Predicate Expression Cost Functions to Guide Evolutionary Search for Test Data

Leonardo Bottaci

Hull University, Hull, HU6 7RX, UK
l.bottaci@dcs.hull.ac.uk

Abstract. Several researchers are using evolutionary search methods to search for test data with which to test a program. The fitness or cost function depends on the test goal but almost invariably an important component of the cost function is an estimate of the cost of satisfying a predicate expression as might occur in branches, exception conditions, etc. This paper reviews the commonly used cost functions and points out some deficiencies. Alternative cost functions are proposed to overcome these deficiencies. The evidence from an experiment is that they are more reliable.

1 Introduction

Several researchers are using evolutionary search methods to search for test data with which to test a program. The fitness or cost function depends on the test goal but almost invariably an important component of the cost function is an estimate of the cost of satisfying a predicate expression as might occur in a branch condition, an exception condition, etc.

As an example of the most basic instrumentation, consider a search for test data that will execute a given sequence of branches in a program. A record may be kept of the values of all branch predicate expressions executed. A cost for the given input is computed by counting the number of branches that have not been satisfied. This is the method used by Pargas *et al.* [5].

Although a count of undesired branch decisions provides some guidance to the search; all the test cases that fail to satisfy the same branch are given the same cost. At this point, the cost surface over the search space has become flat and the search becomes random. As an example, consider the program fragment below.

```
...
if (a <= b)
   ...        // EXECUTION REQUIRED TO ENTER THIS BRANCH
```

Suppose a test case is required to cause execution of the true branch of the conditional shown above. If the required branch is difficult to enter, many test cases will cause `a <= b` to be false. To discriminate between these tests, the program is instrumented to calculate a cost measure that penalises those tests

E. Cantú-Paz et al. (Eds.): GECCO 2003, LNCS 2724, pp. 2455–2464, 2003.

that may be considered to be "far from" satisfying `a <= b`. As an example of a possible cost function for the condition `a <= b`, the value of $a - b$ increases as `a` becomes larger than `b` and a zero or negative cost indicates that a solution has been found. Through instrumentation, the subject program has in effect been converted into another program that computes a function that is to be minimised to zero. This approach has been used by Korel [2], Tracey *et al.* [7], [8], Wegener *et al.* [9], Jones [1] and Michael [4] [3].

Clearly, the effectiveness of the search depends on the reliability of the cost functions used for the relational and logical expressions. This paper reviews the commonly used cost functions and points out some cases where they are deficient. Alternative cost functions are proposed to overcome these deficiencies. A small experiment provides some evidence that the alternative cost functions are more reliable, and more so for relatively simple programs.

2 Cost Functions for Relational Predicates

Table 1 shows the commonly used cost functions for the relational predicates. a, b are numbers and ϵ is a positive number.

Since, logically equivalent expressions express the same condition, ideally they should have equal costs. In purely integer domains, $a + 1 \leq b \Leftrightarrow a < b$. Since $cost(a + 1 \leq b) = a + 1 - b$ and $cost(a < b) = a + \epsilon - b$, this entails $\epsilon = 1$. In real domains, ϵ is the smallest positive real.

It should be emphasised that $a - b$ as the cost of satisfying $a \leq b$ is at best a heuristic. Usually, the input test case determines the values of the operands a, b only indirectly and any intervening statements have the potential to produce a cost surface as a function of the inputs that is far more complex than $a - b$. Nonetheless, many arithmetic operations do not destroy the reliability of the heuristic and it remains effective where inputs are modified in the manner illustrated in the example below

```
a := a * a - k;
if (a <= 0)
```

It is, of course, possible for the heuristic to be deceptive. Consider the program fragment

```
a := (a * a + 1) mod 65;
if (a <= 0)
```

Table 1. Relational predicate cost table using conventional cost functions

Predicate expression	Cost of not satisfying predicate expression
$a \leq b$	$a - b$
$a < b$	$a - b + \epsilon$
$a = b$	$abs(a - b)$
$a \neq b$	$\epsilon - abs(a - b)$

In this fragment, the cost of `a <= 0` decreases as `a` (input) approaches 0. At $a = 0$ the cost function attains a local minimum value of 1 but the solution is $a = 8$. It is also possible for the heuristic to be completely uninformative as in

```
a := random(a);
if (a <= 0)
```

3 Cost Functions for Logical Operators

Cost functions may be defined for the logical operators *not*, *or* and *and* in order to define a cost for compound predicate expressions such as `a <= b and not(a > 0)`. Consider first the logical negation operator.

