# Genetic C Programming with Probabilistic Evaluation

Jacqueline Christmas
Computer Science department
University of Exeter
J.T.Christmas@exeter.ac.uk

## ABSTRACT

We introduce the concept of *probabilistic program evaluation*, whereby the order in which the statements of a proposed program are executed, and whether individual statements are executed at all, are controlled by probability distributions associated with each statement. The sufficient statistics of these probability distributions are mutated as part of the GP scheme. We demonstrate the method on the simple problems of swapping two array elements and identifying the maximum value in an array.

## Keywords

Genetic programming; probabilistic algorithms; C

## 1. INTRODUCTION

We start with the assumption that somewhere within a potential program are at least some of the building blocks necessary to achieve a solution and that the proposed probabilistic evaluation method is able to identify them and the order in which they should be executed.

The fitness of a proposed program in the Genetic Program (GP) [2] population is evaluated by executing it multiple times with different input values and, crucially, different random draws from the probability distributions. Using different input values in each run ensures that the solution is tested for generality. Different random draws from the probability distributions results in different instances, or *versions*, of the proposed program, until the distributions have converged on an ordering and selection of statements.

## 2. THE GP

We explicitly model the programs as language-specific trees (though not with the degree of constraint imposed by [1]) and interpret these trees as part of the fitness evaluation process. Each potential program represents a syntactically correct C program based on the following statement types: assignment, increment (e.g. `x += y`), if-then-else, and while and for loops. At present only integer values and simple arithmetic (+, -, *, /) and conditions (==, !=, >, >=, <, <=) are supported. Variables are integers or arrays of integers and may be constant or updateable. Each potential program is evaluated as though it was preceded by the following declarations:

```
int w = 0;                  const int i0 = 0;
int x = 0;                  const int i1 = 1;
int y = 0;                  const int i2 = 2;
int z = 0;                  const int i3 = 3;
int intList[] = {...};      const int N  = ...;
```

where `intList` is a fixed-size array containing `N` values. The scope of these variables is considered to be global within a particular program.

### 2.1 GP process

The number of mutations to each new potential program is fixed at 10% of the number of statements in the program, with a minimum of 1; the first is a structural mutation (statement deletion, replacement, addition, or statement-specific mutation) and the remainder are changes only to the probabilistic evaluation distributions. Program depth is controlled so that only assignment statements can be introduced by mutations at depths greater than zero. This does not prevent greater depth occurring as a result of crossover.

### 2.2 Probabilistic evaluation

Each statement has associated with it two probabilistic elements. The first is a threshold value between 0 and 1 (initialised to 0.5) which controls whether or not the statement is executed. At evaluation time a value is drawn from the uniform distribution $\mathcal{U}(0,1)$; if the value drawn exceeds the threshold value then the statement is executed, otherwise it is not.

The second probabilistic element is the mean and standard deviation for a Gaussian distribution ($\mathcal{N}(\mu,\sigma^2)$) that determines in which order the statements in a given block are executed. For a given statement block, random draws are made from these ordering distributions and the statements are executed (subject to the threshold value described above) in the order of increasing draw values. For a new statement block $\mu$ is initialised to the line number of the statement within the block $(1, 2, \dots)$ and $\sigma$ to 10.

Each potential program is evaluated in this manner until either it finishes, or a "runtime" error occurs. In either case the result of the program is dictated by the value of the program's variables at the point of completion.



*GECCO '15 July 11-15, 2015, Madrid, Spain*



ACM ISBN 978-1-4503-3488-4/15/07.

DOI: http://dx.doi.org/10.1145/2739482.2764642

A mutation to the threshold is made by adding the result of a draw from the Gaussian distribution $\mathcal{N}(0.01, 0.2)$ (the mean greater than zero gives a slight tendency to switch statements off), with resulting values outside the range 0 to 1 limited to 0 and 1 as appropriate. A mutation to the evaluation order distribution, $\mathcal{N}(\mu, \sigma^2)$, is made by adding draws from the Gaussian distributions $\mathcal{N}(0, 5)$ and $\mathcal{N}(0, 1)$ respectively to $\mu$ and $\sigma$.

