SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2006; **16**:175–203
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr.354

Search-based software test data generation for string data using program-specific search operators‡

Mohammad Alshraideh and Leonardo Bottaci*,†

Department of Computer Science, The University of Hull, Hull HU6 7RX, U.K.

SUMMARY

This paper presents a novel approach to automatic software test data generation, where the test data is intended to cover program branches which depend on string predicates such as string equality, string ordering and regular expression matching. A search-based approach is assumed and some potential search operators and corresponding evaluation functions are assembled. Their performance is assessed empirically by using them to generate test data for a number of test programs. A novel approach of using search operators based on programming language string operators and parameterized by string literals from the program under test is introduced. These operators are also assessed empirically in generating test data for the test programs and are shown to provide a significant increase in performance. Copyright © 2006 John Wiley & Sons, Ltd.

Received 18 January 2006; Accepted 30 May 2006

KEY WORDS: software test data generation; genetic algorithms; string matching

1. INTRODUCTION

The goal of automatic test data generation in unit testing is to generate test data that can satisfy a given test coverage criterion. A common criterion for unit testing is branch coverage, i.e. the test set should execute every branch in the program unit under test. Branch coverage is also a common criterion for assessing research in automatic test data generation and is the criterion adopted for the empirical

*Correspondence to: Leonardo Bottaci, Department of Computer Science, The University of Hull, Hull HU6 7RX, U.K.
†E-mail: l.bottaci@dcs.hull.ac.uk

‡A version of this paper was originally presented at *UKTest2005: The Third U.K. Workshop on Software Testing Research*, held at the University of Sheffield, U.K., 5–6 September 2005. It is reproduced here in revised and extended form with the permission of the Workshop organizers.

investigation reported in this paper. The techniques used in achieving branch coverage require the satisfaction of predicate expressions generated from the program under test and, as such, they can be used as the basis of methods that generate test data for a variety of coverage criteria.

A number of different automatic software test data generation methods have been investigated [1]. These methods may be placed into one of two broad categories known as static methods and dynamic methods. Static methods aim to analyse the static structure of the program under test in order to compute suitable test cases. Static methods exploit control- and data-flow information and may use symbolic execution [2–4] but the program under test is not executed.

Dynamic methods aim to exploit information gained by execution of the program under test. The most basic dynamic method is random test data generation [5]. In this method, test data are generated randomly. Each test case is then executed and either retained or discarded according to whether it executes branches not executed by any other test case so far retained. Unfortunately, the likelihood that a test, generated randomly, will execute a difficult-to-reach branch is very low. As an example, consider the problem of generating an input to execute the target branch of the program fragment shown below:

```
...
if (s == "HELLO") {
   //TARGET
}
```

The probability that a randomly generated input will set the variable `s` to be equal to the string `HELLO` is likely to be very small. In general, random test data generation performs poorly and is generally considered to be ineffective at covering all branches in realistic programs [6].

Guided or heuristic search is a more effective test data generation approach. Various heuristic search methods have been used to generate test data including gradient descent, tabu search and simulated annealing [7], but a commonly used method is to employ a genetic algorithm [8,9]. A genetic algorithm, which was used in the work reported here, has the following features.

1. The genetic algorithm maintains not one but a population of many candidate solutions. A candidate solution is an attempt to solve the problem. In the context of test data generation, a candidate is a potential test case.
2. A fitness (or cost) function evaluates the utility of each candidate in the search for a solution.
3. A probabilistic selection function selects candidates from the population with the highest probability of selection accorded to the most promising candidates.
4. Search operators, also known as genetic operators, modify the selected candidates (known as parents) to create new candidates (known as offspring). Two search operators invariably associated with genetic algorithm search are crossover and mutation. The purpose of the crossover operator is to combine elements from two different but promising candidates in the hope of producing offspring that outperform their parents. Mutation is a genetic operator that modifies one or more elements of a selected candidate. The purpose of mutation is to explore the space of candidates that are similar to a given candidate.

Figure 1 shows the basic steps of a genetic algorithm. First, the population is initialized, either randomly or with user-defined candidates. The genetic algorithm then iterates through an evaluate–select–produce cycle until either a solution is found or some other stopping criterion applies.

DOI: 10.1002/stvr

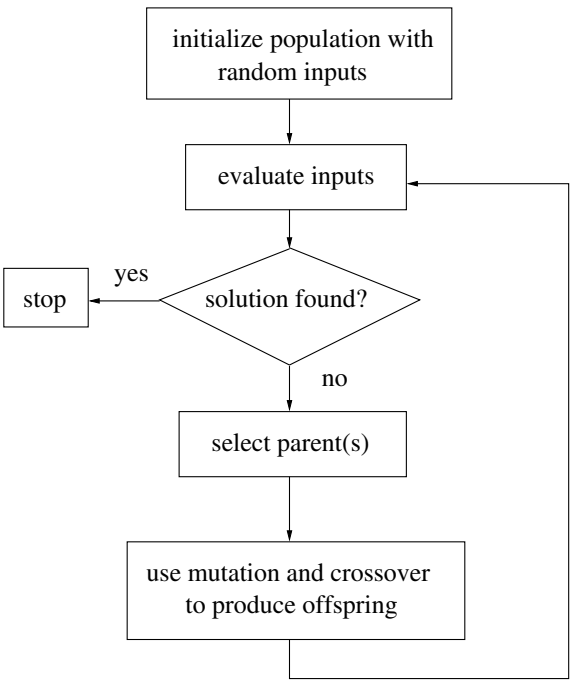

Figure 1. Flowchart of test data generation using a genetic algorithm.

The effectiveness of a genetic algorithm depends crucially on the reliability of the guidance provided by the cost function. The cost function is a metric that evaluates each candidate in terms of its 'closeness' to a solution; more specifically, it estimates the number of search operations that must be performed to transform the candidate into a solution.

Clearly the number of search operations required to transform a given candidate into a solution depends on the nature of the transformation performed by each operator. This important dependence between the cost function and the search operators can be illustrated by considering a simple search problem over the set of eight 3-bit strings from `000` to `111`. Assume that these strings constitute a set of inputs that is to be searched for the single input, `011`, which executes the target branch as shown below:

```
P(s) {
  t = F(s);
  if (t == 011) {
     //TARGET
```

For simplicity of explanation, a simple hill-climbing search process will be used. In each iteration, each candidate string `s` is submitted to the program `P` and the string `t` is compared to the required

DOI: 10.1002/stvr

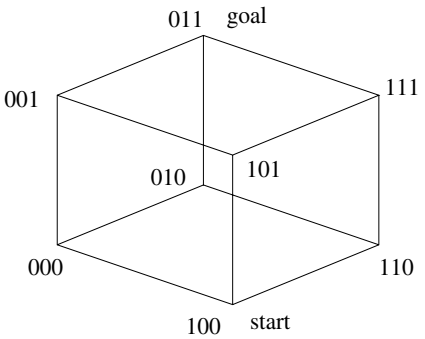

Figure 2. The search space defined by the eight 3-bit strings and a single bit-inversion search operator.

or goal string by computing a Hamming distance. If the candidate is not a solution, new candidates are generated from it, in this example by a search operator that inverts each bit to produce new candidate strings. The string (or one of the strings) with the lowest cost is used as the start for the next iteration. In this example, assume, initially at least, that `F` is the identify function and that the initial candidate is `100` from which the bit-inversion search operator generates the strings, `000`, `110` and `101`. The original string is discarded since the lowest Hamming cost is now 2. One of the new strings is selected and the iteration repeats. It is clear that this search converges rapidly to the goal string of `011`.

The space induced by the bit-inversion operator on the set of eight strings is shown in Figure 2. Since each edge represents an application of the search operator, the Hamming distance between any two strings is precisely the number of applications of the search operator required to 'search' for one string from the other. As a consequence of this, the Hamming distance is a reliable cost function for this space.

The reliability of the Hamming distance is, however, dependent on the use of the bit-inversion search operator. This can be illustrated by considering an alternative search operator that right shifts a random bit into a given string and discards the leftmost bit. The space produced by this operator is shown in Figure 3. Consider again the problem of finding the solution string `011` by applying this new search operator to the initial candidate `100`. From Figure 3, it can be seen that this can be achieved by two applications of the search operator. Notice, however, that in this case the Hamming distance (shown against each string) does not correspond to the number of applications of the search operator required to reach the solution. Moreover, if this Hamming distance were to be used to guide the search, the search would follow a path from `100` to `010` to `001` where it would remain stuck at a local minimum. For the right bit-insertion operator, the Hamming distance is not a reliable cost function. An appropriate cost function would count the number of right bit-insertion operations (equivalently, the edges between any two strings in the space shown in Figure 3) between a given string and a solution string.

