A Novel Linear-Order Algorithm for Adaptive Random Testing of Programs with Non-Numeric Inputs

Arlinta C. Barus
Institut Teknologi Del
email: arlinta@del.ac.id

Tsong Yueh Chen Fei-Ching Kuo
Department of Computer Science and Software Engineering
Swinburne University of Technology, Australia
email: {tychen, dkuo}@swin.edu.au

Huai Liu
Australia-India Research Centre for Automation Software Engineering
RMIT University, Australia
email: huai.liu@rmit.edu.au

Robert Merkel
Clayton School of Information Technology
Monash University, Australia
email: robert.merkel@monash.edu

Gregg Rothermel
Department of Computer Science and Engineering
University of Nebraska - Lincoln
email: grother@cse.unl.edu

November 28, 2014

Abstract

Adaptive Random Testing (ART) is a family of testing techniques that aim to enhance the failure-detection effectiveness of random testing (RT) by spreading random test cases evenly throughout the input domain. ART has been empirically shown to be effective on software with numeric inputs. However, there are two aspects that need further improvement to make ART's adoption more widespread — applicability to programs with non-numeric inputs, and the high computation overhead of many ART algorithms. We present a linear-order ART algorithm for software with non-numeric inputs. The key requirement for using ART with non-numeric inputs is an appropriate "distance" measure. We use the concepts of categories and choices from category-partition testing to formulate such a measure. We investigate the failure-detection effectiveness of our technique by performing an empirical study on 14 object programs, using two standard metrics — F-measure and P-measure, being compared to RT and another linear-order ART algorithm using the forgetting technique. Our ART algorithm statistically significantly outperforms RT, according to both F-measure and P-measure, on 10 of the 14 programs studied, and exhibits performance similar to RT on three of the four remaining programs. The selection overhead of our ART algorithm is close to that of RT.

Keywords: Random testing, adaptive random testing, category-partition method.

1 Introduction

Random Testing (RT) — that is, testing software by randomly generating inputs — is a standard black-box software testing approach. The IEEE Guide to the Software Engineering Body of Knowledge (SWEBOK) [9, Section 3.2.4 of Chapter 4] lists RT as one of four common input-domain based testing techniques, along with equivalence partitioning, pairwise testing, and boundary value analysis. Orso and Rothermel [37] categorize RT as one of four major test input generation techniques, while the other three are symbolic execution-based, search-based , and hybrid techniques. Arcuri, Iqbal, and Briand [6] observe that RT is "one of the most used automated testing techniques in practice".

RT has been shown to be practically useful in many contexts. One variant of RT, fuzz testing, revealed the widely-publicized Heartbleed security vulnerability in the OpenSSL library [22], and was used by Microsoft as a primary testing technique for developing secure software [30]. Regehr [42] used RT to find previously undetected real-life faults in interrupt-driven embedded software. Groce et al. [26] report on the use of RT to validate mission-critical flight software for uncrewed space probes. There are many other reports of similarly successful applications, including case studies on Java and .NET libraries [39], database systems [7], and real-time embedded systems [5]. RT has been used for both unit testing [39] and system testing [7, 48].

Direct comparisons between RT and other state-of-the-art testing methods show that RT often compares very favorably to them. Sharma et al. [44] empirically compared the effectiveness of RT and shape abstraction, which they note is a "state-of-the-art systematic technique". They found that RT was similar in effectiveness to shape abstraction, and had much lower computational overhead; therefore, it was more cost-effective. Furthermore, they found three real faults with RT that had not previously been detected. Arcuri et al. [6] conducted a theoretical analysis of RT and concluded that "there are practical situations in which random testing is a viable option." Arcuri et al. [5] also observed that RT can be tailored to "whatever time and resources are available for testing", unlike, for instance, coverage-based strategies. The combination of industrial interest and empirical and theoretical support makes RT an important area for continued research.

Adaptive Random Testing (ART) [20] is a class of testing techniques designed to improve the failure-detection effectiveness of RT by increasing the diversity across a program's input domain of the test cases executed. The Fixed-Size Candidate Set ART technique (FSCS-ART) was the first ART technique, and is also the most widely studied. To generate an additional test case using FSCS-ART, a number of candidate test cases are randomly generated. The candidate that is the most "distant" from previously executed test cases, according to a criterion known as the *max-min criterion*, is selected as the next test case, while other candidates are discarded. The *Cartesian distance* measure is used to determine the distance between numeric inputs.

Various studies using programs with numeric inputs [15, 17, 31, 32, 45] have shown that several ART techniques require substantially fewer test cases than RT to reveal failures in programs. However, as Ciupa et al. [21] observe, test case selection overhead can result in FSCS-ART having poorer overall cost-effectiveness than RT. The reduction in test cases required to reveal failures was, in their experiments, outweighed by selection overhead. Arcuri and Briand [3] argue that the high selection overhead of FSCS-ART renders it unsuitable for practical use. They also observe that the effectiveness of FSCS-ART on programs with very low failure rates has not been studied – a fact that, itself, can be attributed to high selection overhead. A number of techniques, such as mirroring [14] and forgetting [10], have been proposed to reduce the overhead of various ART algorithms. More recently, Shahbazi et al. [43] proposed a new ART approach, Random Border Centroidal Voronoi Tessellations (RBCVT), that takes advantage of the properties of the Voronoi tessellation to achieve test case diversity. The authors developed a novel algorithm (RBCVT-Fast) that has an $O(n)$ selection overhead (that is, the process of generating n test cases takes $O(n)$ time). However, RBCVT-Fast, as presented, is only directly applicable to simple input domains representable as a d-dimensional real space.

In this paper, we present an ART algorithm that can be applied to software with non-numeric, structured input formats, retains FSCS-ART's failure-revealing effectiveness, and has an $O(n)$ selection overhead. Because the Cartesian distance measure used for numeric inputs cannot be directly applied to non-numeric inputs, we provide an approach that relies on the concepts of *categories* and *choices*, originally proposed as part of the *category-partition* black-box testing technique [38], to form the basis of a new "distance measure". We evaluate the effectiveness of this ART algorithm on 14 object programs that have non-trivial input formats – seven programs from the Software-artifact Infrastructure Repository (SIR) [24], six standard Unix utilities, and the larger GNU `grep` program. We evaluate our approach using two standard effectiveness metrics, the *F-measure* and *P-measure*, as well as test case generation time.

The remainder of this article is organized as follows. Section 2 provides essential background on ART, followed by a brief description of the concepts of categories and choices. Section 3 specifically describes the theoretical framework for applying ART to non-numeric software, and our linear-order ART algorithm to take into account the characteristics of our distance measure. Section 4 presents our empirical study, including details on the study setup. Section 5 presents our experiment results, including quantitative statistical analysis of those results. Section 6 presents further interpretation and discussion of the results. Section 7 discusses related work. Some concluding thoughts, including recommendations for future study, are offered in Section 8.

2 Preliminaries and Background

2.1 Adaptive Random Testing

As previously stated, to improve the performance of RT, Chen et al. [20] proposed an enhanced technique for RT, known as ART. Their approach was based on the intuition of "even spread". A number of studies [2, 8, 46] have found evidence that faults tend to cause erroneous behavior to occur in contiguous regions of the input domain. Therefore, Chen et al. [20] argued that two test cases whose inputs were "close" to each other in the input domain were more likely to have the same execution behavior than two test cases that were more "widely separated". Hence, they reasoned that a technique that spreads test cases more evenly would identify failures using fewer test cases.

