Genetic Algorithms and the Automatic Generation of Test Data

Marc Roper, Iain Maclean, Andrew Brooks, James Miller and Murray Wood
Dept. Computer Science,
University of Strathclyde,
Livingstone Tower,
Richmond Street,
Glasgow G1 1XH, U.K.
Tel: +44 (0)141 552 4400 x2956,
Fax: +44 (0)141 552 5330,
email: marc@cs.strath.ac.uk

RR/95/195 [EFoCS-19-95]

Abstract

Although it is well understood to be a generally undecidable problem, a number of attempts have been made over the years to develop systems to automatically generate test data to achieve a level of coverage (branch coverage for example) . These approaches have ranged at early attempts at symbolic execution to more recent dynamic approaches and, despite their variety (and varying degrees of success), all the systems developed have involved a detailed analysis of the program or system under test. In a departure from this approach, this paper describes a system developed to explore the use of genetic algorithms to generate test data to automatically meet a level of coverage.

Genetic algorithms are commonly applied to search problems within AI. They maintain a population of structures that evolve according to rules of selection, mutation and reproduction. Each individual in the environment receives a measure of its fitness in the environment. Reproduction selects individuals with high fitness values in the population, and through crossover and mutation of such individuals, a new population is derived from which individuals may be even better fitted to their environment. Translating these concepts to the problem of test data generation, the population is the set of test data, each element in the set (e.g. a group of data items used in one run of the program) is an individual, and the fitness of an individual corresponds to the coverage it achieves of the program under test.

A system has been developed to support this process. It takes the program to be tested (currently in C) and instruments it with probes to provide feedback on the coverage achieved. The system creates an initial population of random data based on a description of the input data and then performs an iterative search which involves running this data

and measuring its coverage (and hence, fitness). A sample of this population is selected (depending on the fitness value) to go forward to the new population and a proportion of this new population is then subjected to mutation and crossover. The process is then applied again until a maximum level of fitness is reached by the test data. The details of the system are described within the paper and its application demonstrated on several programs. The paper concludes with an evaluation of the system so far and a plan of future work (such as stopping the system from trying to find solutions in the presence of infeasible paths).

Keywords: Automated testing, test data generation, genetic algorithms.

1 Introduction

In this paper we are concerned with the problem of taking an arbitrary program (free from supporting information in the form of a specification, for example) and automatically generating test data to achieve a certain level of coverage of the program (e.g. execute every statement, or the true and false outcomes of every branch etc.). In spite of being an undecidable problem [11] a number of attempts have been made over the years to develop systems to automatically generate test data. Recent developments have made significant progress, but are still hampered by problems - often caused by having to perform a detailed analysis of the program or system under test (for example, to develop symbolic value for input variables or to maintain a trace of the value of variables). The approach described in this paper uses the ideas of Genetic Algorithms (GAs) to automatically develop a set of test data to achieve a level of coverage (branch coverage is used within the paper). Using GAs neatly sidesteps many of the problems encountered by other systems in attempting to automatically generate test data.

The paper gives a brief overview of automatic test data generation followed by an introduction to genetic algorithms. The combination of the two ideas is described in section 4, and section 5 describes the system built to implement this. Section 6 demonstrates the application of the system and section 7 contains our initial analysis and plans for further development.

2 Automatic Test Data Generation

Early automatic test generation tools fell into the category known as pathwise test data generators (the approach and potential problems is well described in [6] and reviewed in other papers — e.g. [9]). In this approach, the program is viewed as a directed graph. A path is then chosen through this graph and is symbolically executed in order to give a predicate for that path (the conjunction of the predicates along the path) which defines a subset of the input space. Choosing values from this input space will cause this path to be taken. If no value can be found then it is indicative of contradictory predicates and an unexecutable path. Even if the path is executable the path predicates will either be linear or non-linear. If they are linear then linear programming techniques may be applied to solve them and thus find a value to follow that path. If they are non-linear then the problem is more difficult and it is necessary to use non-linear programming techniques often coupled with heuristics to increase efficiency. Further problems in dealing with structures such as loops, arrays, module calls, infeasible paths and dynamic data structures have prevented pathwise test data generators moving from research prototypes to commercial products.