Some researchers avoid an explicit cost function for logical negation and instead rewrite expressions that contain negation into a form that is negation free. So for example, `not (a <= b)` is rewritten as `a > b`. Introducing a cost function for logical negation avoids the need to rewrite expressions; indeed by introducing a cost function for each relational and logical predicate, cost functions may be isomorphic to predicate expressions which allows cost functions to be constructed in a simple syntax directed manner. This is an important consideration when building tools.

The cost function for negation can be derived from the requirement that $cost(a \leq b) = cost(\neg(a > b))$, i.e. $a - b$ should equal $-(b - a + \epsilon) + \epsilon$ and hence $cost(\neg a) = -cost(a) + \epsilon$.

The use of logical negation also introduces the need for a cost to be assigned to a true predicate expression. In particular, the logical constant, $true$ may be given a cost of $-maxcost$ and the logical constant, $false$ has a cost of $maxcost$. The use of these large absolute values reflects the fact that the logical constant $false$ can never be satisfied, and $true$ can never be falsified. In this scheme, all cost functions are bounded by $-maxcost$ and $maxcost$.

Consider next the cost function for a disjunction. Clearly, when the operands have different truth values, the cost of the disjunction should be the cost of the true operand. This leaves the cases where the operands have the same truth value. In this case, a popular choice for the cost of a disjunction is the cost of the operand with the lowest cost i.e. the cost function is the min function. The common corresponding cost function for the conjunction is the max function.

The cost tables for logical negation, *or* and *and* are summarised in Table 2, where c_a is the cost of a boolean expression a. In Table 2, c_a and c_b are positive ($false$) and $c_{a'}$ and $c_{b'}$ are non-positive ($true$).

The cost functions of Table 2 have been used by a number of researchers. Tracey *et al.* [7] use essentially the same cost functions although their's are restricted to nonnegative values, they measure only the cost of not satisfying an expression and negation is removed by rewriting expressions. A notable difference, however, is the use of + rather than max for conjunction so that $cost(a \wedge b) = cost(a) + cost(b)$.

Table 2. Logical operator cost table using conventional cost functions

a	b	$\neg a$	$a \vee b$	$a \wedge b$
c_a	c_b	$-c_a + \epsilon$	$min(c_a, c_b)$	$max(c_a, c_b)$
c_a	c'_b		c'_b	c_a
c'_a	c_b	$-c'_a + \epsilon$	c'_a	c_b
c'_a	c'_b		$min(c'_a, c'_b)$	$max(c'_a, c'_b)$

4 Cost Function Reliability

4.1 Analytical Considerations

The functions *min* and *max* have become popular for various forms of logical reasoning under uncertainty [6] since Zadeh proposed them for fuzzy logic [11]. In fuzzy logic, *true* is represented by 1 and $false$ by 0 and intermediate values lie in between. The truth value of a fuzzy disjunction is the maximum of the truth values of the operands, the value of a fuzzy conjunction is the minimum of the operand truth values. The common use of the functions *min* and *max*[1] should not obscure the fact that the intended interpretation in test data search is quite different.

Fuzzy logic is concerned with formalising reasoning with vague concepts where the vagueness is formalised as a fuzzy set. The justification for the use of *min* and *max* in fuzzy logic comes from the corresponding union and intersection operations for fuzzy sets. The costs associated with logical expressions to guide search are not intended to measure vagueness.