To implement this idea, Chen et al. [20] introduced a distance-based ART algorithm, also known as Fixed-

Size Candidate Set ART (FSCS-ART). In the FSCS-ART algorithm, two sets of test cases are considered: the *executed set*, **E**, which records those test cases that have already been executed and have not revealed a failure, and the *candidate set*. To select a new test case, a set of k candidates $(c_1, c_2, ..., c_k)$ is first "generated randomly" as the candidate set. From these, the best candidate c_o is selected according to a criterion. The remaining candidates are then discarded, and testing is conducted with c_o, which is then added to **E**. Testing continues until a pre-specified stopping criterion is met, such as the detection of failures, the execution of the required number of tests, or the exhaustion of testing resources.

In the original work on FSCS-ART, the criterion used was the *max-min* criterion. For each candidate c_i, the Cartesian distance to each member of **E** is calculated, and the smallest distance for c_i is recorded as d_i. Of the candidates, the candidate c_o with the largest d_i is selected (in other words $d_o \geq d_i \; \forall i, 1 \leq i \leq k$); if multiple candidates have the same distance value, one is chosen arbitrarily. An alternative selection criterion is the *max-sum* criterion. In this case, for each candidate, the *sum* of the distances to each member of **E** is calculated, and the candidate for which this sum is the largest is chosen.

ART algorithms may consider the entire set of previously executed test cases when selecting the best candidate. However, as Chan et al. [10] showed, it is possible to greatly reduce the selection overhead of ART techniques, while retaining much or all of their failure-revealing effectiveness, by evaluating only a subset of **E** when selecting the best candidate. Chan et al. call this technique *forgetting*.

2.2 Categories and choices

To test software with non-numeric input formats using FSCS-ART, two things are required:

- A method for randomly sampling inputs from the software's input domain.
- A way of measuring the "distance" between elements of the software's input domain.

The first requirement is common to all RT techniques, while the second is unique to ART; hence, the latter is the focus of this work.

To understand our new distance measure, it is first necessary to consider why the Cartesian distance is an effective distance measure for ART on software with numeric inputs. Most numerical software consists primarily of compositions of smooth continuous functions [8]. Given two inputs that are close to each other, as measured by the Cartesian distance, it is likely that the execution patterns of that software will be similar, and thus that the failure behavior of the software will also be similar; that is, either the outputs on both inputs will conform to the specification, or they will not. It is this similarity in execution patterns that we seek to measure in a broader range of software.

To achieve this, we have developed an approach based on the concepts of *categories* and *choices* from the *category-partition method*. The category-partition method is a black-box functional testing method, similar to the earlier technique of testing based on equivalence classes [35], and first introduced by Ostrand and Balcer [38]. In this method, the tester must identify input parameters or environmental conditions that affect the execution of the functional unit under test. These properties are characterized in terms of *categories*. Each category is then partitioned into disjoint partitions, called *choices*, that cover values the category may take. Each choice represents "a set of similar values that can be assumed by the type of information in the category". For instance, consider a transaction processing system that handles a large range of monetary and non-monetary quantities (for instance, it may deal with cash and credit transactions, and the transfer of items from a stock inventory); an appropriate category may be "unit type", with choices "cash", "credit"', and "inventory item". Here, a transaction involving a cash amount of \$123.45 would have a unit type of "cash", while a transaction involving the transfer of 10 widgets would have a unit type of "inventory item".

In the category-partition testing method, constraints (stated within the software specification) are used to identify which combinations of categories and choices are valid, and which are not. Then, all valid combinations of categories and choices are generated as *test frames* – the basis of the test suite. Each test frame is then fleshed out into a concrete test case using representative data for each choice in the frame. In our present work, we do not utilize the concept of constraints; we simply use the concepts of categories and choices to formulate a distance measure for ART. Our approach is presented in detail in Section 3.1.

3 Theoretical Framework

3.1 A distance measure for non-numeric, structured inputs

Our approach to distance measurement, originally presented in Merkel [34] and Kuo [29], makes use of the concepts of categories and choices from the category-partition method described in Section 2.2.

In the category-partition method, categories and choices are used to obtain test frames, and concrete test cases are generated from these. In our approach, we work in the opposite manner: for a given concrete input, we identify its relevant test frame. Categories and choices are still defined as described above. However, rather than simply generating all valid test frames from the defined categories and choices, we take two program inputs, determine their categories and choices, and use this information to calculate the distance between them, with a greater distance representing the situation in which the two inputs are more dissimilar. Technically speaking, given two program inputs x and y, our distance measure is a count of the number of categories in which x and y have different choices.

More formally, let us denote the set of categories by $\mathbf{A} = \{A_1, A_2, \ldots, A_g\}$, where g denotes the total number of categories. For each A_i, let us denote its choices by $\mathbf{P}_i = \{p_1^i, p_2^i, \ldots, p_h^i\}$, where h denotes the number of choices for A_i. We remind the reader that the choices for a single category are disjoint.

It should be noted that any input is a combination of input values chosen such that the inputs correspond to choices from a non-empty subset of $\mathbf{A}$. For input x, let us denote the corresponding non-empty subset by $\mathrm{A}(x) = \left\{A_1^x, A_2^x, \ldots, A_q^x\right\}$, where q refers to the number of categories associated with x. Since categories are distinct and their choices are disjoint, input x in fact consists of values chosen from a non-empty subset of choices, denoted as $\mathrm{P}(x) = \left\{p_1^x, p_2^x, \ldots, p_q^x\right\}$, where p_i^x $(i = 1, 2, \ldots, q)$ is the choice of the category A_i^x for x.

For any two inputs x and y, we define $\mathrm{DP}(x, y)$ as the set that contains elements in either $\mathrm{P}(x)$ or $\mathrm{P}(y)$ but not both. That is, $\mathrm{DP}(x, y) = (\mathrm{P}(x) \bigcup P(y)) \setminus (\mathrm{P}(x) \bigcap \mathrm{P}(y))$. Now, we define $\mathrm{DA}(x, y) = \left\{A_m | A_i \text{ if } \exists p_j^i \in \mathrm{DP}(x, y)\right\}$. In other words, $\mathrm{DA}(x, y)$ represents the set of categories in which inputs x and y have different choices. Then, the distance measure between x and y is defined as $|\mathrm{DA}(x, y)|$ (the size of $\mathrm{DA}(x, y)$); that is, the *number* of categories that appear in either x or y but not both, or in which the choices in x and y differ.

For example, consider again the transaction processing system used in Section 2.2. Assume that we use the categories and choices shown in Table 1 (note that each choice represents a group of potential values):

Assume that we have three program inputs, x, y, and z, for the transaction processing system, the processed transactions and relevant categories/choices of which are summarized in Table 2.

We can calculate DP, DA, and $|\mathrm{DA}|$ for each pair of these three inputs as shown in Table 3. By our measure, x and y have a distance of 3, x and z have a distance of 1, and y and z have a distance of 3.