More recent approaches are exemplified by the work of Korel[8] and DeMillo and Offut[4, 3]. Korel's approach is to monitor the execution of the input data and if the path followed is not the one selected (i.e., the input data has caused it to stray in some fashion), then the values of input variables which would cause the correct path to be followed are automatically found. This searching activity is speeded up by the use of dynamic data flow analysis to pinpoint the variables responsible for the wrong path. The approach can also handle the presence of arrays and pointers which are reknowned for being notoriously difficult to handle. DeMillo and Offut's system supports mutation testing by automatically generating test data to kill mutants (created

by another tool). The system uses a constraint satisfaction system (which captures the testing requirements imposed by the mutations) along with pathwise analysis and symbolic evaluation techniques. Promising as both these approaches are, they have still encountered difficulties and areas which need further attention (such as handling procedure calls and improving the optimization techniques in the former, and the complexity of the algorithms presenting scalability problems in the latter). Further reviews of testing tools may be found in [2, 7, 1].

3 Genetic Algorithms

Genetic algorithms are commonly applied to a variety of problems involving search and optimisation within the AI domain (see [5, 10] for example). The principle behind GAs is that they create and maintain a *population* of *individuals* represented by *chromosomes* (essentially a character string analagous to the chomosomes appearing in DNA). These chromosomes are typically encoded 'solutions' to a problem (e.g. they may be a sequence of nodes in a graph representing Travelling Salesman tours). The chromosomes then undergo a process of *evolution* according to rules of *selection*, *mutation* and *reproduction*.

Each individual in the environment (represented by a chromosome) receives a measure of its *fitness* in the environment. This is an indication of how successful the individual is at competing in the environment. Reproduction selects individuals with high fitness values in the population, (i.e. those individuals which are considered to be 'successful') and through *crossover* and mutation of such individuals, a new population is derived in which individuals may be even better fitted to their environment. The process of crossover involves two chromosomes swapping chunks of data (genetic information) and is analogous to the process of sexual reproduction. Mutation introduces slight changes into a small proportion of the population and is representative of an evolutionary step.

The process may sound rather haphzard and random but a GA is emphatically not a random search for a solution to a problem. Despite using stochastic processes, the approach is much better than random.

The pseudocode for a simple GA is:

```
initialise(population);
   evaluate(population);
while (not done) do
{
   select(population);
   crossover(population);
   mutate(population);
   evaluate(population);
}
```

The algorithm will iterate until the population has evolved to form a solution to the problem, or until a maxiumum number of iterations have taken place (suggesting that a solution is not going to be found given the resources available).

4 Combining Genetic Algorithms and Test Data Generation

The problem we have to solve is given an arbitrary program, find a set of data that will test to program to a particular level of coverage. In translating the concepts of genetic algorithms to the problem of test data generation we first of all consider our population to be a set of test data. This test data is randomly generated according to the format and type of data used by the program under test (but uses no information about the internal structure of the program). The genetic algorithm is going to take this population and evolve it towards a solution (i.e. the test data will be changed according to the operations of selection, mutation and crossover until it achieves the required level of coverage).

Each individual in the population (a chromosome) is an element in the test data set (i.e. a group of data items used in one run of the program), and the fitness of this chromosome corresponds to the coverage it achieves of the program under test. In this case we are considering branch coverage (although the ideas are equally - and easily - applicable to any level of coverage such as multiple condition coverage, any of the dataflow measures, or other approaches such as mutation analysis), so in a program with two sets of branches (say an `if-then-else` statement inside a `while` loop) a group of data items (individual) which covered all 4 branches would have a fitness level of 1.0, whereas one which covered only two would have a fitness level of 0.5 and so on. The population is evaluated by running the program under test with each individual and assessing the fitness of this individual.

Once the whole population has been evaluated, individuals are selected to contribute towards the next generation (there are a number of ways in which this may be done - the details are discussed in the next section). Each individual may then be subjected to crossover (i.e. swapping elements with another individual) and mutation (effecting small changes in the individual). The likelihood of crossover and mutation taking place is governed by (adjustable) system parameters.

In the traditional GA approach the population would evolve until there was one individual from the whole set which represented the solution (e.g. the nodes in a Hamiltonian path). In our case, this would correspond to one group of data items achieving maximum coverage of the program (exercising the true and false outcomes of every branch, for example). Whilst this is feasible for some programs, the majority of programs cannot be 'covered' by just one group of data items — it might take many groups and several runs of the program to achieve the desired level of testing. So, to make the approach feasible, the population evolves until a combined subset of the population achieves the desired level of coverage. This is done by noting which parts of the program each individual has 'visited' and halting the evolution when a set of individuals have 'visited' the entire program. The solution is this set.