The above example illustrates a general principle concerning the duality between search operators and cost functions. A realistic search problem is complicated by factors omitted from the example. The search operators are applied to values in the input domain but the cost function is applied to values

DOI: 10.1002/stvr

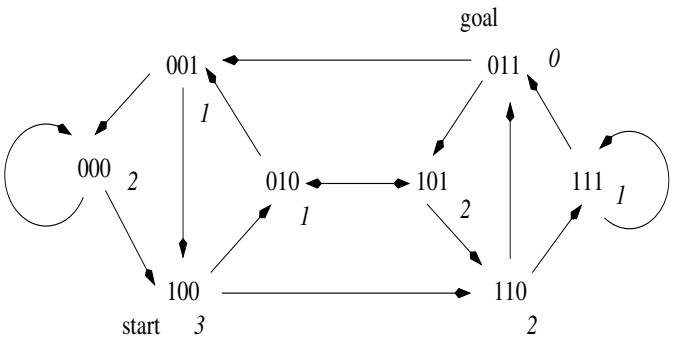

Figure 3. The search space defined by the set of eight 3-bit strings and a right bit-insertion operator.

that are arguments to the target branch predicate expression. The example avoids this issue because `F` is assumed to be the identity mapping but, in general, it is not. In the general case, the cost function may lose some of its reliability but often it is still sufficiently reliable to guide the search to a solution.

Current test data generation work [1,10–12] has been limited largely to programs in which predicates compare numbers, as illustrated in the example below:

```
if (y == 30) {
     //TARGET
}
```

For such predicates, a typical cost function is the absolute difference between `y` and 30. This cost function is appropriate for search operators that make adjustments to a single numerical value.

Unlike numeric predicates, string equality does not suggest a single 'obvious' cost function. In research that has considered string predicates, one approach has compared strings by comparing their underlying bit string representations using the bit Hamming distance as a cost function [13]. Another approach [14] reduces the problem of string search to a sequence of character searches. In this approach, only a character matching cost function is used. There are a number of string matching algorithms, used in information retrieval and biological applications, which may be useful for defining cost functions but, as yet, none of these have been applied to the problem of searching for string test data.

This paper consists of two main parts. In the first part, some potential string cost functions and corresponding search operators are considered including two new cost functions. These cost functions are assessed by comparing their performance on a number of sample test programs. In the second part of the paper, a new type of search operator is introduced with the aim of biasing the search towards strings that occur as literals in the program under test. The performance of the cost functions when used with the new operators are assessed by again generating test data for the sample programs. The results show that the new search operators provide a substantial improvement in efficiency. The main contribution of this paper is that it provides a new method for searching for string test data and demonstrates that it is potentially quite efficient.

DOI: 10.1002/stvr

2. COST FUNCTIONS AND SEARCH OPERATORS FOR STRING PREDICATES

This section considers the extent and nature of the string search space that is relevant to programs in practice. It then considers some existing cost functions and some new cost functions for the string predicates of equality, ordering and regular expression matching.

2.1. String search space

Modern software uses 16-bit character strings. The space of strings formed from the 16-bit character set is huge, so much so that a search process may be prohibitively slow to be of practical benefit. A preliminary investigation was thus done to try to establish the size and structure of the space of strings that are used in practice. A large body of software, the .NET Framework Shared Source Common Language Infrastructure (SSCLI) [15], was scanned to extract string literals. The strings were, in turn, scanned in order to produce a frequency distribution for the occurrence of each character in a SSCLI string literal. In over 13 million characters, only 850 were outside the 8-bit range. Over 99% of the characters were within the range of the 93 'printable' characters from the space character to the tilde. In practice, this means that the vast majority of the 16-bit character set need never be explored when searching for test data for typical programs.

In order to exploit the marked non-uniformity of the distribution of characters typically used in string processing programs, in the work reported here characters were restricted to have an ordinal value between 0 and 127, i.e. within the lower seven bits. Characters outside this range were excluded entirely because the vast majority of programs do not require them. Since non-printing characters occur in the strings of only a very small proportion of programs, it is probably better to deal with these as a special case. This might be done by the tester, with a knowledge of the program under test, setting the parameters of the search space to a specific set of characters.

It was also observed that many of the string literals consisted of English or English-like text as might be used for identifiers, the names given to products, organizations, etc. Given a relative frequency table for the occurrence of character pairs in English text, a random English-like string may be constructed. Initially, the first character is selected randomly from the English alphabet. The selection of subsequent characters, however, is biased so that consecutive pairs of characters in the string occur with the same relative frequency as they occur in English language texts. This approach can be applied to natural languages other than English of course, providing the language can be identified. Again, this is probably most easily done by the tester.

The search space also depends on the minimum and maximum length of the input string that is generated. It is clearly inefficient to set a maximum length that is greater than is required for a particular program. It is thus expected that the tester, with knowledge of the program under test, will specify both the minimum and maximum length of the input string.

A random string from the set of 'practical' strings may therefore be constructed by selecting a random string length and then selecting, with some specified probability, from strings generated from characters selected from a uniform distribution over the 7-bit character space or from English-like strings.

DOI: 10.1002/stvr

2.1.1. *Mutation operators*

There are three basic kinds of string mutation operator: deletion, insertion and substitution. A single deletion operator was defined to delete a random character from a given string. Two insertion operators were defined. The uniform insertion operator inserts a character, selected randomly from the range 0 to 127, into a given string at an insertion position selected randomly. Over time, random insertions within an English-like string will reduce its English-like property. To counter this, an English-like insertion operator was defined. This operator inserts a letter from the English alphabet selected probabilistically according to letters that precede and follow the insertion point. If the insertion point is not preceded or followed by a letter from the alphabet, a random letter is inserted according to the frequency of single English letters.

Two character substitution operators were defined. To accompany the binary Hamming cost function (see below), a binary character substitution operator was defined. This operator inverts a random bit within the seven low-order bits of a character. To accompany the cost functions that compare the ordinal values of characters, a character substitution operator was defined to replace a character with another of a similar ordinal value. The new character is selected from a Gaussian distribution with mean at the given character. If the mean is selected then the new character is chosen randomly from the characters adjacent to the mean. The standard deviation of the distribution was set, rather arbitrarily, at 16.

2.2. String equality cost functions

Of the many string matching metrics in the literature [16], many use a vector space approach in which each string is equated with a point, in a Euclidean space, say, and the Euclidean distance between the two points is used to derive a match cost. Another category of string matching metrics produces a distance value in terms of the number of primitive operations, typically insertion, deletion and substitution of single characters, required to transform one string into another. Functions differ in terms of the particular costs attached to the particular operations. In some applications, for example, it is more important to match digits than letters or leftmost character matches may be more important than matches in other positions. In a spelling correction application, for example, the cost of substituting one character for another may depend on the proximity of the two characters on the keyboard. The approach of considering the number of operations required to transform one string into another seems to be the most promising for defining a cost function that estimates the number of search operations required to transform a string to a solution. It is only this second type of metric that is considered in this paper.

2.2.1. *Binary Hamming distance*

Traditionally, many genetic algorithms have used a binary string representation for candidates. The mutation operator has been bit-inversion and the binary Hamming distance (*HD*) has been the cost function. This representation could be used for character strings by simply working with the underlying bit representation of each character. Once each string is converted to a bit string by concatenating the bit patterns of each character, the Hamming distance of two equal length strings may be easily computed.

Since character strings vary in length, a Hamming distance must be defined for unequal length strings. The comparison of unequal length strings requires consideration of the cost of inserting

DOI: 10.1002/stvr

or deleting characters. Strictly, a binary representation should allow only single bits to be inserted and deleted, which is clearly inappropriate, but a character insertion can be considered to be seven consecutive bit-insertion operations. On the assumption that each bit operation should have an equal cost, the cost of inserting additional bits or removing excess bits was thus taken to be equal to the cost of mismatched bits. The Hamming distance function was therefore extended so that any bits in one string that extend beyond the length of the shorter string are counted as mismatched.

Strings are left-aligned, which may lead to unrepresentative costs because of a failure to take account of deletion and insertion. An example is *HD*(XHELLO, HELLO) where only one L matches and the cost is therefore relatively high.

2.2.2. *Character distance*

Rather than use a binary space in which to map characters and strings, characters may be mapped into an ordinal space according to each character's ordinal value. In this space, a substitution operator would be sensitive to the ordinal value of the character it substitutes. This suggests a new cost function based on the pairwise comparison of character values. This new cost function, called character distance (*CD*) is defined as the sum of the absolute differences between the ordinal character values of corresponding character pairs. For strings of unequal length, any character without a corresponding character increases the cost by 128, the size of the character search space. More precisely, let string $s = s_0 s_1 \ldots s_{k-1}$ be of length k, where s_i is the ordinal value of the ith character. Similarly, let string $t = t_0 t_1 \ldots t_{l-1}$ be a string of length l, where $l \geq k$, then

$$CD(s, t) = \sum_{i=0}^{i=k-1} |s_i - t_i| + 128(l - k)$$

CD, like *HD*, is sensitive to the alignment of different length strings.

2.2.3. *Edit distance*

Of the many existing string comparison metrics used in information retrieval and biological matching, the vast majority are derived from the edit distance. The edit distance (*ED*) (or Levenshtein distance [16]) is derived explicitly from consideration of three operators that perform character insertion, character deletion and character substitution. The edit distance between two strings is the minimum number of deletions, insertions or substitutions required to transform one string into another. For example $ED(\texttt{TEST}, \texttt{TOT}) = 2$, because one deletion (or one insertion) and one substitution is sufficient to match the two strings. The edit distance function is defined by the recurrence relation below where $s : a$, $t : b$ are character strings, each consisting of a possibly empty string s, t, followed by the character a, b, respectively:

$$ED(s : a, t : b) = \min(ED(s : a, t) + 1, ED(s, t : b) + 1, ED(s, t) + ED(a, b))$$

The edit distance of two characters is one unless they are equal, in which case it is zero. The edit distance of an empty string and a given string is the length of the given string.