Table 1: An Example of Categories and Choices

Category	**Choice**
Unit type	Cheque
	Credit
	Inventory item
Customer type	Business
	Personal
	Government
	Other
Status	Accepted
	Rejected

Table 2: Three Example Inputs

Input	**Processed Transaction**	**Category and Choice**
x	A cleared cheque payment of \$123.45 from Anycorp, a business customer.	Unit type:Cheque
		Customer type:Business
		Status:Accepted
y	A credit card payment of \$543.21 from Mr. Fred Phisher, a personal customer whose dubious identity leads to the payment being rejected.	Unit type:Credit
		Customer type:Personal
		Status:Rejected
z	The dispatch of 12 widgets from stock to Othercorp, a business customer. The order is accepted.	Unit type:Inventory item
		Customer type:Business
		Status:Accepted

3.2 A linear-time ART algorithm

We now present an ART algorithm for structured inputs using the category-choice distance measure that takes advantage of the measure's particular properties to achieve a linear test case selection time (i.e., selecting n test cases takes $O(n)$ time). Compared to FSCS-ART, our algorithm also requires a candidate set, but uses the max-sum criterion in an innovative way that calculates the sum of the distance between each candidate and all previously executed test cases. We call this algorithm "*ARTsum*".

Before we present the algorithm, let us first briefly recall the naive implementation of the max-sum criterion as follows. Suppose that n test cases have been selected and executed, denoted by $\mathbf{E} = \{e_1, e_2, \ldots, e_n\}$. Each test case e_j $(j = 1, 2, \ldots, n)$ is associated with a set of choices $\mathrm{P}(e_j) = \left\{p_1^{e_j}, p_2^{e_j}, \ldots, p_q^{e_j},\right\}$. $\mathrm{P}(e_j)$ can be rewritten as a tuple $\mathrm{R}(e_j) = \left(r_1^{e_j}, r_2^{e_j}, \ldots, r_g^{e_j}\right)$, where g is the total number of categories associated with the software, $r_i^{e_j} = 0$ means that e_j is not associated with category A_i, and $r_i^{e_j} = l$ $(l \geq 1)$ means that e_j is associated with the lth choice of A_i. Similarly, a candidate c can also be associated with the tuple

Table 3: Calculation of Distances Among x, y, and z

Between the pair of	**DP**	**DA**	**|DA|**
(x, y)	Unit type:Cheque	Unit type	3
	Unit type:Credit		
	Customer type:Business	Customer type	
	Customer type:Personal		
	Status:Accepted	Status	
	Status:Rejected		
(x, z)	Unit type:Cheque	Unit type	1
	Unit type:Inventory Item		
(y, z)	Unit type:Credit	Unit type	3
	Unit type:Inventory Item		
	Customer type:Personal	Customer type	
	Customer type:Business		
	Status:Rejected	Status	
	Status:accepted		

$\mathrm{R}(c) = \left(r_1^c, r_2^c, \ldots, r_g^c\right)$.

Define a function on e_j and c as follows:

$$\mathrm{D}(i,j) = \begin{cases} 0 & \text{if } r_i^c = r_i^{e_j}, \\ 1 & \text{if } r_i^c \neq r_i^{e_j}, \end{cases} \tag{1}$$

where $i = 1, 2, \ldots, g$ and $j = 1, 2, \ldots, n$. The distance between c and e_j can then be calculated as $\mathrm{dist}(c, e_j) = \sum_{i=1}^{g} \mathrm{D}(i,j)$. Note that $\mathrm{dist}(c, e_j)$ is effectively equal to $|\mathrm{DA}(c, e_j)|$; these two different calculation techniques always return the same result.

Given the foregoing, the sum of the distances from c to all executed test cases can be calculated as:

$$\mathrm{sum\_dist}(c, \mathbf{E}) = \sum_{j=1}^{n} \left(\sum_{i=1}^{g} \mathrm{D}(i,j) \right). \tag{2}$$

Clearly, if we calculate the sum of the distances according to Equation 2, the selection of the next test case requires $O(n)$ time (note that g is a constant). Therefore, a naive implementation of max-sum using Equation 2 has a computation overhead of $O\left(n^2\right)$ for selecting n test cases.

Recall the three example test cases x, y, and z shown in Table 2. Suppose that $\mathbf{E} = \{x, y\}$ and z is the candidate. We have $\mathrm{R}(x) = (1,1,1)$, $\mathrm{R}(y) = (2,2,2)$, and $\mathrm{R}(z) = (3,1,1)$. Then, we can calculate $\mathrm{dist}(x,z) = 1+0+0 = 1$ and $\mathrm{dist}(y,z) = 1+1+1 = 3$ (as also given in Table 3). Hence, we finally get $\mathrm{sum\_dist}(z, \mathbf{E}) = 1+3 = 4$.

Our linear-order *ARTsum* algorithm is based on the following theorem.

Theorem 1. *Define a tuple of integers* $\mathbf{S} = \left(s_1^0, s_1^1, \ldots, s_1^{h_1}, s_2^0, s_2^1, \ldots, s_2^{h_2}, \ldots, s_g^0, s_g^1, \ldots, s_g^{h_g}\right)$*, where* g *is the total number of categories,* h_i *is the total number of choices for the* i*th category* A_i $(i = 1, 2, \ldots, g)$*,* s_i^0 *denotes the number of previously executed test cases that are not associated with the* i*th category* A_i*,* $s_i^{l_i}$ $(l_i = 1, 2, \ldots, h_i)$ *denotes the number of previously executed test cases associated with the* l_i*th choice of* A_i*. Let* n *denote the number of previously executed test cases, that is,* $n = |\mathbf{E}|$*. By definition,* $\sum_{v=0}^{h_i} s_i^v = n$ $\forall i, 1 \le i \le g$*. For a candidate* c *associated with* $\mathrm{R}(c) = \left(r_1^c, r_2^c, \ldots, r_g^c\right)$*, the sum distance between* c *and* $\mathbf{E}$ *is:*

$$\mathrm{sum\_dist}(c, \mathbf{E}) = \sum_{i=1}^{g} \left(n - s_i^{r_i^c}\right). \tag{3}$$

Proof. For each r_i^c $(i = 1, 2, \ldots, g)$, we can find a corresponding value $s_i^{r_i^c}$ from $\mathbf{S}$, where $s_i^{r_i^c}$ effectively means the number of executed test cases that satisfy $r_i^c = r_i^{e_j}$. Therefore, $\left(n - s_i^{r_i^c}\right)$ is equal to the number of executed test cases that satisfy $r_i^c \neq r_i^{e_j}$. According to Equation 1, $\left(n - s_i^{r_i^c}\right) = \sum_{j=1}^{n} \mathrm{D}(i,j)$. Following Equation 2:

$$\mathrm{sum\_dist}(c, \mathbf{E}) = \sum_{j=1}^{n} \left(\sum_{i=1}^{g} \mathrm{D}(i,j) \right) = \sum_{i=1}^{g} \left(\sum_{j=1}^{n} \mathrm{D}(i,j) \right) = \sum_{i=1}^{g} \left(n - s_i^{r_i^c}\right).$$

Thus, Theorem 1 holds; that is, Equation 3 gives exactly the same results as Equation 2. $\square$

Consider the three test cases x, y, and z in Table 2 again. Given that $\mathbf{E} = \{x, y\}$, we can let $\mathbf{S} = (0,1,1,0,0,1,1,0,0,1,1)$, where $s_1^0 = 0$ because both x and y contain a choice for the first category, $s_1^1 = 1$ because only x contains the first choice of the first category, $\cdots$, $s_3^2 = 1$ because only y contains the second choice of the third category. For the candidate z, we let $\mathrm{R}(z) = (3, 1, 1)$. According to Equation 3, we can calculate $\text{sum\_dist}(z, \mathbf{E}) = (n - s_1^3) + (n - s_2^1) + (n - s_3^1) = (2 - 0) + (2 - 1) + (2 - 1) = 4$.