## 2.3 Fitness function

Since we are looking for a generalised solution, not one that is specific to a single test case, we want to test a potential solution against multiple test cases. We also want to test multiple different versions of the potential program, each generated by a different set of draws from the probability distributions associated with each statement. If the potential program contains the necessary statements, the more versions we test the more likely we are to come upon a version that is able to solve the problem at hand. We combine these two requirements by generating $T$ test cases and then evaluating each test case using a different version of the potential program. The test cases are the same for all potential programs in a single run of the GP.

The fitness is dependent on three criteria: the number of test cases correctly processed (to be maximised), the number of runtime errors and the number of program statements (both to be minimised). We might also wish to reward partial solutions.

Replacing members of the GP population with new programs that are no worse than them, but which have the probabilistic evaluation parameters mutated, seems to lead to better convergence.

# 3. RESULTS

The GP has so far been tested on two problems: to *swap* the contents of the first two elements in an array (which may be achieved using only assigment statements) and to find the *max*imum value in an array (which requires the use of a loop and an if statement). Examples of the programs generated by the GP are shown in figure 2.

For the *swap* program the GP was run 100 times, each for 100,000 iterations, for each possible combination of three run criteria:

1. using partial correctness as a fitness criterion (the number of instances where one or other element in the array was correct, but not both)
2. using statement execution threshold values
3. using statement ordering distributions

The selection of criteria are described below as a Y/N triple; thus YNY means that the GP was run with partial correctness and probabilistic ordering, but no use of the thresholds to decide whether or not individual statements are to be executed. Table 1 shows a summary of results for each of the 100 test runs for each of the 8 possible criteria combinations.

With no partial correctness and no probabilistic evaluation, only 3% of the test runs result in programs that are able to perform the swap in all test cases; in 88% of cases no correct solutions were found at all. Including any one of the run criteria (cases NNY, NYN and YNN) dramatically reduce the latter.

Figure 1 shows statement ordering probability distributions for one successful and one unsuccessful *swap* program.

| test criteria | % full solutions | % zero solutions | % errors | average stmts |
|---|---|---|---|---|
| NNN | 3 | 88 | 2 | 33 |
| NNY | 87 | 4 | 50 | 63 |
| NYN | 70 | 18 | 14 | 24 |
| NYY | 87 | 6 | 13 | 18 |
| YNN | 16 | 2 | 6 | 26 |
| YNY | 48 | 2 | 20 | 44 |
| YYN | 51 | 1 | 10 | 20 |
| YYY | 63 | 2 | 8 | 25 |

Table 1 Comparison of results for *swap* for 100 test runs for each of the 8 possible criteria combinations. For those test runs that result in correct programs, column 4 is the proportion of test cases where the program ends with an error, and column 5 is the average number of program statements in each program.

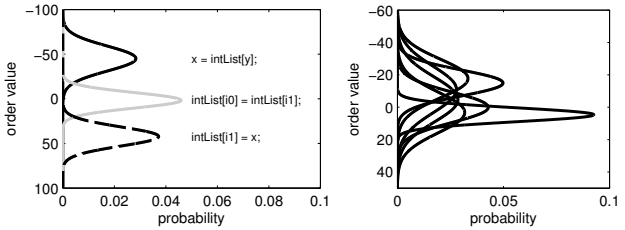


Figure 1 *Swap* programs: the statement order probability distributions for (left) a successful solution and (right) an unsuccessful solution. The solution has converged on a set of distinct distributions, while the non-solution has not.

The distributions are distinct in the successful case, but it is clear that there is no meaningful separation in the unsuccessful case and different versions of the program will result in different orderings of the statements.

# 4. CONCLUSIONS

The threshold that dictates the probability of a particular statement being executed might usefully be replaced with a Beta distribution, which would indicate some level of certainty in the decision.

# 5. REFERENCES


[1] T. Castle. *Evolving high-level imperative program trees with genetic programming*. PhD thesis, University of Kent, 2012.

[2] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.


```
x = intList[i1];
intList[i1] = intList[z];
intList[w] = x;
```

```
for (z = w * i2;i3 >= w - z;z += i2 / i2)
{ while (intList[z] > x)
   { x = intList[z]; } }
```

Figure 2 Example correct (top) *swap* and (bottom) *max* programs generated by the GP.