In considering the suitability of the edit distance as a cost function, the size of the range is important. The range of edit distance values is equal to the maximum length of the two strings compared.

DOI: 10.1002/stvr

Consider, for example, that there are over 10^{10} strings in the space of strings of length 5 and yet only six edit distance values. This means that the 'surface' of the cost function produced by evaluating each point of the search space with respect to a given goal will be largely flat.

Ordinarily, such an indiscriminate cost function would provide little guidance to the search but the cost function is nonetheless reliable given the search operators it assumes. A cost of one for the substitution of a character by any other character assumes a search operator which generates, in a single step of the search, all strings that may be formed by a single character substitution. In a practical search, the number of successors that such an operator would produce is too large to allow the easy identification of the best successor.

A more practical substitution operator would produce a single string by substituting a given character with a single character. If the new character value is defined to be adjacent to that of the character it replaces, then the difference in ordinal values estimates the number of substitution operations required to substitute one character for another. The ordinal distance is also a reasonable estimate when the new character value lies close to the original as is the case with the Gaussian character substitution operator.

To accommodate this kind of search operator, the edit distance function can be modified to take account of the difference in character values whenever a character is substituted. The edit distance of two characters can be taken to be equal to the absolute difference in their ordinal values. The ordinal edit distance (*OED*) could thus be defined as

$$OED(s : a, t : b) = \min(OED(s : a, t) + k, OED(s, t : b) + k, OED(s, t) + |a - b|)$$

where k is the insertion or deletion cost and a, b in $|a - b|$ are interpreted as ordinal values.

To choose a value for k, note that the argument for the practicality of using a character substitution operator that produces a single string applies also to the insertion operation. A practical insertion operator would therefore produce a single string from the insertion of a single character. The number of insertion operations now required to insert a given character is a function of the character set size. For this reason, the cost of insertion, k, was chosen to be 127. Given that any match that can be achieved by an insertion into one string can also be achieved by a deletion in the other, the cost of deletion was also chosen to be 127.

Using 127 as the cost of insertion and deletion, however, gives $OED(\texttt{XHELLO}, \texttt{HELLO}) = 127$ and yet $OED(\texttt{GDKKN}, \texttt{HELLO}) = 5$, which is too low since the search effort required to match `GDKKN`, `HELLO` (four substitutions) should be higher than the effort to match `XHELLO`, `HELLO` (a single deletion). The problem is that substitution costs become unreasonably low as corresponding character values approach each other. The low, non-zero substitution costs were therefore offset away from zero while retaining the maximum cost at 127. This was done by setting the substitution cost to be $127/4 + 3|a - b|/4$ when $|a - b| > 0$ and zero otherwise. This gives $OED(\texttt{GDKKN}, \texttt{HELLO}) = 162.5$, which is higher than 127. *OED* was thus defined as

$$OED(s : a, t : b) = \min(OED(s : a, t) + 127, OED(s, t : b) + 127, OED(s, t) + 127/4 + 3|a - b|/4)$$

The significant difference between *HD* and *CD* compared to *OED* is that the cost of *OED* does not depend on a leftmost alignment of strings. For example, in $OED(\texttt{XHELLO}, \texttt{HELLO})$ five characters match.

DOI: 10.1002/stvr

2.2.4. *String ordinal distance*

Zhao and Lyu [14] propose a string comparison cost function. In this function, each character string is represented by a non-negative integer. This integer is constructed by regarding each character of the string as a 'digit' in a base equal to the character set size. If 128 is the size of the character set then the numerical value of the string $s = s_0 s_1 \ldots s_{k-1}$ is $\xi(s) = 128^{k-1}s_0 + 128^{k-2}s_1 + \cdots + 128^2 s_{k-3} + 128s_{k-2} + s_{k-1}$. The string ordinal distance cost function (*OD*) is thus defined for two strings s, t as $OD(s, t) = |\xi(s) - \xi(t)|$.

In practice, $\xi(s)$ may be very large for long strings. *OD* may be too large to be represented using programming-language-provided integer data types even when two strings differ in only a single character, if it is the leftmost character. Zhao and Lyu [14], avoid integer overflow because the algorithm that searches for a matching string searches for only a single character at a time. This means that, in practice, all string comparisons are in fact character comparisons. Used in this way, the ordinal distance cost function becomes equal to the conventional cost functions used for numeric predicates.

To search for a string match one character at a time, however, is not usually efficient. Circumstances arise in which good candidate solutions are neglected because of the particular order in which character matches are pursued. Consider, as an example, the search for a string to match `HELLO`. Consider further that `GABCD` and `XELLO` are two candidates in this search but the first character only is being used to give a cost for the match. Because `H` is closer to `G` than to `X`, the string `XELLO` is discarded in favour of `GABCD`. This particular example is clearly a mistake. The general problem is that a character-by-character search imposes the additional search problem of finding an order in which character positions are to be matched. The example program fragment below requires that the second character of the string be matched before the first:

```
if (s[0] == 'H' && s[1] != 'E') {
  s[0] = 'X';
}
if (s == "HELLO") {
     //TARGET
```

If the ordinal distance cost function were to be used to compare strings rather than characters it would penalize mismatches in the leftmost characters far more than mismatches in the rightmost characters. Given that each character is equally likely to be modified by a search operator, this difference in cost values renders the cost function unreliable in many situations. The ordinal distance cost function is also very sensitive to the lengths of the strings compared. For example, the distance between `HELLO` and `XHELLO` is an order of magnitude higher than the distance between `HELLO` and `GABCD`. For these reasons, the string ordinal distance was considered unsuitable as a cost function for string equality.

2.3. String ordering

String test data may be required to satisfy string ordering predicates such as $s < t$. String ordering may be determined from the ordinal value of each character in the character set or it may be determined using language or culture-specific rules. It is only the ordering induced by the character ordering that is considered here.

DOI: 10.1002/stvr

2.3.1. *Ordinal value ordering*

For a given maximum string length, and when all strings are right null padded if required to achieve the maximum length, a set of strings may be totally ordered according to the ordinal value of the string. When two strings are out of order, the difference in ordinal values may be used as the cost, *OV*, thus

$$\begin{aligned} OV(s, t, <) &= \xi(s) - \xi(t) + 1, \quad s \geq t \\ OV(s, t, <) &= -OV(t, s, \leq), \quad s < t \\ OV(s, t, \leq) &= \xi(s) - \xi(t), \quad s > t \\ OV(s, t, \leq) &= -OV(t, s, <), \quad s \leq t \end{aligned}$$

This cost function has disadvantages, however. The problem of the very large values it produces has already been mentioned. In addition, it is not clear that the cost values produced are a reasonable estimate of the cost of satisfying a string ordering predicate. In the majority of cases, when two random strings are compared for a given ordering, it is only the most significant character of each string that is relevant. Consider, for example, the cost of satisfying the predicate expression `XYA < NKL`. This predicate may be satisfied by modifying only the first character in either string. This observation motivated the definition of the following string order cost function.

2.3.2. *Single character pair ordering*

The cost of satisfying the predicate expression `XYA < NKL` should depend on the cost of satisfying `X < N` since this is how the string predicate expression may be solved with the least modification to the strings. A predicate such as `X < N` may be satisfied by modifying either character. In general, the cost of satisfying a character predicate is calculated according to the cost functions for numerical predicates [17]. In the case of $<$, it is the difference in ordinal values plus one; in the case of $\leq$, it is the difference in ordinal values.

For some string predicate expressions, there may not be a choice of character that may be modified. For example, if the string alphabet consists only of the 26 letters from `A` to `Z` then two characters must be modified to satisfy `Z < A`. Because twice as many character modifications are required, such predicates are considered more costly to satisfy.

In general, the cost of satisfying a string predicate expression may be based on the number of character pairings where a single character modification is sufficient to satisfy the string predicate expression. In the example `XYA < NKL`, a single character modification is sufficient to satisfy the predicate expression only in the first pairing. Until the first pairing is at least equal, modifying characters of subsequent pairings is futile. In the example `NNP < NNC`, a single character modification in any three of the character pairings is sufficient to satisfy the predicate expression and consequently the cost of this example should be lower. The cost is highest when no such character pairings exist.

On the basis of the above observations, a cost function, known as single character pair (*SCP*) cost, was defined. A cost for an unsatisfied string ordering predicate expression is calculated as follows: for each character pair formed from corresponding characters in two strings, left-aligned and right null padded where necessary to be of equal length, a character pairing cost is calculated. A character pairing cost is calculated as follows: if no single character modification in that pair can satisfy the string predicate expression, the cost for that pair is 2×127. 2×127 was chosen to be significantly

DOI: 10.1002/stvr

larger than 127, which is the largest possible cost for a single character modification. If modification of a single character in a pair, a, b, may satisfy the string predicate then the cost for the pair is $a - b + 1$ assuming $a < b$ is required.