Theorem 1 implies that if Equation 3 is used, the selection of a next test case requires a constant time. Now, we present our Algorithm *ARTsum*.

Algorithm *ARTsum*

```
1: Initialize $\mathbf{S}$ (as defined in Theorem 1) by setting each $s_i^{l_i}$ as 0, where $i = 1, 2, \ldots, g$ ($g$ denotes the total number
   of categories)
2: Set $n = 0$ and $\mathbf{E} = \{\}$
3: Define an integer $k > 0$ as the number of candidates to be generated
4: while Termination condition is not satisfied do
5:     Increment $n$ by 1
6:     if $n = 1$ then
7:         Randomly generate a test case $e_n$
8:     else
9:         Randomly generate $k$ candidates $c_1, c_2, \ldots, c_k$
10:        for all $c_u$ $(u = 1, 2, \ldots, k)$ do
11:            Calculate $\text{sum\_dist}(c_u, \mathbf{E})$ according to Equation 3
12:        end for
13:        Set $e_n = c_o$, where $\forall u, \text{sum\_dist}(c_o, \mathrm{E}) \geq \text{sum\_dist}(c_u, \mathrm{E})$
14:    end if
15:    Add $e_n$ into $\mathbf{E}$
16:    Update $\mathbf{S}$ by incrementing each $s_i^{r_i^{e_n}}$ by 1, where $i = 1, 2, \ldots, g$
17: end while
```

In the *ARTsum* algorithm, **S** is dynamically updated during the testing process. Once the candidate c_o with the largest sum distance is selected as the new test case e_n (refer to Line 13 in the Algorithm), we update **S** accordingly by incrementing each $s_i^{r_i^{e_n}}$ $(i = 1, 2, \ldots, g)$ by 1 (refer to Line 16 in the Algorithm). Note that both updating **S** after executing a test case and the distance calculation for the candidate using Equation 3 are independent of the number of test cases; therefore selecting a single test case takes constant time. Thus, selecting a set of n test cases takes $O(n)$ time.

4 Empirical Study

4.1 Research questions

In the previous section we proved that the *ARTsum* algorithm generates test cases in linear time. However, we also wish to evaluate the approach's failure-detection effectiveness, and empirically assess its computational overhead, in order to draw conclusions about its cost-effectiveness in practice. Therefore, we conducted an empirical study directed at the following research questions:

RQ1 How effective is the *ARTsum* algorithm at revealing failures?

RQ2 How does the actual selection overhead of the *ARTsum* algorithm compare to its overhead calculated via theoretical complexity analysis, and to the overhead of alternative techniques?

4.2 Object programs

To address our research questions, we chose to study three sets of object programs: seven programs from the Software-artifact Infrastructure Repository (SIR) [24], six small Unix utilities, and the regular expression

Table 4: 14 C Programs as Experimental Objects

Name	**Source**	**Brief description**	**LOC**	**No. of faults**
`printtokens`	SIR	lexical analyzer	483	7
`printtokens2`	SIR	lexical analyzer	402	10
`replace`	SIR	search and replace tool	516	31
`schedule`	SIR	scheduler	299	9
`schedule2`	SIR	scheduler	297	9
`tcas`	SIR	collision alarm logic	138	41
`totinfo`	SIR	basic statistics	346	23
`cal`	SunOS	calendar display	163	11
`comm`	SunOS	file comparator	144	27
`look`	SunOS	file searcher	135	29
`sort`	SunOS	file sorter	842	48
`spline`	SunOS	curve interpolation	289	16
`uniq`	SunOS	file comparator	125	29
`grep`	GNU	regular expression processor	3,161	20
Total			7,340	310

processor component of the larger utility program GNU `grep`. The fourteen programs, all implemented in C, are summarized in Table 4.

These three sets of programs present complementary strengths and weaknesses as experiment objects. The SIR repository provides object programs, including a number of pre-existing versions with seeded faults, as well as a pool of test cases that can be further fuzzed to provide test cases for RT and candidates for ART. However, these programs are small and there are only a limited number of faulty versions available for each program. The Unix utilities that we use are also relatively small and simple, but they are provided with sets of faults in the form of mutants. The `grep` program (even when restricting attention to its regular expression processor component) is a much larger system for which mutation faults could be generated. We provide further details on each of these sets of object programs next.

4.2.1 Program set 1: SIR programs

The seven object programs selected from the SIR [24] repository were `printtokens`, `printtokens2`, `replace`, `schedule`, `schedule2`, `tcas`, and `totinfo`. These programs were originally compiled by Hutchins et al. [28] at Siemens Corporate Research. We used these programs for several reasons:

- The programs are of manageable size and complexity for an initial study.
- The input format of the programs is non-trivial, but manageable.
- Faulty versions of the programs are available.
- All programs and related materials are available from the SIR, facilitating replication of our studies.

The `printtokens` and `printtokens2` programs are independent implementations of the same specification. They each implement a lexical analyzer. Their input files are split into lexical tokens according to a tokenizing scheme, and their output is the separated tokens.

The `replace` program is a command-line utility that takes a *search string*, a *replacement string*, and an *input file*. The search string is a regular expression (using its own unique format different from, for instance, that of `grep`). The replacement string is text, with some metacharacters to enable certain features. The `replace` program searches for occurrences of the search string in the input file, and produces an output file where each occurrence of the search string is replaced with the replacement string.

The `schedule` and `schedule2` programs are also independent implementations of the same specification. They each implement a priority scheduler that takes three non-negative integers and an input file. The integers indicate the number of initial processes at the three available scheduling priorities. The `schedule` and `schedule2` programs then take as input a command file that specifies certain actions to be taken – for instance, adding a new process for the scheduler to manage. The programs then calculate the scheduling of the processes. The output of the programs is a list of numbers indicating the order in which processes exit from the system.

The `tcas` program is a small part of a much larger aeronautical collision avoidance system. It performs simple analysis on data provided by other parts of the system (or, in this case, the experimenter) and returns an integer indicating what action, if any, the pilot should take to avoid collision.

The `totinfo` program computes some descriptive summary statistics from its inputs.

For the SIR programs, we used the existing faulty versions present in the repository for comparison. While these were seeded faults rather than actual ones, they were created by multiple persons based on their own experiences with faults. Table 4 lists the numbers of faults utilized for each of the programs. For `replace`, one of the faults in SIR was not killed by any of the existing test cases in the test pool, so we excluded this fault from our study. We also excluded a `schedule2` fault for the same reason.

4.2.2 Program set 2: Unix utilities

The second set of object programs used in this study is a set of Unix utilities, `cal`, `comm`, `look`, `sort`, `spline`, and `uniq`. These object programs were distributed as part of SunOS 5.8 and appear to be part of BSD 4.3. These programs are part of a set of ten programs used for test suite reduction experiments by Wong et al. [47], and in a prior analysis of the coverage achieved by ART [12].

These Unix utilities were not provided with test data, thus we generated test data for them randomly.

For these Unix utilities, faults in the form of program mutations had previously been generated by the automated C mutation tool, Proteum [23]. Proteum can apply a total of 71 mutation operators in four classes: statement, operator, variable, and constant. All of these operators were applied to create mutations in the programs. Not all generated mutations were used, as some failed on virtually every test case, whereas others produced behavior equivalent to that of the original program. We used the following procedure to filter the initial set of mutants provided with the program for use in this study.