Given that the costs of predicate expressions are intended to estimate search effort, some properties which could be considered essential are:

1. The cost of a disjunction should be no more than the cost of either disjunct, i.e. $cost(a) \geq cost(a \vee b)$ and $cost(b) \geq cost(a \vee b)$.
2. The cost of a conjunction should be no less than the cost of either conjunct, i.e. $cost(a) \leq cost(a \wedge b)$ and $cost(b) \leq cost(a \wedge b)$.
3. The cost of logically equivalent expressions should be equal.

Property 1 can be justified on the grounds that adding an alternative means of satisfying a condition cannot make that condition more difficult to satisfy and so cannot increase the cost. The *min* function satisfies property 1 but makes the assumption that the cost of a disjunction is the maximum cost consistent with property 1. The argument for property 2 is analogous to that for the disjunction. The *max* function satisfies property 2 but makes the assumption that the cost of a conjunction is the lowest cost consistent with property 2.

Property 3 requires the cost functions to be consistent with the associative, commutative and distributive laws. Examples of other laws include: $cost(a)$

[1] In fuzzy logic truth increases with numerical value so *min* corresponds to *max* and vice versa.

should equal $cost(a \vee a)$ and $cost(a \vee b)$ should equal $cost(\neg(\neg\ a \wedge \neg\ b))$. The cost functions of Table 2 satisfy all three of these properties.

Although all three properties would appear necessary, it might be advantageous to trade some violation of the third property for a more reliable or informative function. Recall the use by Tracey [7] of + instead of *max* as the cost of a conjunction. To use + instead of *max* for the cost of a conjunction, while retaining *min* for the cost of a disjunction is to give up the property that logically equivalent expressions have the same cost. In particular, the distributive law of disjunction over conjunction is not satisfied because $cost(a \vee (b \wedge c)) = min(a, b+c)$[2] but $cost((a \vee b) \wedge (a \vee c)) = min(a, b) + min(a, c)$. De Morgan's law is not satisfied either.

A possible consequence of this is illustrated in the code fragment below where two logically equivalent expressions appear in distinct subgoals. Assume that the test goal is to find a test that will execute either of the statements `z := 0;`, i.e. to satisfy either `(x < 4 or y < 4)` or `not (x >= 4 and y >= 4)`.

```
if (x + y >= 16)
   if (x < 4 or y < 4)
      z := 0;
else
   if (not (x >= 4 and y >= 4))
      z := 0;
```

To exaggerate the inconsistency and also for the sake of clarity, variables are taken to be integer so that $\epsilon = 1$. When $\mathtt{x} = 8$, $\mathtt{y} = 8$, the cost is $min(5, 5) = 5$ but when $\mathtt{x} = 7$, $\mathtt{y} = 8$, which is a better test, the cost is higher at $-(-3+-4)+1 = 8$.

The possible bias this might introduce in examples such as that shown above might well be compensated for by a more general reliability advantage of + over *max*. + rewards a decrease in the costs of both conjuncts more than it rewards a corresponding decrease in the cost of just one. For example, consider the problem of satisfying the condition `x = 0 and y = 0`. A move by a search algorithm from the (x, y) point $(4, 6)$ to the point $(3, 5)$ is rewarded by + in a cost decrease of 2 but *max* produces only a cost decrease of 1. Similarly, a detrimental move from $(4, 6)$ to $(5, 7)$ is penalised more heavily by + than by *max*. The function + would seem to be more discriminating.

Continuing in this direction, the use of *min* as the cost function for disjunction may be reconsidered in the hope of finding some different function, analogous to +, that is more discriminating than *min*. Consider, for example, the following program fragment.

```
x := 1;
while (x <= 0 or y = 5)   // EXECUTION REQUIRED TO ENTER LOOP
```

[2] Note that the notation of c_a as the numeric cost of the boolean expression a is being dropped from here on. A symbol a may denote either a boolean expression or its numeric cost, as determined by the context.