The cost of the string predicate expression is the sum of the character pair costs divided by the length of the longest string to give an average cost for the modification of a character in a character pair. When two empty strings fail a string ordering predicate, the cost is 2×127. For consistency with the cost function for logical negation [17], the cost of a satisfied string ordering predicate expression is the arithmetic negation of the cost of the logical negation of the string predicate expression.

2.4. Regular expression matching

Consider an input string that does not match a regular expression. An equality cost comparison of the input string with any string in the regular expression set will produce a positive value. The lowest such cost obtained may be taken as the cost of matching the input string to the regular expression.

Clearly, if this cost were to be computed by enumerating each string in the regular set, its computational cost would be unacceptably high. Efficient algorithms exist, however, for matching a string against a regular expression allowing up to a given number, k, of mismatches [16]. The common basis of these algorithms is the construction of a non-deterministic finite state machine containing k copies of the finite state machine that recognizes the regular expression to be matched. Each copy is associated with a specific number of mismatches. The initial state of the sub-machine associated with zero mismatches is the initial state of the approximate match machine. Until a mismatch occurs, the machine state remains within this sub-machine. Once a mismatch is detected, there is a non-deterministic transition to a state within the sub-machine associated with one mismatch. If there are no more mismatches, the state remains within this sub-machine; otherwise, it moves to the sub-machine associated with two mismatches and so on.

In terms of computing a match cost, the metrics computed by such algorithms suffer from the same problem as the edit distance, i.e. when the metric is interpreted as a cost function for search, the corresponding search operators are prolific with the result that any string is never very far, in terms of the number of operators that need be applied, from satisfying a regular expression. The maximum cost is the maximum number of mismatches, which depends on the length of the input string. Although the edit distance metric was easily modified to allow 'fine-grained' costs that depend on the particular mismatched characters, the non-deterministic finite state machines are not so readily modified without incurring the potentially high computational cost of computing the minimum cost path in a graph of transitions.

Myers and Miller [18], however, give an efficient algorithm for approximate regular expression matching in which specific mismatches may be assigned user-defined real-valued costs. The algorithm computes a cost by finding a minimum cost path through the edit graph, which is constructed from the graph of the finite state machine representing the regular expression. The edit graph shown in Figure 4 is the graph that results for an evaluation of the match of `cab` with $(a|b)a^*$. In this figure, each state is labelled with the input it accepts, where non-labelled states denote empty transitions. Each 'row' of the graph is the graph of the regular expression finite state machine. There is one row for each symbol in the text.

Various edges in the edit graph are labelled with a character alignment, i.e. a matching of a symbol in the pattern with a symbol in the input text. The symbol ε is the missing character. Alignments on edges

DOI: 10.1002/stvr

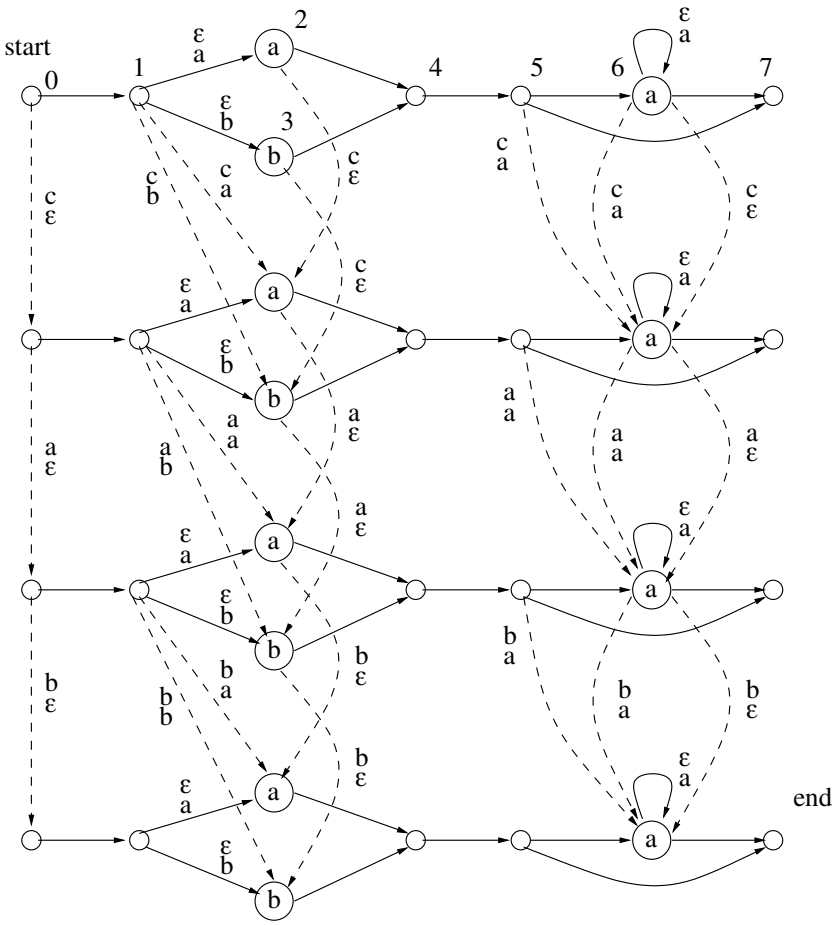

Figure 4. The regular expression edit graph for the match of the string `cab` and the regular expression $(a|b)a^*$.

within a row (solid lines) represent deletion of a symbol in the pattern. Edges between rows (dashed lines) represent ways in which a symbol in the text may be either deleted or substituted with a symbol in the pattern. At state 1 in the first row, for example, a or b may be deleted from the pattern or c may be substituted by a or b.

Each alignment may be assigned a cost depending on the characters involved. In the work reported here, the alignment of two characters was defined to have the cost as given by *OED*, i.e. $127/4 + 3|a - b|/4$ when $|a - b| > 0$ and zero otherwise. The alignment of a character with a missing character, an insertion or deletion was given a cost of 127. A description of the algorithm, given by Myers and Miller [18], is beyond the scope of this paper but it can be seen from Figure 4 that

DOI: 10.1002/stvr

Table I. Costs used to compute the distance between the string `cab` and the regular expression $(a|b)a^*$ with the edit graph shown in Figure 4. Costs are rounded for clarity.

	States							
Input	0	1	2	3	4	5	6	7
Empty string	0	0	127	127	127	127	254	127
`c`	127	127	32	31	31	31	158	31
`a`	254	254	127	158	127	127	31	31
`b`	381	381	254	254	254	254	62	62

a path from start to end aligns c with b (state 3, row 2), aligns a with a (state 6, row 3) and aligns b with a (state 6, row 4). To compute the cost of this path, the algorithm produces the cost table shown in Table I. Each entry in the table is the cost of matching the text up to the row, with the pattern up to the column. The first row of costs corresponds to the cost of matching the empty string. It can be seen, for example, that the cost of matching `c` with $(a|b) = 31$ (row 2, state 4) and `ca` with $(a|b)a^* = 31$. The cost of `cab` with $(a|b)a^*$ is in the bottom rightmost entry.

The efficiency of the algorithm is $O(S \times R)$, where S and R are the lengths of the input string and the regular expression. The key to this efficiency is that two traversals of the edit graph in which each node is visited in topological order are sufficient to compute the minimum cost path.

No competing algorithm for inexact matching of regular expressions was considered (in fact, the authors are not aware of any other approach to approximate regular expression matching that allows real-valued costs).

3. EMPIRICAL ASSESSMENT OF STRING SEARCH OPERATORS AND COST FUNCTIONS

In order to assess the reliability of the cost functions introduced in the previous section, an empirical investigation was carried out. A number of test programs were assembled and, for each program, an attempt was made to generate inputs to achieve branch coverage. These programs, which include predominantly string predicates, are described in Table II. The size of each program is given as lines of code (*LOC*). For each program, the total number of relational predicate expressions (*RPE*) in each program is listed, and the number that are string relational predicate expressions are given in parentheses; the remaining relational predicates are numeric. One or more relational expressions may be combined with logical connectives into a branch predicate. The number of branches is also listed. The programs are available from the authors on request.

3.1. Experimental parameters

Each of the cost functions and associated search operators were implemented in a prototype test data generation tool. The tool has been constructed by modifying the JScript (JavaScript) language compiler

DOI: 10.1002/stvr

Table II. The JScript functions used for empirical investigation.

Name	Description	*LOC*	*RPE* (str)	Branches
Calc	Simple prefix calculator accepting as input an operator string and two numeric operands	40	11 (11)	22
Cookie	Reads a specific name–attribute pair as from a cookie	23	6 (5)	10
DateParse	Validates name of day of the week and decodes name of month	52	19 (19)	26
FileSuffix	Given the elements of a file pathname, checks that the file suffix is consistent with directory	40	11 (10)	22
Order4	Checks if four argument strings are in a specific increasing or decreasing order	15	14 (6)	6
Pat	Checks for the presence of an argument string x, within another argument string y, the presence of the reverse of x, both x and its reverse and the two palindromes formed by concatenating x with its reverse	62	14 (10)	28
Regex	To execute one of three target branches, the argument string must be a floating point number in scientific notation, a date in dayname, daynumber, monthname format or a URL in a restricted format	20	3 (3)	6
Stem	Removes a few given suffixes from a word to leave the word stem	44	11 (8)	16
Txt	Translates a few words and phrases into an abbreviated text form	29	11 (11)	14
Title	Validates that a person's title and sex are consistent	36	21 (21)	12

within the SSCLI and can therefore be used to test functions within programs written in the JScript language. The program must include directives to specify any input domain constraints that are to be applied. The program is then parsed and semantic analysis is carried out. The tool then inserts instrumentation code at each branch in the function. This instrumentation code calculates the cost of each branch predicate whenever it is executed.