5 The System

A system has been developed in C++ to support this process and its architecture is represented in Figure 1. It prompts for the name of the program to be tested (written in C or C++) and instruments it with probes to provide feedback on the coverage achieved (and hence the fitness

of the chromosome to its environment). This instrumentation process is similar to that carried out by dynamic analysers and inserts probes at the beginning of every block of code - i.e. at the beginning of each function and after the true and false outcomes of each condition. As was mentioned earlier, this is enough to monitor branch coverage. To generate data to achieve a higher level of coverage, either the probe module would have to be enhanced or a proprietary dynamic analyser incorporated into the system. The probes are calls to a function (also automatically inserted into the program under test) which writes the probe number to a file each time it is activated. The probe numbers are unique, starting at 0 and incrementing by 1 for each probe inserted. The instrumented program is then compiled.

The system then prompts for the user to provide a description of the format of the input data (in terms of number of files used, data types and size), and converts this into a chromosome representation (an internal character string representation that the GA can use) of the individual. So, if our input consisted of two integers and a character, each individual would consist of a 9-byte character string (4 for each integer and 1 for the character). The GA maintains this internal representation to allow the crossover and mutation operations to take place easily. A description of the data is also maintained (to allow it to be converted back into its original representation). The system then prompts for details of mutation and crossover rates and the population sizes (the number of individuals in the population).

An initial population of random data is then created and the instrumented program is compiled. The random data generated depends on the data type. Floats have the largest range, integers are constrained to `maxint`, characters to any character and strings to printable characters. Each individual in the population is initialised and stores its data in the internal form.

Once all the system parameters are established and the initial population created, the system then performs the iterative search which proceeds as follows:

1. An individual from the population is selected and its internal data is translated into the data format that the program can use (i.e. 4 characters turn back into and integer etc.). The instrumented program is then executed with this data. For each individual, the route it takes through the program will activate a set of probes (which write their unique number to a file) and after each execution, this file is examined to obtain the level of fitness achieved by the individual (i.e. how much of the program it has visited) and which parts of the program it has covered (this is noted in a bit string associated with each individual).

2. The individual is returned to a temporary population and the bit string which recorded the areas of the program visited is compared with that of the other individuals in this temporary population. If the individuals in this temporary population combined would achieve the maximum level of coverage then the iteration halts and the temporary population becomes the 'solution'. Otherwise step 1 is repeated until there are no more individuals in the population (i.e. the fitness of each has been assessed and they are collected in the temporary population).

3. The individuals in this temporary population are then selected to go forward to the new population. A number of approaches to selecting elements of this population have been implemented — 'above average' which selects those individuals with an above average level of fitness, 'roulette' which favours individuals with a higher level of fitness, and (the one

