs.

from runs which did not result in a success. This was because we are interested in the gradual improvement of solutions and eventually all runs would result in a solution (as mutation eventually would produce a solution). Also the 'initial' jump could be considered to be from 1 test cases to e.g. 4 test cases, i.e. in the generation of the initial population. We did not include initial jumps, but started collecting data after the population was generated. If we had included this data, it would reflect more about the initial generation method, rather that the dynamics of moving from one generation to the next.

As we expected from our observations of the graphs in fig. 1, large jumps in the number of test cases are less likely than smaller jumps. This supports the idea of only incrementing by a single test case. If we had incremented by two test cases, each time an individual was found which satisfied the current test set,

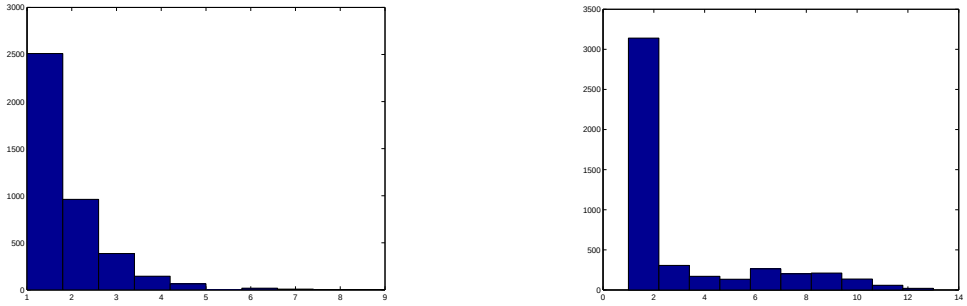

Fig. 4. The frequency in the size of the jumps in test set size. Increments are typically 1 or 2, but larger jumps are observed.

it is likely that it can only solve one of the two new test case, and therefore we are wasting evaluations on the second new test case.

The histogram on the right (modular) is much flatter. The reason we suggest for this is that, during the module part of the search we make a number of attempts at evolving a module, therefore we expect to see bigger jumps.

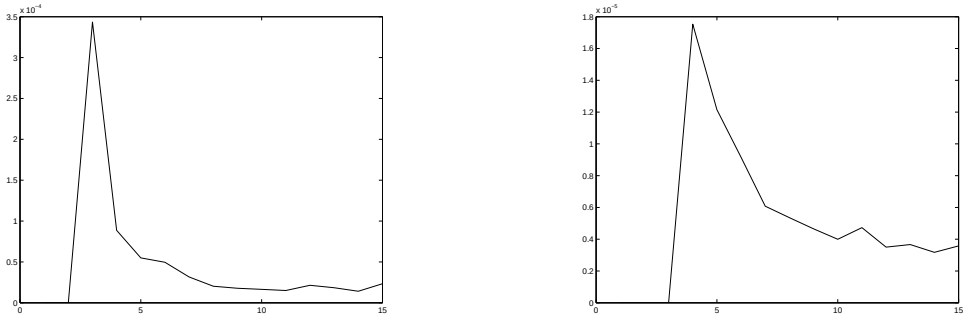

Fig. 5. Shows the number of generalizing solutions per evaluation on the vertical axis (note the scale), against the final test set size on the horizontal axis.

Efficient vs final number of test cases. Graphs in fig. 3 show how the generalization improves as we increase the maximum size of the test set. To generate those curves we generated 1000 *successful* runs and asked the question; given 1000 solutions that solved however many test cases, how many generalized. This is perhaps slightly misleading as we do not take into account the number of evaluations (i.e. more evaluations are required the larger the size of the final test

set). We can take this into account by dividing by the number of evaluations. Fig. 5 shows the number of successes which generalize per evaluation on the vertical axis, for different final test set sizes on the horizontal axis. As expected there is a peak, but this appears more to the left than one may expect by examining fig. 3 alone. We cannot explain the rugged fluctuations on the right of the graphs, observed with a higher final number of test cases. Examination of how the number of evaluations varies with the maximum size of test set also shows these fluctuations.

5 Discussion

In this work we have concentrated on the quantity of test cases, and ignored quality which is equally, if not more, important. We feel quantity is an easier point to tackle as this can be reflected directly by a number. Quality however, is a little more difficult to quantify. We have given formal definitions which can begin to form a basis for further work.

Even if quality is taken into account, this may not be the whole picture. Just because a set of test cases is representative does not mean it is easy to evolve a solution for them; redundancy may aid evolution. Hillis (reported in Mitchell [Mit96]) pointed out, we may want to gradually increase the difficulty of the problem. Ultimately we want to be varying the *complexity* and quality (diversity) of the test cases and the *number* of test cases is secondary.

In all of the methods it is assumed that each test cases take the same amount of time to evaluate. While this is true for some problem domains, it is not true in general, and it may be misleading to conceal computation time by counting the number of evaluations rather than using real clock time.

During this preliminary study none of the graphs here present any real surprises, (except for graphs 5, with its unexplained fluctuations). We make the following observations;

1. there is a threshold number of test cases needed for generalization (fig. 2).
2. as the number of test cases in the test set increases we are more likely to generalize (fig. 3).
3. small increments in the number of test cases are more common than larger jumps (fig. 4).
4. there is an optimal maximum test set size yielding the maximum number of generalizing solutions solutions per evaluation (fig. 5).

All observations appear to be independent of the representation being used, but more work is needed to establish if this is generally true. Having made a number of qualitative observations, we would like to make quantitative predictions to improve efficiently without jeopardizing generalization.

6 Summary and Further Work

We prove adding a test case to a test set never reduces the complexity of a solution and therefore never increases the probability of finding a solution. This

is true for any search operator and representation, which is encouraging, as the empirical observations made appear to be independent of the two representations we use.

We have presented possibly the simplest incremental approach to testing. Our next step is to use a dynamically selected random subset, as this would overcome a few problems (i.e, if a hard test case was included early on, in the current approach, it would remain in the test set for the remainder of the evolution). As pointed out by Gathercole et. al. [GR94a] this dynamic nature may assist evolution. We also plan to further the theoretical work started in this paper.

References

[BNKF98] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, dpunkt.verlag, January 1998.

[GR94a] Chris Gathercole and Peter Ross. Dynamic training subset selection for supervised learning in genetic programming. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature III*, volume 866, pages 312–321, Jerusalem, 9-14 1994. Springer-Verlag.

[GR94b] Chris Gathercole and Peter Ross. Some training subset selection methods for supervised learning in genetic programming. Presented at ECAI'94 Workshop on Applied Genetic and other Evolutionary Algorithms, 1994.

[GR97] Chris Gathercole and Peter Ross. Tackling the boolean even N parity problem with genetic programming and limited-error fitness. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 119–127, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Mit96] Melanie Mitchell. *An Introduction to Genetic Algorithms.* Complex Adaptive Systems. MIT-Press, Cambridge, 1996.

[TA97] Astro Teller and David Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 321–328, Stanford University, CA, USA, 13-16 1997. Morgan Kaufmann.

[Woo04] John R. Woodward. Function set independent genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004)*, Seattle, USA, 26-30 June 2004. Morgan Kaufmann.

[ZC99] Byoung-Tak Zhang and Dong-Yeon Cho. Genetic programming with active data selection. *Lecture Notes in Computer Science*, 1585:146–153, 1999.