- Determine the failure rates of the mutants using RT with a sample size of 100,000.
- Discard all mutants that are not killed by any of the 100,000 random test cases.
- Discard all mutants with failure rates greater than 0.1.
- Identify mutants that have exactly the same set of failure-revealing inputs. For each such set of mutants, randomly select one for use in the study.

After performing this filtering process, we found that four of the programs utilized initially by Wong et al. (`checkeq`, `col`, `crypt`, and `tr`) exhibited very high failure rates (greater than 0.1) among mutants. In other words, it is easy to kill the mutants in these programs, and the mutants are thus liable to be detected by any testing technique. Thus, we excluded these four programs from further study.

4.2.3 Program set 3: GNU `grep`

Our final program set involves version 2.5.1a of the GNU `grep` [41] program.[1] This program's "man" page describes it as follows:

> The `grep` command searches one or more input files for lines containing a match to a specified pattern. By default, `grep` prints the matching lines.

We chose `grep` for our study for several reasons:

- As a GNU project, current and historical versions are freely available including source code and a partial, but still useful, change history. As with the SIR programs, this permits replication of our study.
- The `grep` program is in wide use, providing an opportunity to demonstrate the real world relevance of our techniques.
- The `grep` program, and its input format, are of greater complexity than the the programs in the other test sets, but still manageable as a target for automated test case generation.

The `grep` program's input format is typical of Unix command-line utilities. We categorize its inputs into three components: *options*, which consist of a list of commands to modify the searching process, *pattern*, which is the regular expression to be searched for, and *files*, which refers to the input files (these may also include special "files" such as the Unix standard input) to be searched.

The `grep` program required a different approach to testing than the SIR programs, because the scope of its functionality is larger. Constructing test infrastructure to test all of `grep`'s functionality would have been impractical. Therefore, we restricted our focus to the regular expression analyzer of `grep`. This portion of `grep` was still much larger than the other programs studied, consisting of 3,161 lines of code and 1,423 branches.

As faults in `grep`, while we did consider using the faults present in SIR, most of these did not relate to the regular expression analyzer, and were therefore not useful to us. Furthermore, an examination of `grep`'s software change log (included in its distribution) showed that the vast majority of the faults found and fixed in `grep` were either platform-specific, or manifest themselves so rarely that they would render experimental comparisons impractical.

We were able to find one reported `grep` fault in the public version history for the program that was suitable for our use. The fault, documented in the change log file, indicates that certain regular expression ranges were compared using raw byte codes to range boundaries, rather than using the relevant collation sequence determined by the particular Unix locale [40]. Obviously, this fault is exposed only when the locale setting is different from the default setting, so we used the locale "en_US.UTF-8" while performing testing to detect the fault.

One real fault is not sufficient to support a comprehensive study, so we also used program mutation to generate additional faulty versions of `grep` for our experiment. Due to limitations in the ability to restrict Proteum to creating mutants for a specific part of `grep` (the regular expression analyzer) it was impractical to use it to generate sufficient mutants for `grep`. Therefore, we developed a custom tool that applied two types of mutation operators – *statement mutation* and *operator mutation*. One statement mutation operator

[1] A version of `grep` is present in the SIR. However, because of compatibility issues between the SIR version and our experiment platform, we did not use that version.

that we applied replaced `continue` statements with `break` statements and *vice versa* – these statements are common in the regular expression analyzer in `grep`. Another statement mutation operator replaced labels on `goto` statements. The operator mutation replaced a single arithmetic or logical operator with another. Each mutant had only one mutation operation applied to it. We generated a total of 19 mutants for our study, resulting in a total of 20 faulty versions of `grep`.

4.3 Variables and measures

4.3.1 Independent variable

The independent variable in our experiment is the test case selection strategy. As choices for this variable, we include, of course, an implementation of the *ARTsum* algorithm described in Section 3. As baseline techniques for use in comparison, we selected two additional techniques, RT and *ARTmif*.

RT (*random testing with replacement*) is a natural baseline choice, because *ARTsum* is designed as an enhancement to RT, and assessing whether *ARTsum* is more cost-effective than RT is important. Generally speaking, an automated oracle is assumed when RT is applied. In our experiments, the base programs (for which seeded faults already existed or were generated) were used to simulate the automated oracles.

The *ARTmif* algorithm is an existing alternative linear-order ART approach, that combines the max-min criterion with forgetting to allow constant-time selection overhead. FSCS-ART can be implemented straightforwardly using the category-choice distance metric and the max-min selection criterion described previously. However, selecting n test cases has an overhead of $O\left(n^2\right)$, which may lead to inferior cost-effectiveness, depending on the failure rate and the execution time of the program under test. A "forgetting" technique can be used to reduce the overhead of the approach to $O\left(n\right)$ if an ART algorithm considers only a fixed-sized subset of the previously executed test cases when selecting the best candidate. However, prior studies [10] on forgetting always arbitrarily define the size of the subset. In this study, we propose a more precise heuristic for conducting the forgetting process.

By definition, there are a finite number of combinations of categories and choices that can be used in testing a program. Therefore, if candidate test cases are selected randomly, the probability that a new candidate has the same categories and choices as a previously executed test case asymptotically approaches one as the number of previously executed test cases increases. In practice, given that many choices appear rarely in the generated candidates and thus some combinations occur much more frequently than others, the executed set $\mathbf{E}$ does not need to be very large before many candidates turn out to have a nearest neighbor at the same distance. If this occurs, effectively speaking, one of the candidates is chosen arbitrarily and the technique degenerates to random testing; which is a very computationally expensive form of RT.

Based on this observation, we define the following forgetting strategy:

- During each round of test case selection, count how many candidates have the same minimum nearest neighbor distance d_o (that is, any of them can be selected as the next test case according to the max-min criterion).
- When the following two conditions are both satisfied, forget all already executed test cases and then perform max-min FSCS-ART from scratch.
  - 90% of the candidates have the same d_o.
  - d_o is less than or equal to the number of categories divided by 10.

Given a finite number of categories and choices, there is an upper bound on the size of the subset of previously executed test cases. In other words, *ARTmif* has a computational overhead of $O(n)$ for generating n test cases.

Both *ARTsum* and *ARTmif* are based on FSCS-ART, which has been the most effective algorithm in previous studies, and consequently is the most widely studied algorithm in the previous literature. *ARTsum*, as presented in Section 3, is in fact an $O(n)$ implementation of FSCS-ART using the selection criterion of "maximum sum of distances to previously executed test cases" that in previous studies has required $O\left(n^2\right)$ time in a naive implementation, and as such it is unnecessary to repeat effectiveness measurements on the $O\left(n^2\right)$ algorithm. As for the choice of *ARTmif* rather than the straightforward FSCS-ART without forgetting using the selection criterion of "maximum nearest neighbor distance to previously executed test cases", we have observed in preliminary experiments that on the same object programs used in this paper, FSCS-ART without forgetting has effectiveness very similar to that of FSCS-ART with forgetting. Given the comparable effectiveness and the fact that FSCS-ART without forgetting has quadratic overhead, it was clearly less cost-effective than the other two approaches and we therefore have omitted the data for FSCS-ART without forgetting.

4.3.2 Dependent variables

The choice of a metric to use in comparing the effectiveness of testing techniques is non-trivial. Frankl et al. [25] argue that testing procedures should be assessed according to the reliability of the delivered software after the testing and debugging process, but such assessments are not feasible for this kind of laboratory study.