Table 3. As `y` approaches 5, the cost decreases towards 0

`y`	9	8	7	6	5
`x <= 0`	1	1	1	1	1
`y = 5`	4	3	2	1	0
`x <= 0 or y = 5`	0.8	0.75	0.67	0.5	0

Table 4. Proposed logical *or* and logical *and* cost table

a	b	$a \vee b$	$a \wedge b$
a	b	$\frac{ab}{a+b}$	$a+b$
a	b'	b'	a
a'	b	a'	b
a'	b'	$a'+b'$	$\frac{a'b'}{a'+b'}$

When searching for values for the variables `x` and `y` to enter the `while` loop, the value of `x` is 1 when the conditional is first evaluated and so the cost of `x <= 0` is 1 and this will in fact be the cost of the predicate expression for all input values unless `y` = 5. A flat surface in the cost function provides no guidance to the search.

One way to interpret this problem is to consider that the cost of 1 for `x <= 0` should, initially at least, be much higher given the impossibility of changing the value of `x` from this value until entry into the loop. Determining such facts about the values of arbitrary variables in a program is of course just as hard a problem as that of finding test data.

Since the *min* function ignores improvement in the cost of the more costly operand, a more discriminating function might be constructed that takes account of a cost improvement in either operand. Such an alternative is the ratio of the product of the costs to the sum of the costs, i.e.

$$cost(a \vee b) = \frac{ab}{a+b} \ .$$

The table of costs below shows how this cost function solves the problem in the previous example.

The proposed cost functions for *or* and *and* are shown in Table 4. Note that when $a' = b' = 0$ then $cost(a \wedge b) = 0$. The above cost functions satisfy properties 1 and 2 but do not satisfy the property of equal costs for logically equivalent expressions. The error in satisfying De'Morgan's law, i.e. $cost(a \vee b) \neq cost(\neg(\neg a \wedge \neg b))$ is due to the presence of ϵ, the positive offset from zero for the costs of all false predicates that is absent from the costs of true expressions.

This anomaly can be removed by modifying the relational predicate cost functions to produce values symmetrical about zero in the range $[-maxcost, -\epsilon] \cup [\epsilon, maxcost]$ (as shown in Table 5) and to define the cost of logical negation as $cost(\neg p) = -cost(p)$.

Other inconsistencies remain, however. For example, $cost(a) \neq cost(a \vee a)$. The difference between $cost(a)$ and $cost(a \vee a \vee a)$ is greater still with the dif-

Table 5. Relational predicates with costs symmetrical about zero

Predicate expression	Cost of predicate expression	
$a \leq b$	$a - b$,	$a > b$
	$a - b - \epsilon$,	$a \leq b$
$a < b$	$a - b + \epsilon$,	$a \geq b$
	$a - b$,	$a < b$
$a = b$	$abs(a - b)$,	$a \neq b$
	$-\epsilon$,	$a = b$

Table 6. A flat cost function when `x` = 0.0

`i`	6.0	7.0	8.0	9.0	10.0
`i <= 9.0`	$-3.0 - \epsilon$	$-2.0 - \epsilon$	$-1.0 - \epsilon$	$-\epsilon$	$1.0 + \epsilon$
`x = 0.0`	$-\epsilon$	$-\epsilon$	$-\epsilon$	$-\epsilon$	$-\epsilon$
`i <= 9.0 and x = 0.0`	$-\epsilon$	$-\epsilon$	$-\epsilon$	$-\epsilon$	$1.0 + \epsilon$
`not(i <= 9.0 and x = 0.0)`	ϵ	ϵ	ϵ	ϵ	$-1.0 - \epsilon$

ference bounded by $cost(a)$. The difference between $cost(a)$ and $cost(a \wedge a)$ is $cost(a)$ and is unbounded as the number of conjunctions of a increases. In practice, such expressions are likely to be relatively rare because programmers tend to avoid writing expressions that are clearly inefficient.