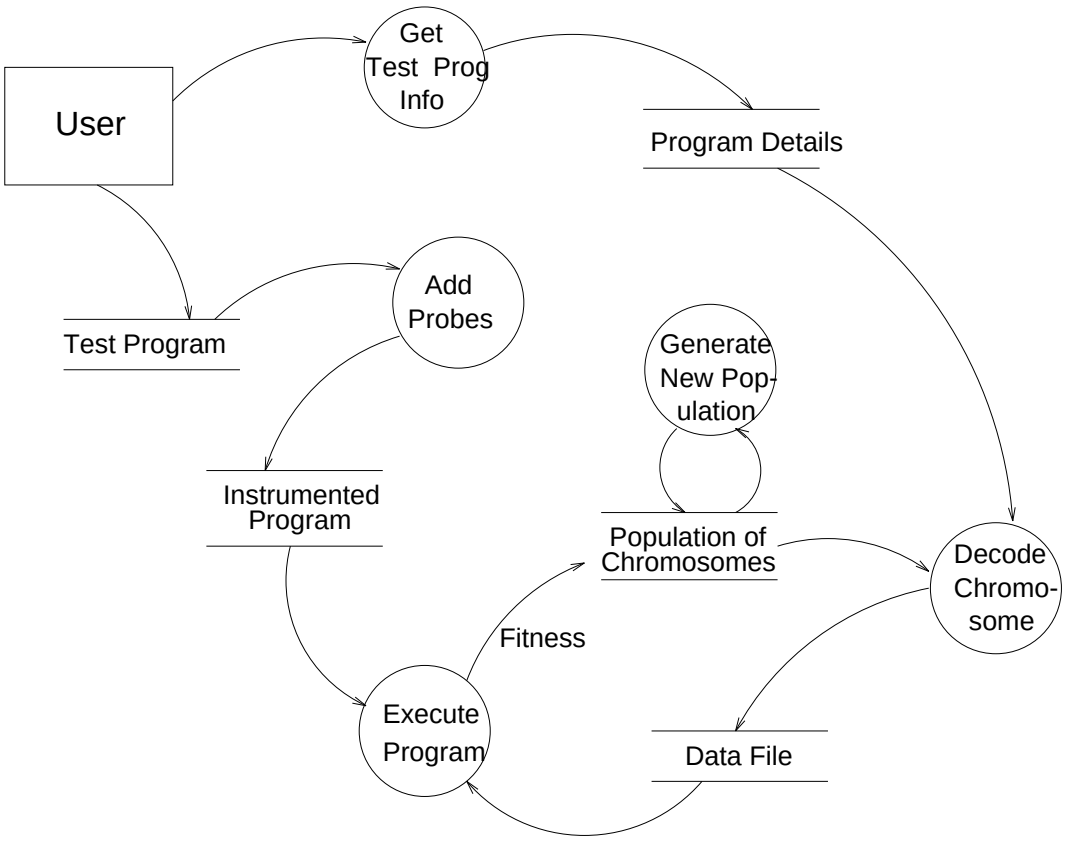

Figure 1: System Architecture

currently being experimented with) 'tournament' which randomly chooses a sub-population and then selects the individual in this sub-population with the highest level of fitness. The selected individuals may then be subjected to crossover and mutation. If crossover does occur then the data belonging to two individuals is broken in half at some point and recombined to generate two new individuals. Mutation works by randomly changing some of the bits in the character representation of the data. In this way the population evolves (i.e. the test data changes).

This process is then repeated until either a maximum level of fitness is reached by a subset of the population or the number of generations (new populations) exceeds a predefined limit which suggests that a solution is not going to be found within the near future.

6 Application

To demonstrate the application of the system, consider the following two small contrived examples. The first example below illustrates the execution of the system (the initial interaction with the system has been deleted to save space) and demonstrates the way in which it gravitates towards a solution. The initial set of data is random and any of this data which is successful (executes the first true branch in the program above, for example) then this data (chromosome) will receive a higher fitness value. This information is fed back into the system and the individual is more likely to contribute to future data. This example is solved in 6 generations (from a population size of 30 - smaller than normal for demonstration purposes) which means that the selection, crossover and mutation processes were carried out 6 times. When necessary, the system redirects the standard input to read from a file. The data below is that read from the file and echoed by the program. Note that the input takes a form of two integers and a character. Sometimes the character does not appear which means that the system has generated an unprintable character.

A second example follows this which involves trying to match a 4-character string. This ends up being solved in 25 generations with a population size of 80. The initial program interaction and (voluminous) output has been deleted leaving the contents of the final temporary generation to demonstrate that the system deals with a set of test data that achieves the required level of coverage (as opposed to one precisely matching datum).

```
/* First Example Program */
#include <stdio.h>
main()
{
  int x,y;
  char c;
  // Read 2 numbers and a character
  scanf("%d %d %c", &x, &y, &c);
  printf("%d %d %c\n",x,y,c);
  if (x>=0 && x < 5000)
    {
     printf("One \n");
     if (y>10000 && y<100000)
       {
        printf("Two \n");
        if (c>='a' && c<='z')
          {
           printf("Three \n");
          }
       }
    }
}
```

1st generation

```
13784 9026 &
4268 489 Œ
One
28711 7066
17421 4224 i
23026 22170 Σ
21314 11542 N
1770 8001
One
14231 11538 ˜
8845 25847 w
10850 4147 5
29445 9257
21069 31118
9168 12781
16394 11992 #
7589 28812 V
18035 24742
31448 25844 ,
26395 1561 )
29517 21042
25980 7455
31375 16719
25258 4639
23246 23907 Φ
12668 12056 p
3069 29623
One
Two
10772 20383 ˜
8297 7284
17480 21807 i
12721 21989
20812 4349 C
```