For RQ1, to best characterize the failure-detection effectiveness of the techniques, we use two standard metrics: the F-measure and the P-measure [13]. The F-measure is defined as the average number of tests required by a particular strategy to reveal the first failure. A smaller F-measure reflects better failure-detection performance.

The F-measure is particularly appropriate for measuring the failure-detection effectiveness of adaptive testing strategies, such as ART, in which the generation of new test cases depends on the previously executed test cases. However, evaluation of the F-measure requires an automated oracle (because testing must be stopped after failure detection), which may not always be available. Thus, we also used the P-measure, which can appropriately characterize the testing process without an automated oracle. The P-measure is defined as the probability of detecting *at least one failure* with a test suite generated using a particular technique. A larger P-measure reflects better failure-detection performance.

Note that there is another standard metric, the E-measure, that is defined as the *expected number of failures* detected by a test suite that a particular technique generates. We did not use the E-measure because the E-measure is less appropriate than the P-measure: as observed by Shahbazi et al. [43], multiple failures may be associated with the same software fault.

For RQ2, our dependent variable is simply the execution time required for the test case generation techniques to generate test cases.

4.4 Generation of categories and choices for object programs

The categories and choices used for the object programs considered in this study were designed by the authors. In large part, the selection of appropriate categories and choices is at a tester's discretion; we chose what we regarded as simple approaches for emulating that process.

For the programs taken from the SIR, and the small Unix utilities, limited documentation was available, so we inferred the behavior of each program by examining the test inputs and outputs present in the repository, as well as the source code. To avoid a possible source of bias, while designing categories and choices, we did not examine the faults provided in the repository.

As noted previously, our categories and choices for `grep` were designed to test its regular expression analyzer. To obtain these, we first consulted the user documentation. While this documentation was reasonably clear and extensive, it contained some ambiguities and was not complete, so we also examined existing testing information about the version of `grep` contained in the SIR repository.

Precise details on the categories and choices used for the object programs considered in our study are provided in Appendix A.

4.5 Generation of test cases for object programs

To perform ART on any specific object program of interest, both a method for randomly sampling the software's input space and a distance measure are required. Our primary focus in this work is an approach by which ART can consider non-numeric input data. The generation of random samples from the program's input space is a problem shared with RT, and hence not our concern; it should be dealt with in a more general context. We thus focused on obtaining sufficiently random, valid inputs for the object programs, as follows.

4.5.1 Generation for the SIR programs

Each of the SIR programs had an existing pool of test cases. However, these pools were not large enough (consisting of a few thousand test cases per program) to ensure sufficient randomness for our experiments. Therefore, rather than sampling test cases from the existing pools, we used a number of techniques to dynamically generate test cases on demand. Our broad approach for this has some similarities to fuzz testing. We first analyzed the existing test pools to obtain the probability distributions of certain parameters. Then, according to the probability distributions, the concrete values of these parameters could be randomly chosen. We now describe the details of the technique used for each object program.

`schedule` and `schedule2` These two programs have four input parameters: three integers representing the number of jobs on the 1st, 2nd, and 3rd priorities, and an input file. To generate inputs we applied the following procedure:

1. Randomly choose the number of input parameters, with the following probabilities: 99% for 3, 0.8% for 2, 0.1% for 1, and 0.1% for 0.

2. Randomly choose whether the total number of jobs should be 0, with the following probabilities: 8% for yes, and 92% for no.

3. If "no" is chosen for 2, choose the value of each input parameter, with a 20% probability of selecting 0, and and 80% probability of selecting each of 1 through 10.

4. Randomly select an input file from the 2151 files in the existing test pool.

`printtokens` and `printtokens2` These two programs take a single file as input. To generate inputs we applied the following procedure:

1. Decide the validity of the input, with a 0.5% probability of generating "two input files"; 0.5% for "one non-existing file"; 99% for "one existing file".
2. If one existing file is chosen as input, randomly choose whether the input file is either an original file in the test pool (50% probability) or a file combining two original files in the test pool (50%).
3. If the input file is an original file in the test pool, randomly select one file from the test pool (the number of files in the pool: 4071).
4. If the input file is composed of two original files, randomly select two different files from the test pool and concatenate them.

`replace` There are three input parameters for `replace`: strings representing the regular expression (RE) and the replacing string (RS), respectively, and the input file (F). We used the following procedure to generate them:

1. Randomly choose the number of input parameters according to the following probabilities: 97% for 3 (RE, RS, and F), 2.7% for 2 (RE and F), and 0.3% for 1 (F only).
2. For each parameter to be generated, randomly choose an existing input from the existing test pool and extract the relevant input values.

`tcas` There are twelve input parameters for `tcas`, each of which is an integer. We used the following procedure to generate inputs:

1. Randomly choose whether the input is from the test pool (with probability 50%) or randomly combined based on the parameters from inputs in the test pool (50%).
2. If the input is from the test pool, randomly select one test case from the test pool (the number of test cases in the pool: 1608).
3. If the input is randomly combined based on the parameters in the test pool, for each input value, select a test case in the pool, extract the input parameter from that pool item, and then combine the selected parameters into a new input.

`totinfo` The input for `totinfo` is one file. To generate the input file, we used the following procedure:

1. Randomly choose whether the input file is a file in the original test pool (with probability 50%) or a file combining two original files in the test pool (50%).
2. If the file is to be from the original test pool, randomly select one.
3. If the input file is composed from two files from the test pool, randomly select two different files from the pool and concatenate them.

Table 5: Independent and Dependent Categories for `grep`

Independent Category	**Dependent Category**
NormalChar	Bracket
WordSymbol	Iteration
DigitSymbol	Parentheses
SpaceSymbol	Line
NamedSymbol	Word
AnyChar	Edge
Range	Combine

Table 6: Examples of Test Cases Involving Independent Categories for `grep`

Category	**Possible Choice**	**Test Case**
NormalChar	NormalAlNum	A
WordSymbol	YesWord	\w
DigitSymbol	NoDigit	\D
SpaceSymbol	NoSpace	\S
NamedSymbol	ALPHA	[:ALPHA:]
AnyChar	Dot	.
Range	NumRange	[1-9]

4.5.2 Generation for Unix utilities

Wong et al. [47] developed a random test case generator for the Unix utilities. We used the same generator in our study. Though these utilities have various types of inputs (such as strings, data files, etc.), each of them actually reads in a list of arguments via the command prompt. A random test case can be generated by constructing a list of random arguments. Full details can be found in Wong et al. [47].

4.5.3 Generation for `grep`

For `grep`, we used a generator that was itself based on the categories and choices devised for ART selection.

We first divided the categories into two groups – the independent categories and the dependent categories. The independent group includes all categories that contain elements that can form a regular expression usable as a test case on their own (for instance, a single literal forms a legitimate `grep` regular expression). Dependent categories are those that, without the presence of data that fall into other categories, cannot form an input. Categories 1 through 7 (described fully in Appendix A) were classified as independent categories and the rest were classified as dependent. The dependent and independent categories are listed in Table 5.

We next systematically generated random candidate test cases, which were collectively guaranteed to cover each category and choice. For independent categories, this is straightforward: for instance, the "NormalChar" category has a choice "NormalAlNum". To generate a test case that has this choice, a single character from the set containing all letters and digits is generated randomly. For dependent categories, elements from the dependent categories must be combined with an element from an independent category (based on the constraints from the specification), constructed as discussed above, to make a complete, valid test case; for instance, the dependent category "Iteration" could be combined with a "NormalAlNum" character to form a regular expression. Examples of values for each independent category are shown in Table 6, and example combinations of categories including dependent categories are shown in Table 7.