The cost of each relational predicate expression was calculated according to the cost functions given in the previous section. Where branch predicate expressions consist of two or more relational predicates joined by logical connectives, *and*, *or* and *not*, the cost values were combined according to the scheme given by Bottaci [17]. In the case of logical *and*, for example, the costs of the constituent operands are added whenever they are both false. For nested branches, the costs of the branches in the control dependency condition of the target branch were similarly combined to provide an overall cost value for the candidate input. Unexecuted branches were assigned a high fixed cost. The algorithm for the inexact matching of regular expressions was implemented by adapting the regular expression parser in the SSCLI to construct a finite state machine recognizer for a given regular expression. The algorithm uses the transitions of this recognizer to complete the table of costs.

DOI: 10.1002/stvr

3.1.1. *Input domains*

As described in Section 2.1, all character values were restricted to an ordinal range from 0 to 127 inclusive. A maximum string length of 20 was used to create a large search space and thereby reduce, to a very low probability, the possibility of achieving branch coverage by random data generation.

3.1.2. *Genetic algorithm*

The search was directed to generate data for one branch at a time. The order in which the branches of the program were targeted was arbitrary except that no nested branch was targeted before the containing branch. This is not, in general, a good strategy since it will become stuck at an infeasible branch but it is adequate for the experimental purposes of this paper given that all the branches in the sample programs are feasible.

A steady-state style genetic algorithm, similar to Genitor [19], was used in this work. The cost function values computed for each candidate input were used to rank candidates within the population in which no duplicate genotypes are allowed. A probabilistic selection function selected parent candidates from the population with a probability based on their rank, the highest ranking having the highest probability. More specifically, for a population of size n, the probability of selection is

$$\frac{2(n - rank + 1)}{n(n-1)}$$

A single tree-structured representation was used, both for candidate inputs (phenotype) and for crossover and mutation (genotype). At the top level, a candidate is an array of objects in one-to-one correspondence with the parameters of the program under test. Each object may be a primitive value, i.e. a number, character or string, or an array. This representation has the advantage that all candidates have the same structure. Candidates differ only in the lengths of strings and these occur only at the leaf nodes of the structure.

Single point crossover was used. A cut-point within the tree structure was selected randomly and the resulting parts were exchanged. If the cut-point fell within a string and beyond the length of the shorter string then the single suffix from the longer string was transferred to the shorter string as illustrated below:

```
  parents            offspring

DATA                 DATAATION
GENER|ATION          GENER
```

A genetic algorithm has a number of parameters that may be modified to suit a given problem. The size of the population and the frequency with which selected candidates are mutated are two examples. In the context of test data generation, a search algorithm must be able to perform effectively without significant human intervention as such intervention is not cost effective; hence, no parameter was 'tuned' to suit any particular program under test. In the work reported here, a population size of 100 was always used. At each evaluate–select–produce cycle, either mutation or crossover was applied with equal probability. This means that a third of selected candidates were mutated since two candidates are selected for a single application of crossover. The mutation of a candidate consisted of

DOI: 10.1002/stvr

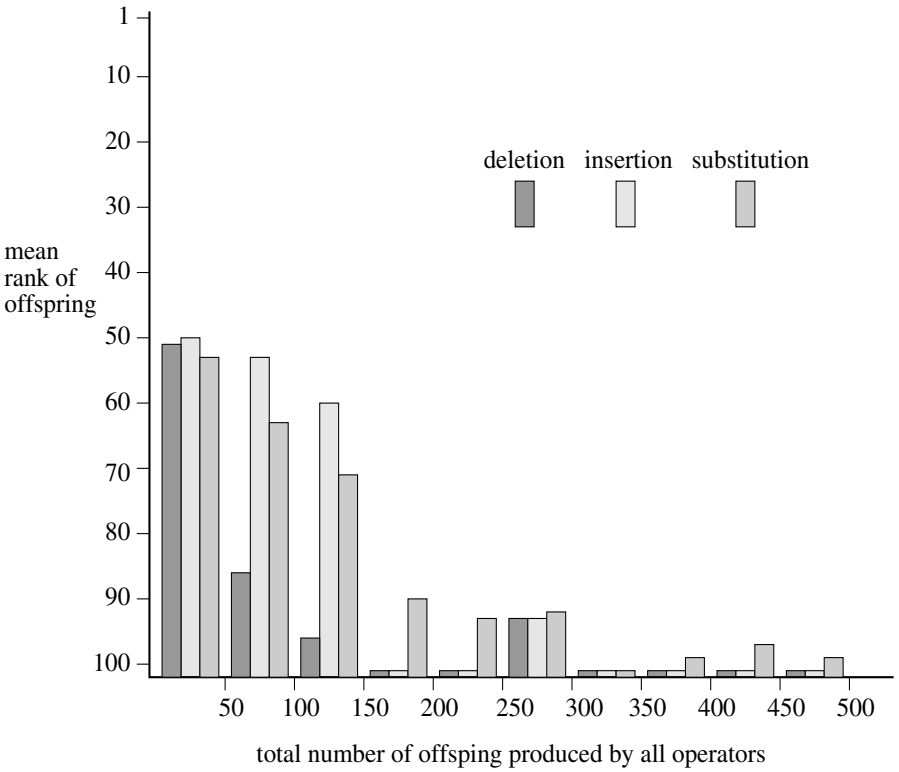

Figure 5. The mean rank of offspring produced by each kind of mutation operator during successive periods of search. The population size is 100 and a rank of 101 indicates offspring not sufficiently fit to enter the population.

a mutation to a single randomly selected primitive value, in this case a string or a number. Of the two character substitution mutations, only the binary bit-inversion operator was used when searching with the Hamming distance (*HD*) cost function; otherwise, the Gaussian substitution operator was used.

There are three kinds of string mutation that may be applied: deletion, insertion or substitution. Initially, the choice of which particular mutation to apply was determined randomly with equal probability. During a number of preliminary runs of the genetic algorithm, however, it was noticed that the effectiveness of the different kinds of mutation operator (i.e. the rank of the offspring produced) varied according to the stage in the search. The bar chart of Figure 5 shows the mean rank of the offspring produced by each kind of mutation for successive periods of the search to find data to execute a single branch in the Calc program.

The three different kinds of mutation operator are broadly equally effective in the early part of a search but in the latter stages only the substitution operator is effective and no offspring produced by the deletion or insertion operators enter the population.

DOI: 10.1002/stvr

In the latter stages of a search for a given target, all the candidate strings will usually be of the same length as the solution. Some of the characters will be correct and others will be close in value. In such a situation, inserting or deleting a character from any candidate string increases its cost beyond that of the lowest ranked candidate. Recall that the cost of an insertion or deletion is the same as the maximum substitution cost.

Clearly, it is inefficient to produce and evaluate offspring that will inevitably be discarded. To reduce this inefficiency, the frequency with which deletion and insertion operations are applied may be reduced as the search progresses. The approach of varying the rate at which different operators are applied has been used with genetic algorithms [20] and is common in evolutionary strategies [21]. Adopting this approach, two sets of results are given in the following section. In one set, the three mutation operators were always applied with equal probability. In the other set, the relative frequency of deletion and insertion was reduced in three stages during the search for a given target. For the first 300 offspring, all operators were applied with equal probability. From 300 to 500 offspring, substitution was five times more probable than deletion or insertion, from 500 to 700 offspring, substitution was ten times more probable than deletion or insertion and beyond 700, substitution was 20 times more probable than deletion or insertion. These values were chosen without detailed analysis and on the basis of inspecting the record of mean offspring rank for each of the three mutation operators for some of the branches in the sample programs. In addition, whenever, the probability of substitution was increased, the standard deviation of the Gaussian substitution operator was reduced in order to localize the search. The initial reduction was from 16 to 10, the second reduction was to 6 and finally to 3. Again, no detailed analysis was done to choose these figures; they were chosen only on the basis that they provide a progressive decrease to a small value.

3.2. Results

For each type of string predicate, results are given for the number of test program executions required to find test data to cover all branches.

3.2.1. Equality

The results shown in the bar chart of Figure 6 show the number of program executions required to find input data to achieve branch coverage, averaged over 20 trials. The probability of character insertion, deletion and substitution was equal throughout the search. These results provide some evidence that *OED* is the most efficient of the three cost functions and that *CD* is more effective than *HD*. Overall, *OED* is about a third more efficient than *CD*. The relative performance of *OED* is consistent across all the programs apart from Pat. Pat does not require its arguments to be any specific string; hence, it is attempting to find two randomly chosen strings that are equal in part. As such, Pat presents relatively weak demands on the test data.