One of the problems with the cost function $\frac{ab}{a+b}$ is that when one operand is very small, changes in the other are not significant. When the cost of $b = \epsilon$ then $\frac{ab}{a+b} = \frac{a\epsilon}{a+\epsilon}$ which because of rounding error can only safely be taken to be ϵ. As an example of this problem, consider the condition `not((i <= 9.0) and (x = 0.0))` where `x` and `i` are real and so ϵ is the smallest positive real. Table 6 shows the cost calculations for different values of `i` when $x = 0.0$. The cost remains the same at ϵ for all values of `i` up to 9 and so provides no guidance for the search.

To assign a cost of ϵ (the least positive value is the relevant number domain and the lowest possible cost for a false predicate expression) to a single relational expression such as $a < b$ leaves no room to give a lower cost to disjunctive expressions that include $a < b$ as one of the disjuncts. Recall the requirement that $cost(a) \geq cost(a \vee b)$.

Cost functions for relational predicates can be modified to overcome this problem by setting the absolute minimum cost for any single relational predicate expression to be some value significantly larger than ϵ, say $R \geq 1$. The costs of relational predicates in this scheme is given in Table 7.

In Table 8 the cost calculations of the previous example are repeated but with the use of the modified relational predicate cost functions with $R = 1$. The cost of `not(i <= 9.0 and x = 0.0)` now decreases as `i` approaches 9.0.

4.2 Experiment to Compare Cost Function Reliability

Ultimately, the validity of cost functions for test data search is an empirical issue. Ideally, cost functions should be compared over a large sample of programs from

Table 7. Modified relational predicate cost functions

Predicate expression	Cost of predicate expression	
$a \leq b$	$a - b + R - \epsilon$,	$a > b$
	$a - b - R$,	$a \leq b$
$a < b$	$a - b + R$,	$a \geq b$
	$a - b - R + \epsilon$,	$a < b$
$a = b$	$abs(a - b) + R - \epsilon$,	$a \neq b$
	$-R$,	$a = b$

Table 8. A decreasing cost function

`i`	6.0	7.0	8.0	9.0	10.0
`i <= 9.0`	−4.0	−3.0	−2.0	−1.0	2.0
`x = 0.0`	−1.0	−1.0	−1.0	−1.0	−1.0
`i <= 9.0 and x = 0.0`	−0.8	−0.75	−0.67	−0.5	2.0
`not(i <= 9.0 and x = 0.0)`	0.8	0.75	0.67	0.5	−2.0

various application areas. A less time consuming experiment can be done with synthetic programs generated automatically. Consequently, a sample of programs was generated by modifying the following program.

```
x := a + b + c;
y := a + b + c;
z := a + b + c;
if (x = a or y = b or z = c)
```

The modification rules, allowed: all occurrences of `+` to be replaced by any arithmetic operator, all occurrences of `=` to be replaced by any relational predicate, all occurrences of `or` to be replaced by any binary logical operator with the possible insertion of a negation operator, in addition two binary operators `leftoperand` and `rightoperand` were allowed throughout (each returning the value of one operand) as a way of reducing the length of expressions. The modification rules, allowed variables (other than `x, y` and `z`) to be replaced with any other variable or a constant drawn from a small set of integers.

For each program generated, at random, three copies were produced, the first was instrumented with the *min* and *max* cost functions of (Table 2) (herein called the min-max functions), the second with the $\frac{ab}{a+b}$ and $a + b$ functions of (Table 4) (herein called the ratio-sum functions) and the third with constant functions so that a uniform random search is done. The relational predicate expression cost functions of Table 7 were used throughout with $R = 1$. For all programs, the test goal was to find values of `a, b` and `c`, each from the domain $[-4999, 5000]$ to cause entry into the conditional.

The genetic algorithm used for the search was of the steady-state variety and similar to Genitor [10]. Test inputs were coded not as binary strings but as strings of integer. Reproduction takes place between two individuals who produce one or two offspring (depending on the choice of reproduction operator). These

Table 9. Performance of cost functions

program	**min-max**		**ratio-sum**		**random**		trivial	failed
type	solns	evals	solns	evals	solns	evals		
simple	1116	504	1309	456	97	436	0	0
complex	1853	308	2059	313	789	301	16992	6237

offspring are then immediately inserted into the population (of size 50) expelling the one or two least fit. The population is kept sorted according to cost and the probability of selection for reproduction is based on rank in this ordering.