Category "Combine" is a special case: it involves either concatenation or selection between alternatives. When this category is selected, a choice (concatenation or selection) is determined. The procedure described in the paragraphs above is then used to generate the two subsidiary elements that are finally combined in the test case. For example, two subsidiary elements "a" and "b" combined based on concatenation are "ab";

Table 7: Examples of Test Cases Involving Dependent Categories for `grep`. (*Dependent Categories and Their Associated Choices are Italicized*)

Combination of Categories	**Possible Combination of Choices**	**Example of Test Cases**
Bracket;NormalChar	*NormalBracket*;NormalAlNum	[A]
Iteration;Range	*Star*;UpcaseRange	[A-Z]*
Parentheses;NormalChar;DigitSymbol	*NormParen*;NormalAlNum;YesDigit	(A\d)
Line;WordSymbol	*BegLine*;YesWord	ˆ\w
Word;DigitSymbol	*EndWord*;NoDigit	\D\>
Edge;Range	*YesEdgeBegEnd*;NumRange	\b[1-9]\b
Combine;*Iteration*;*Parentheses*; NormalChar;Range	*Concatenation*;*Plus*;*NormParen*; NormalAlNum;NumRange	(A[0-9])+

and when combined based on alternation are "a|b".

Note that our test generator does not randomly sample from the entire input domain of `grep`. Instead, only a small subset of the input space is sampled from, as our purpose is to test the regular expression analyzer of grep. We further filtered the randomly generated pool to remove duplicate entries. The final pool contained 171,634 elements.

4.6 Experiment environment

All experiment runs were conducted on a cluster of 64-bit Intel Clovertown systems running CentOS 5. The large number of experimental trials required to collect data with sufficiently narrow confidence intervals consumed a great deal of computer time, making the use of the cluster essential to obtain results in a reasonable time. The object programs were written in standard C and did not require any modifications to compile and run on the nodes in the cluster.

4.7 Experiment design and analysis strategy

4.7.1 Number of candidates

The parameter k — the size of the candidate set used by FSCS-ART — is at the discretion of the tester. Previous work [16] has shown — at least for numeric programs — that failure-detection effectiveness improves as k is increased up to about 10, and then does not improve much further. Therefore, our experiment runs were all conducted with k set to 10.

4.7.2 F-measure

For an experiment run, a test case was generated (using RT or ART) and executed on both an unmodified version of the object program under test and the fault of interest. The number of test cases needed to detect a failure (known as the *F-count*) as indicated by a difference between the outputs of the faulty and original versions of the program, was recorded. For each fault, 2000 runs were performed for the RT, *ARTmif*, and *ARTsum* strategies. This large number of runs is desirable due to the statistical properties of the F-count. Typically, the population distribution of the F-count is geometric for RT and near-geometric for most ART variants [13]; therefore, the standard deviation is very high and obtaining acceptably narrow confidence intervals requires large samples. For each object program and technique, the F-measure was calculated by averaging F-counts across all the experiment runs.

To simplify presentation of the data, we calculated the ratio of the F-measure for each ART technique compared to the F-measure for RT for each fault. We refer to this as the *F-ratio*. The F-measures for

RT on different faults in the same object program vary by orders of magnitude, and these F-measures are not normally distributed. Therefore, to concisely summarize the differences in performance between the methods, we presented the relative performance using RT as the baseline, which is what the F-ratio is.

4.7.3 P-measure

Raw data to calculate P-measures was recorded in the same experiments. For each fault in each object program, 2,000 runs of 1,000 test cases were conducted, and failures (as indicated by differences between the outputs of the faulty and original versions of the program) were recorded.

4.8 Threats to validity

4.8.1 Internal validity

There are relatively few threats to the internal validity of this study. Experimental conditions were identical in all respects, except for the independent variable of the testing strategy, for each experiment treatment. Where testing strategies were concerned, one possible issue involves our implementations of techniques, or of testing oracles. It is possible that these implementations contain errors; however, the amount of programming required to implement each specific testing strategy was small, and the implementations were checked by various authors. The oracle, likewise, is computationally trivial, involving a simple string comparison. Furthermore, the implementations were all created by the same individuals, helping ensure that differences in programming abilities would not bias results. In terms of the execution time comparisons, given that the authors implemented both *ARTsum* and *ARTmif*, it is possible that the implementation of one was more optimized than the other, affecting their relative computational overhead. The implementations were reviewed for obvious inefficiencies and none were found.

4.8.2 External validity

The most obvious threat to external validity in this study is that we consider only 14 object programs. We cannot say whether the techniques we study will exhibit similar results on other software systems without further study. The selection of appropriate categories and choices is a subjective process relying on the knowledge and experience of the testers (which were the authors). Our study considered only one set of categories and choices for each object program. We cannot be sure that other testers, presented with the same software under test, would choose a set of categories and choices that would achieve similar results. The particular faults we used, almost all of which were the result of fault seeding by programmers or randomly applying mutation operators, may not be representative of real faults and fault distributions encountered in industrial practice. A further threat to external validity involved considering the detection of a single fault at a time. There is no reason why the same intuition that explains why ART detects single faults more quickly than RT should not also hold when multiple faults are present; however, this needs to be assessed empirically.

4.8.3 Construct validity

As discussed in Section 4.3.2, none of the metrics used here give a full picture of the fault-finding effectiveness of a testing technique. They all measure failure-detection capability, but do not directly measure the ability of a technique to detect multiple faults in the software under test.

4.8.4 Conclusion validity

Given the large number of experiment runs conducted for each fault, we believe that our tests had sufficient statistical power to draw conclusions about the F-measures and P-measures of each testing strategy at the individual fault level. However, the use of weaker nonparametric tests for statistical significance has limited our ability to show significant differences where they may exist.

5 Data and analysis

5.1 RQ1: Failure-detection effectiveness

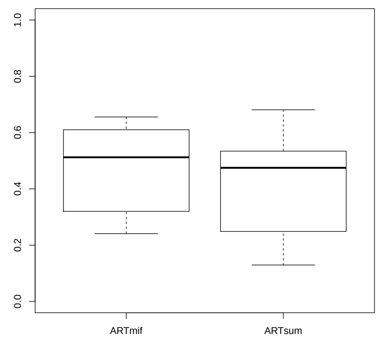
We present failure-detection effectiveness results in terms of the two measures considered, in turn.

5.1.1 F-measure

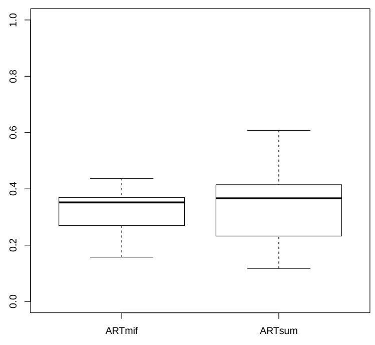
For each object program, we present a boxplot and a table summarizing the results. The boxplot for each program (Figure 1) displays the range of F-ratios (that is, the ratio of the F-measure of ART compared to that of RT) for each of the two ART strategies, for all faulty versions of the object program under test. Smaller F-ratios indicate better performance for ART, and an F-ratio smaller than 1 indicates that ART outperformed RT. The lower and upper sides of the box denote the lower and upper quartiles respectively. The line inside the box indicates the median F-ratio. The whiskers represent the smallest and largest datums within a range $\pm 1.58 \times IQR$, where IQR is the interquartile range. Small circles represent outliers outside this range. Full results are presented in Appendix B.