To explain the poorer performance of *HD* and *CD*, recall that *HD* and *CD* left-align strings and then compare corresponding characters, thus giving relatively high costs in comparing strings such as `HELLO` and `XHELLO`. This problem, however, tends to occur most, early on in the search. Later in the search, for most of the branches of the programs used in the study, the candidate strings have the correct length and a number of corresponding characters match. In this situation, *CD* and *OED* give similar costs. Indeed, the more similar the compared strings, the less likely that *OED* will compute

DOI: 10.1002/stvr

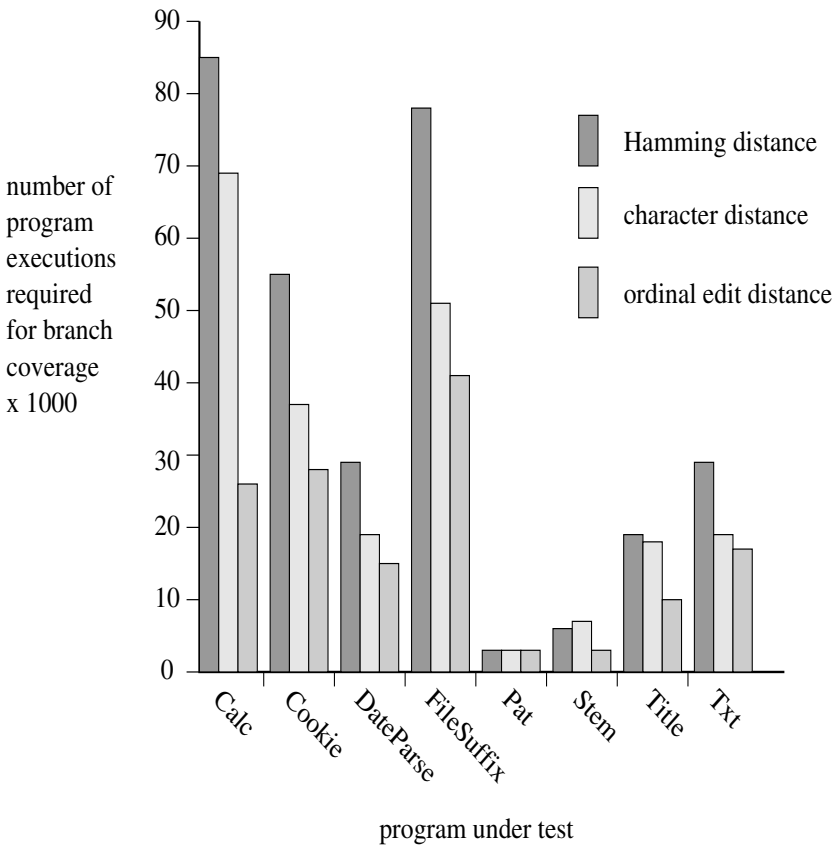

Figure 6. The number of executions of the program under test required to find test data to achieve branch coverage (averaged over 20 trials) with equal probability of character insertion, deletion and substitution.

a cost via a deletion or insertion since these operations cost significantly more than a substitution of similar characters.

It should be noted that one of the most difficult search problems, which was presented by the FileSuffix program, was not related to the performance of any cost function. A fragment of this program is shown below:

```
fileparts = file.Split(".");
lastpart = fileparts.Length - 1;
if (lastpart > 0) {       ...
  if (fileparts[lastpart] == "exe") {
      //TARGET
  }
```


DOI: 10.1002/stvr

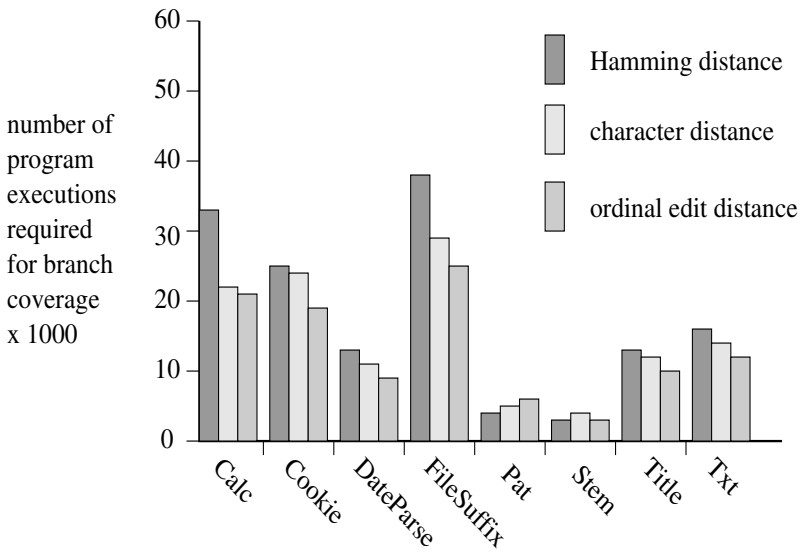

Figure 7. The number of executions of the program under test required to find test data to achieve branch coverage (averaged over 20 trials) with a progressive increase in the probability of character substitution.

In this program, an input string `file` is split into substrings at each occurrence of the dot character. A branch predicate is then satisfied if at least two substrings are produced and the last substring is equal to the string `exe`. Until a string is generated that contains a dot, only one substring is produced and `lastpart > 0` produces a constant cost of 1 and so the search receives no guidance. Once a dot was inserted, the search progressed steadily under the guidance of the cost function.

The results shown in the bar chart of Figure 7 again show the number of program executions required to find input data to achieve branch coverage, averaged over 20 trials, but in this case the probability of character substitution was progressively increased and the standard deviation of the distribution used by the Gaussian substitution operator to select the replacement character was reduced. Comparing these results with those of Figure 6, Figure 8 shows the average number of executions over all programs according to cost function and mutation probabilities. It is clear that there is a significant improvement in efficiency to be gained from increasing the probability of a substitution mutation. Note, however, that the superiority of the ordinal edit cost function declines when the probability of a substitution mutation is increased. This can be explained by noting that the ordinal edit distance will give relatively low costs to matches such as (`XHELLO`, `HELLO`) compared with matches such as (`GDKKN`, `HELLO`). Awarding (`XHELLO`, `HELLO`) a low cost is ineffective, however, if deletion or insertion is rarely applied.

3.2.2. String ordering

The empirical assessment of string order relations such as $\leq$ is not straightforward. There is a reasonable probability of satisfying such an order relation given two randomly selected strings,

DOI: 10.1002/stvr

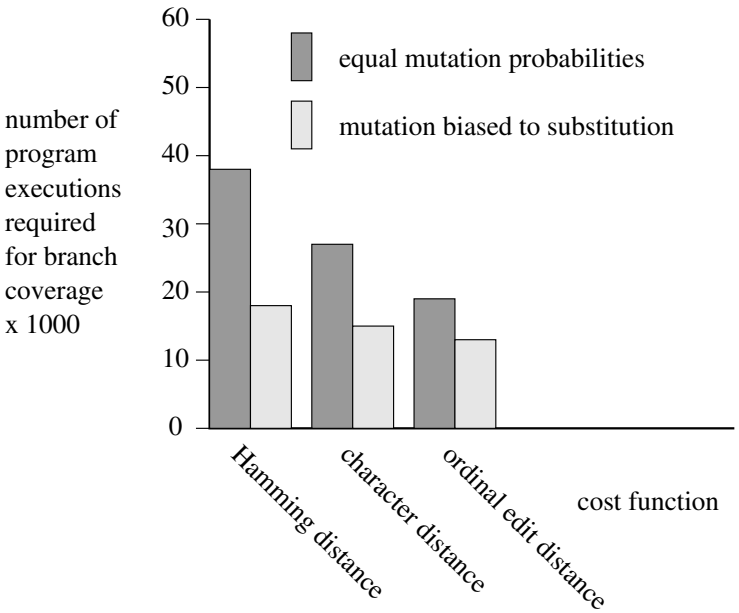

Figure 8. The average number of executions of the program under test required to find test data to achieve branch coverage for sample programs (averaged over 20 trials).

something that is most unlikely for an equality predicate. A situation in which a predicate such as $\leq$ is difficult to satisfy is when a constraint such as $s \leq t \leq u$ applies. Here, the value of t may be difficult to find if s is close to u. This problem becomes more difficult as s approaches u and, in the limiting case, the difficulty of satisfying $s \leq t \leq u$ equals the difficulty of satisfying $s = t = u$. A test program was thus required to impose a difficult-to-satisfy order relation on the program inputs but not so difficult that the order relation was, in effect, an equality relation. The program shown below was therefore written:

```
Order4(s, t, u, v) {
  if ((4 < s.Length && s.Length < 7) || (4 < t.Length && t.Length < 7)
      (4 < u.Length && u.Length < 7) || (4 < v.Length && v.Length < 7)) {
    if (s < t && t < u && u < v) {
      //TARGET
    }
    else if (s > t && t > u && u > v) {
      //TARGET
    }
  }}
```

The lengths of the strings were restricted in order to make the target branches more difficult to satisfy. Table III shows the number of executions required to find test data to execute all branches

DOI: 10.1002/stvr

Table III. The number of executions required to find test data to achieve branch coverage (averaged over 50 trials).

	No mutation bias		Mutation bias		
Program name	Ordinal value	Single character pair	Ordinal value	Single character pair	Random
Order4	1711	1622	1702	1813	23 115

Table IV. The number of executions required to find test data to achieve branch coverage (averaged over 20 trials).