4.3 Results

For each set of cost functions, Table 9 shows the number of trials (solns column) in which a solution was found. Three attempts were made at each problem and the table shows the total number of solutions found. The column headed 'evals' shows the mean number of fitness function evaluations used per solution, counting only cases where a solution was found. A number of the programs generated were not counted against any cost function because they were either too easy to solve (column headed 'trivial'), namely a solution was found in less than 10 random attempts, or too difficult (column headed 'failed'), because a solution was not found within the limit on fitness function evaluations, set at 1000.

A crude attempt was made to distinguish between simple and more complex programs according to the syntactic complexity of the arithmetic expressions. All programs of the following form (where only `or, and` may be replaced by `or, and`) were classed as simple.

```
x := a;
y := b;
z := c;
if (x = 1 or y = 1 and z = 1)
```

Simple programs all have a smooth cost surface (smoothest in the case of ratio-sum) but the solution set is small. In Table 9, the set of programs classed as complex is simply the entire sample of synthetic programs generated.

It can be seen that for simple programs, the performance of the ratio-sum cost functions is better, the data shows a 17% outperformance. For all programs, the ratio-sum cost functions outperform the min-max cost functions by about 10%. A possible explanation for this difference is that there is less opportunity for ratio-sum to take advantage of an improvement in more than one predicate expression cost at once since such moves are less likely to occur as the jaggedness of the cost surface increases.

5 Conclusion

Several researchers are using evolutionary search methods to search for test data with which to test a program. The fitness or cost function depends on the test

goal but almost invariably an important component of the cost function is an estimate of the cost of satisfying a predicate expression as might occur in a branch condition, an exception condition, etc.

It has been shown that the set of commonly used cost functions for the satisfaction of logical predicates (the min-max functions) perform poorly in certain cases. An alternative set of cost functions (the ratio-sum functions) has been proposed which overcome these specific problem cases. To determine the effectiveness of the ratio-sum functions on a wider range of problems, an experiment was done to compare cost functions for a sample of synthetic programs. It has been shown the ratio-sum cost functions modestly outperform the min-max cost functions but more so for relatively simple programs. A possible explanation for this is that the ability of ratio-sum to take advantage of an improvement in more than one predicate expression cost at once can be better exploited in simple programs.

Synthetic programs are an inexpensive way of subjecting cost functions to a relatively large sample of programs but they can at best provide only an insight into the behaviour of different cost functions. In terms of assessing the reliability of different cost functions in practice, they can be no more than a prelude to an experiment with a large sample of real programs. Work is underway to do this.

References

1. B. F. Jones, H. Sthamer, and D.E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.
2. B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.
3. G. McGraw, C. Michael, and M Schatz. Generating software test data by evolution. Technical Report RSTR-018-97-01, RST Corporation, Suite 250, 21515 Ridgetop Circle, Sterling VA 20166, 1998.
4. C. Michael, G. McGraw, M. Schatz, and C. Walton. Genetic algorithms for dynamic test data generation. Technical Report RSTR-003-97-11, RST Corporation, Suite 250, 21515 Ridgetop Circle, Sterling VA 20166, 1997.
5. R. P. Pargas, M. J. Harrold, and R. P. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9:263–282, 1999.
6. Judea Pearl. *Probabilistic reasoning in intelligent systems.* Morgan Kaufmann, 1988.
7. N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. *Software Engineering Notes*, 23(2):73–81, March 1998.
8. N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test data generation for exception conditions. *Software – Practice and Experience*, 30:61–79, 2000.
9. J Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43:841–854, 2001.
10. D. Whitley. The genitor algorithm and selective pressure: why rank based allocation of reproductive trials is best. *Proceedings of the Third International Conference GAs.*, pages 116–121, 1989.
11. L. A. Zadeh. Fuzzy logic and approximate reasoning. *Synthese*, 30:407–428, 1975.