2nd generation

```
3069 29623
One
Two
545262589 21431
1770 9026 &
One
13784 8001
11261 291767
67372778 8001
13784 9026 &
16344 10050
3069 29623 &
One
Two
268449240 8396613
20978653 2388919
35837 29623
1752 9026 &
One
2889160 139073
1770 8001
One
9680 11074 &
9688 74562 &
612371451 4223927
1770 3904
One
```

```
19449 1061815
4268 321
One
1770 8169 Œ
One
263914 67116865
1770 16785217
One
4268 489 Œ
One
3069 29623
One
Two
1770 8001
One
3037 15937
One
Two
1770 29623
One
Two
1073776362 134225737


Generations 3, 4 and 5 deleted


6th generation

-2147479563 29623
3069 29623
One
Two
67111885 13239
527359 134246839
3069 3167031
One
3069 46007
One
Two
-1409283075 29623
4213759 1078490039
1575677 29623
1051645 31667
-2147480579 95159
3036 29623
One
Two
3065 29623
One
Two
8391645 291767
805309437 29623
3065 29622
One
Two
-2147480579 29623
16780093 30643
2557 4223895
One
3069 539128791
One
8982525 537162679
34081788 15287
527197 8401847
3069 31671
One
Two
1073744733 67138487
536906749 61847
304090109 536900535
3069 29623
One
Two
3061 536900535
One
3005 95159 i
One
Two
Three

Finished with 6 Generations
```

```
/* Second Example */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


void message(void)
{
}

int main()
{
  FILE *fp;
  char name[4];
  if((fp=fopen("/var/tmp/data.dat","r"))==NULL)exit(1);
  read(fileno(fp),name,4);
  printf("%s\n",name);
  if(name[0]=='M'){;}
  if(name[1]=='a'){;}
  if(name[2]=='r'){;}
  if(name[3]=='c'){;}
  message();
}


Finished with 25 Generations
 Contents of temporary population:

H'r#
haOc
`Og
Ha#
H_c
IAa
IiOb
Has
Ok
HaK
H`Oc
MiOc
```

7 Analysis and Conclusions

We have presented an initial exploration into the possibilities of using Genetic Algorithms to automatically generate test data and built a system to support this process. The results of the system so far are encouraging and the system presents several advantages over the more traditional approaches to automatic test data generation in that no analysis or interpretation of the program under test is necessary, the function minimisation problems are avoided and the system can deal with any type of program — procedure and function calls pose no problems, nor do dynamic data types.

There is much more in the way of evaluation that we intend to carry out. For example, changing the mutation rate, crossover rate and population size can greatly affect the speed at which a solution is found. Experimenting with the system so far has found that the best results have been achieved with a mutation rate of 30% at the chromosome level and 5% at the gene level (meaning that there is a probability of 0.3 that a chromosome will be selected for mutation and if selected there is a probability of 0.05 that each gene — a bit in the string — will be changed), a crossover rate of 40% and a population size of 80. Further experimentation involves quantifying the performance of the system on large scale software and evaluating its effectiveness as a testing tool.

There are also a number of possible improvements under investigation. These include basing the initial population on a partial solution (e.g. a set of functional tests) rather than a random population, and use the system to "fill in the gaps" which the functional tests have missed. A reduction in time to achieve an optimal solution may also be improved by storing previous good solutions which have been lost in the reproduction process. A further development involves the problem of stopping the system from trying to find solutions in the presence of infeasible paths. Other enhancements involve controlling the mutation and crossover so they do not 'destroy' their data types - the second example in the application shows a simple mutation can change a string into an unprintable character.

References

[1] C. J. Burgess. Software testing using an automatic generator of test data. In *Proc. SQM 93 - Software Quality Management*, pages 541–556. Elsevier Science Publishers, 1993.

[2] Richard A. DeMillo, W. Michael McCracken, and R. J. Martin a nd John F. Passafiume. *Software Testing and Evaluation.* Benjamin Cummings, 1987.

[3] Richard A. DeMillo and A Jefferson Offut. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering Methodology*, 2(2):109–127, April 1993.

[4] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[5] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[6] J.C. Huang. An approach to program testing. *Computing Surveys*, 7(3):113–127, September 1975.

[7] Darrel Ince. The automatic generation of test data. *The Computer Journal*, 30(1):63–69, 1987.

[8] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[9] Ronald E. Prather and J.Paul Myers, Jr. The path prefix software testing strategy. *IEEE Transactions on Software Engineering*, SE-13(7):761–765, July 1987.

[10] M. Srinivas and Lalit M. Patnaik. Genetic algorithms: A survey. *IEEE Computer*, 27(6):17–26, June 1994.

[11] Elaine J. Weyuker. The applicability of program schema results to programs. *Int. Jnl. Comput. Inform. Sci.*, 8:387–403, October 1979.