	Mutation operator probability	
Program name	Equal	Bias to substitution
Regex	16 122	8023

averaged over 50 trials using each of the candidate cost functions for string ordering. The results are given with and without the bias to the substitution mutation operator. The table also shows the average number of executions required to find test data when a two-valued cost function was used, i.e. a single cost value for true and a single cost value for false. This indicates the difficulty of finding test data by random search. Fifty trials were used to distinguish more accurately the performance of the cost functions compared to random search.

There is no evidence to suggest that *SCP* is more or less efficient than *OV* in terms of performance. There is also no advantage in increasing the probability of the substitution operator. This is understandable given that the search is not aiming to generate a fixed length string. As can be seen from the number of random candidates generated before satisfying the order predicates, the order relations are not particularly difficult to satisfy in this example and this may be true of order predicates more generally. *SCP* has practical advantages though: it is easier to implement since it does not require additional work to represent large numbers that exceed the capacity of the native numerical types.

3.2.3. Regular expression matching

Table IV shows the number of executions required to find test data to satisfy the regular expressions in the program Regex. Again, increasing the probability of substitution over deletion and insertion improves the efficiency of the search.

DOI: 10.1002/stvr

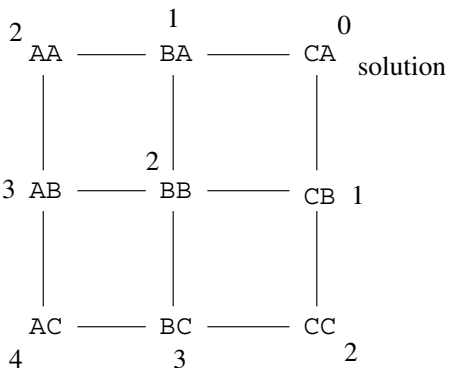

Figure 9. A small search space of nine strings with increment and decrement character mutations. The cost of each string compared to CA is shown.

4. PROGRAM-DEPENDENT SEARCH OPERATORS

Many of the programs that process strings contain string literals. The examination of the SSCLI code showed that about 65% of string predicate expressions contain a string literal. A program may match a string literal with a string input or a string derived from the input. A simple example is given below:

```
P(s) {
  t = F(s);
  if (t == "AC") {
    //TARGET
```

If the string literal, i.e. AC, were to be generated as a candidate solution and F is the identity function, the search would produce a solution immediately. In general, however, the relationship between the input string and the string comparison in a branch predicate may not be so direct. In practice, F may not be the identity function and the input may be processed by any number of statements before a string comparison is made in a branch predicate. The effect of these statements is to add a transformation to the search space.

The reliability of the cost function need not necessarily suffer as a result of such transformations. To illustrate this, assume the function F reverses its input. In this case, it is the input CA that executes the target branch, not the program literal AC. For simplicity, assume a search space of only nine strings as shown in Figure 9. The bidirectional edges of the graph indicate possible string modifications by a search operator that may only 'increment' or 'decrement' a single character in the string. Against each string is shown the cost of that string compared to AC. The string reverse operation performed by F does not reduce the reliability of the cost function since the costs decrease steadily towards the solution CA.

In this example, using the program literal string AC as a candidate solution provides the worst possible start for the search as the minimum distances from other strings are all shorter. This observation prompted consideration of applying a search operator to counter the effect of F.

DOI: 10.1002/stvr

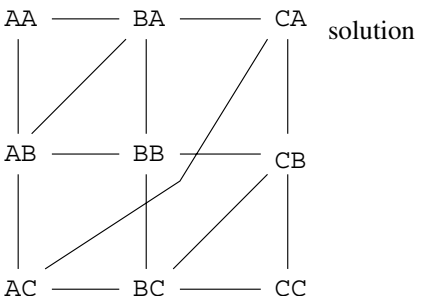

Figure 10. The search space after the addition of a reverse search operator.

Adding a string reverse search operator (reverse is its own inverse) leads to the space shown in Figure 10. The minimum number of applications of a search operator necessary to transform the initial input string AC to the solution CA is now just one. Additional search operators reduce path lengths but they do so at the expense of increasing the number of paths. In the particular case of moving from AC to CA, however, the mean number of operations is reduced. Although the addition of the reverse operator increases the number of edges, they are all shortcuts on paths from AC to CA.

It could be argued that the program P with the reverse operator F is unusual in not reducing the reliability of the cost function. The potential advantage of seeding the population with string literals and using 'inverse string operations' as additional search operators is not restricted, however, to programs for which the cost function is always reliable. The following program, which reverses only selected strings, is an example of a transformation that reduces the reliability of the cost function:

```
Q(s) {
  if (s[0] == 'A' || s[1] == 'C' || (s[0] == 'C' && s[1] == 'A')) {
    t = Reverse(s);
  }
  else {
    t = s;
  }
  if (t == "AC") {
    //TARGET
```

This can be seen from Figure 11, which shows a local minimum at strings with equal first and second characters. Nonetheless, even in this case, the addition of a reverse string search operator overcomes the local minimum and also leads to a solution.

In moving from these particular observations to a general search strategy it is necessary to accept that it is not, in general, possible to determine how a program transforms its input. Indeed, if the inverse computation problem were decidable, there would be no need to search for test data. It is possible, however, to assemble a collection of search operators that perform the inverses of typical string processing functions such as string concatenation, insertion and deletion. It is hoped that the

DOI: 10.1002/stvr

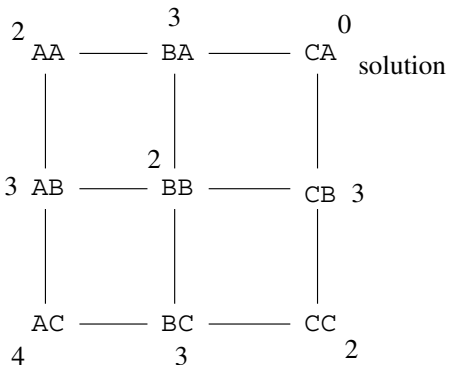

Figure 11. The program Q renders the cost function unreliable.

use of such operators, together with any string literals drawn from the program under test, should, in general, improve the efficiency of the search.

4.1. String operations biased towards program string literals

To exploit the observations of the previous section, the random string generator used to generate initial candidate solutions was extended to comprise two components. One component is the former string generator which selected strings from two distributions, a uniform distribution of strings with characters with a range of ordinal values from 0 to 127 and an English-like distribution. The second component generates strings that are either program literals or formed by concatenating these literals. The reason for concatenating literals is that programs often test if a string is a substring of another. Concatenating literals, rather than inserting a single literal into an arbitrary string, increases the chances of selecting the required literal and is also useful in the case in which the test program requires more than one literal to be a substring of a string.

The current mutation operators will, over time, decrease the proportion of candidates in the population that contain a program string literal. Consequently, three additional mutation operators were defined. One operator deletes a program literal from a given string if such a literal exists. An operator inserts a program literal or the concatenation of two literals into the string it is mutating. Another operator replaces a random substring of the string it is mutating with a program literal or the concatenation of two literals. The length of the substring replaced is equal to that of the string to be inserted so that overall there is no change in length.

The reason for replacing an equal number of characters follows from a characteristic of the search that was discussed earlier, i.e. the convergence of the search to a population of strings with the same length. If a mutation operator modifies the length of a candidate string then the cost function is likely to penalize it to the extent that it does not enter the population.

DOI: 10.1002/stvr

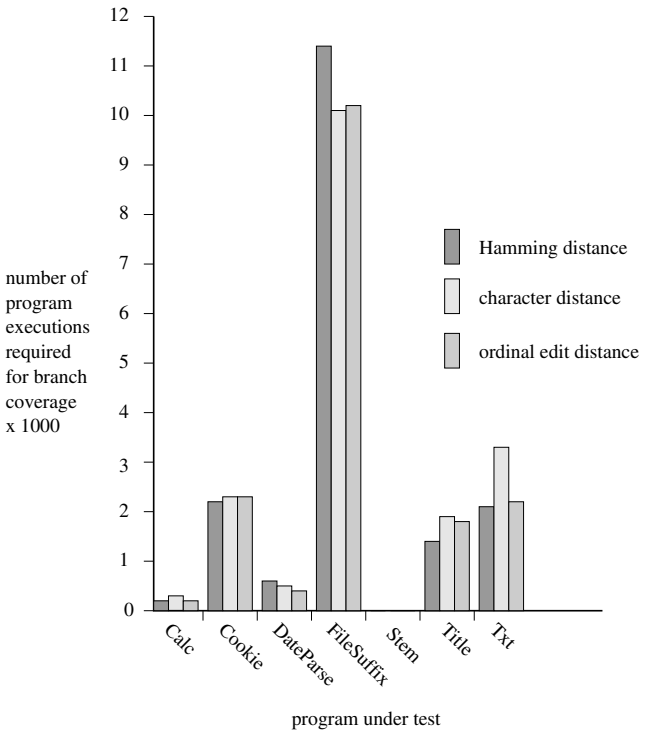

Figure 12. The number of executions of the program under test required to find test data to achieve branch coverage (averaged over 20 trials) using program-specific search operators.

4.2. Empirical assessment of program-specific search operators

The programs listed in Table II, except Order4 and Pat which contain no string literals, were used to assess the performance of the program-specific search operators. The test tool collects program literals during a traversal of the program abstract syntax tree. Character literals are also collected and treated as strings. Any string that is parsed as a regular expression is also used as a source of string literals, i.e. the text strings and characters that remain after discarding the pattern meta-characters. The random string generator was set to generate each type of string, i.e. 7-bit character, English-like and literal, with equal probability.

Again, the aim was to find input data to execute all the branches in each of the programs. For each program and cost function, 20 trials were done. The average number of program executions required to achieve branch coverage over 20 trials is shown in the bar chart of Figure 12. These results show a significant improvement in performance compared to the results of the previous section as can be seen in Figure 13, which compares the results of the ordinal edit function with respect to the use of program-dependent search operators.

DOI: 10.1002/stvr

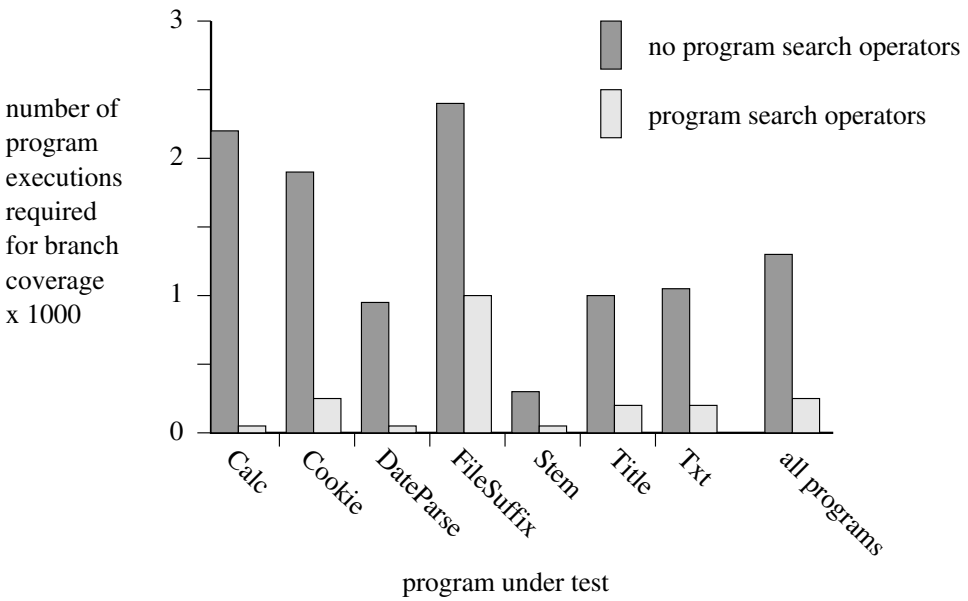

Figure 13. A comparison of the number of executions of the program under test required to find test data to achieve branch coverage (averaged over 20 trials) with and without program-specific search operators. Only the values obtained with the ordinal edit distance are compared and mutation is biased towards substitution.

Overall, the use of program-specific search operators leads to about a fivefold improvement in search efficiency. Note that the results in Figure 12 show the performance of the various cost functions to be broadly similar. This is probably explained by the fact that, with program-specific search operators, much less search is performed and hence the cost function is likely to have less influence on the overall performance. In the case of Stem, for example, no guided search was required. The initial population strings, created from the program literals, were sufficient to achieve branch coverage.

Table V shows the number of executions required to find test data to satisfy the regular expressions in the program Regex given the use of program-specific search operators. The improvement compared to the previous results is not as great as was found for the string equality predicates. This is explained by the absence of suitable string literals in the Regex program. The string literals that appear within a regular expression represent only parts of the string that is to be matched.

4.3. Discussion

A program may contain many string literals. For example, the Calc program contains a 'switch' statement in which each branch compares a string from the input against a specific literal. Although the program may contain many literal strings, in searching for test data to execute a specific branch, only one or two of these literals may be useful. For this specific branch, mutation operators that introduce the other literals will slow down the search. A very simple strategy which exploits this observation is

DOI: 10.1002/stvr

Table V. The number of executions required to find test data to achieve branch coverage (averaged over 20 trials).

	No program-dependent operators		Program-dependent operators	
Program name	No mutation bias	Bias to substitution	No mutation bias	Bias to substitution
Regex	16 122	8023	13 762	6810

to bias the search for a given branch that contains one or more literals to those literals. This strategy would have improved the performance of the program-specific operators for almost all of the sample test programs. More generally, the literals that should be used to bias the search are those that appear in any statement that may influence the branch predicate expression. Such literals could be identified from a data-flow analysis of the program.

5. CONCLUSIONS

This paper considers the problem of generating test data where the test data is intended to cover program branches which depend on string predicates such as string equality, string ordering and regular expression matching. Current work in automatic test data generation has been limited largely to programs containing predicates that compare numbers and almost no work has been done on generating test data to satisfy string predicates.

A dynamic test data generation approach is adopted and the problem is seen as one of defining appropriate search operators and corresponding cost functions with which to guide a search.

A relatively simple but important aspect of the search for string data is the definition of the search space of strings. The space of 16-bit character strings is far larger than the space that need be searched in practice, being the space restricted to strings containing characters in the seven low-order bits.

For string equality, an adaptation of the binary Hamming distance was considered, together with two new string-specific match cost functions. New cost functions for string ordering and regular expression matching were also defined.

For string equality, a version of the edit distance cost function with fine-grained costs based on the difference in character ordinal values was found to be the most effective in a small empirical study. In addition, a progressive increase in the probability with which the character substitution mutation operator is applied has also been shown to improve the performance of the search. Two functions for string ordering were investigated but there was no significant difference in their performance in the limited empirical investigation.

The most significant improvement in performance, however, was obtained by exploiting the presence of string literals in programs that process string data. This paper presents program-dependent string search operators that focus the search in the region of such string literals. In the empirical investigation, the use of these operators was shown to give a fivefold increase in performance. The use of program-dependent string search operators has been shown to be far more important than the particular choice of cost function that guides the search.

DOI: 10.1002/stvr

REFERENCES

1. McMinn P. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability* 2004; **14**(2):105–156.
2. King JC. A new approach to program testing. *Proceedings of the International Conference on Reliable Software*, Los Angeles, CA, April 1975. ACM Press: New York, 1975; 228–233.
3. King JC. Symbolic execution and program testing. *Communications of the ACM* 1976; **19**(7):385–394.
4. Beizer B. *Software Testing Techniques* (2nd edn). Van Nostrand Reinhold: New York, 1990.
5. Duran JW, Ntafos SC. An evaluation of random testing. *IEEE Transactions on Software Engineering* 1984; **10**(4):438–444.
6. Coward PD. Symbolic execution and testing. *Information and Software Technology* 1991; **33**(1):53–64.
7. Tracey N, Clark J, Mander K. Automated program flaw finding using simulated annealing. *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 1998)*, Clearwater Beach, FL, March 1998. Published as *Software Engineering Notes* 1998; **23**(2):73–81.
8. Holland JH. *Adaptation in Natural and Artificial Systems*. University of Michigan Press: Ann Arbor, MI, 1975.
9. Goldberg DE. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley: Reading, MA, 1989.
10. Wegener J, Baresel A, Sthamer H. Evolutionary test environment for automatic structural testing. *Information and Software Technology (Special Issue on Metaheuristic Algorithms in Software Engineering)* 2001; **43**(14):841–854.
11. Harman M, Hu L, Hierons R, Baresel A, Sthamer H. Improving evolutionary testing by flag removal. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, New York, July 2002. Morgan Kaufmann: San Francisco, CA, 2002; 1359–1366.
12. Korel B. Automated software test data generation. *IEEE Transactions on Software Engineering* 1990; **16**(8):870–879.
13. Jones BF, Sthamer H, Eyres DE. Automatic structural testing using genetic algorithms. *Software Engineering Journal* 1996; **11**(5):299–306.
14. Zhao R, Lyu MR. Character string predicate based automatic software test data generation. *Proceedings of the 3rd International Conference on Quality Software (QSIC 2003)*, Dallas, TX, November 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003; 255–263.
15. Stutz D, Neward T, Shilling G. *Shared Source CLI Essentials*. O'Reilly: Sebastopol, CA, 2003.
16. Navarro G. A guided tour to approximate string matching. *ACM Computing Surveys* 2001; **33**(1):31–88.
17. Bottaci L. Predicate expression cost functions to guide evolutionary search for test data. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003): Part II*, Chicago, IL, July 2003 (*Lecture Notes in Computer Science*, vol. 2724). Springer: Berlin, 2003; 2455–2464.
18. Myers EW, Miller W. Approximate matching of regular expressions. *Bulletin of Mathematical Biology* 1989; **51**(1):7–37.
19. Whitley D. The GENITOR algorithm and selective pressure: Why rank based allocation of reproductive trials is best. *Proceedings of the 3rd International Conference on Genetic Algorithms*, Fairfax, VA, June 1989. Morgan Kaufmann: San Francisco, CA, 1989; 116–121.
20. Davis L. *Handbook of Genetic Algorithms*. International Thomson Computer Press: London, 1996.
21. Schwefel H-P. *Evolution and Optimum Seeking*. Wiley: New York, 1995.

DOI: 10.1002/stvr