Table 8 presents direct pairwise comparisons of the F-measures of the two ART techniques for each object program. Each cell in the table represents the number of faults on which the technique listed *above* the cell outperformed the technique listed to the *left*. For instance, in Table 8(a), the entry in the top-right hand corner of the table shows that *ARTmif* had a *smaller* F-measure than RT on all 11 of the faults. Similarly, the entry in the bottom left-hand corner of the table shows that RT outperformed *ARTsum* on 0 of the 11 faults.

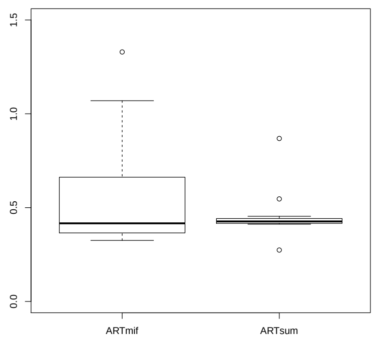
Because the numbers of faults for each object program was small and their F-measures were not normally distributed, conventional parametric hypothesis testing (such as T-tests or ANOVAs) is not suitable for analyzing our results. Therefore, to test whether the performance differences were statistically significant, we conducted a Friedman test for each technique. The Friedman test examines whether the rankings of the techniques across trials (faults, in this case) are as would be expected if they were sampled from the same population. To use an overall α (probability of a non-significant difference being incorrectly classified as significant) of 0.05 across the entire paper, we used the Holm-Bonferroni method [27] to determine which programs exhibited statistically significant differences. Note that in the nonparametric statistical test, it is irrelevant whether we used the F-ratio or the unadjusted F-measure, as the ranking is unaffected by the transformation. On all programs except `schedule`, the testing techniques exhibited failure-detection results that were statistically significantly different. To determine which techniques performed significantly differently for each fault, post-hoc comparisons using the Wilcoxon signed-rank test with corrections for multiplicity were used. A **bold** number in the tables indicates that the differences between techniques was statistically significant. For instance, the fact that *ARTmif* outperformed RT on 17 of the 20 `grep` faults is statistically significant, whereas the fact that *ARTsum* outperformed *ARTmif* on 11 of the 20 `grep` faults is not.



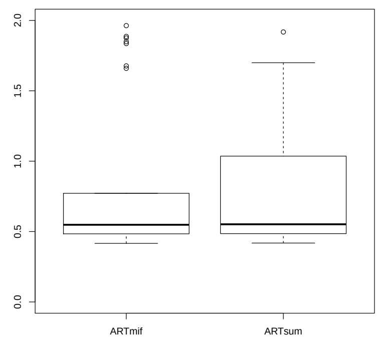
(a) `cal`



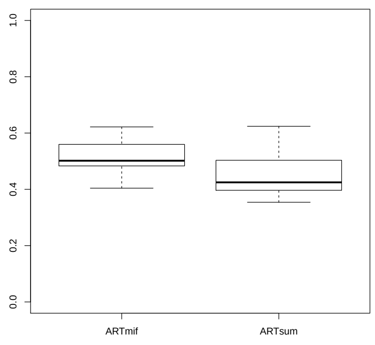
(b) `comm`



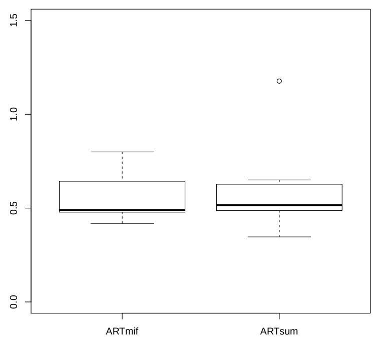
(c) `grep`



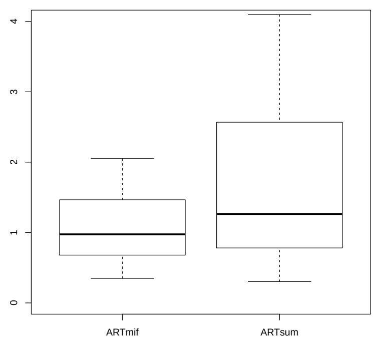
(d) `look`



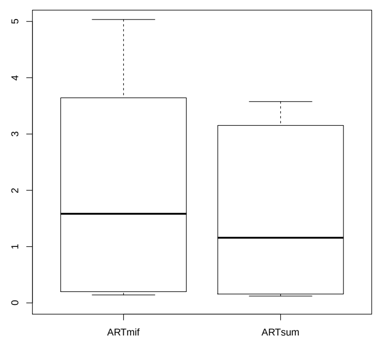
(e) `printtokens`



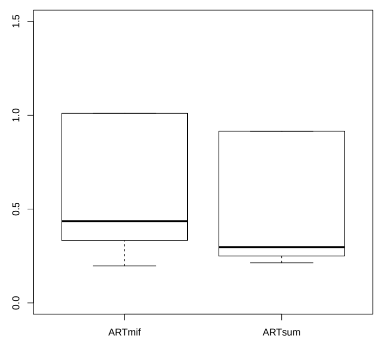
(f) `printtokens2`



(g) `replace`

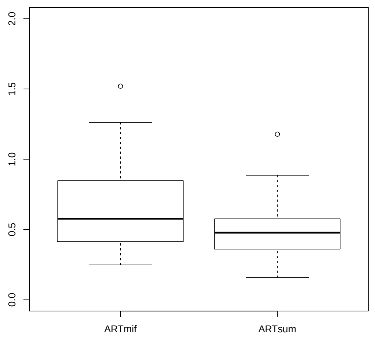


(h) `schedule`

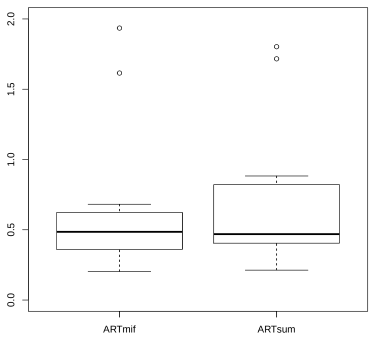


(i) `schedule2`

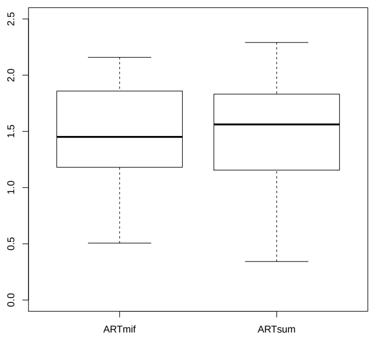
Figure 1: Boxplots of F-ratio distributions for ART techniques for each object program



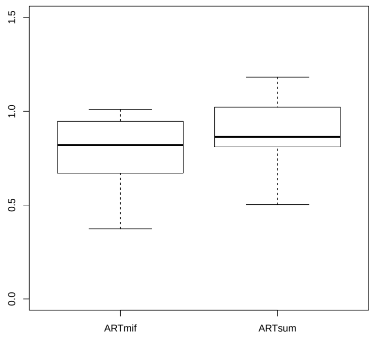
(j) `sort`



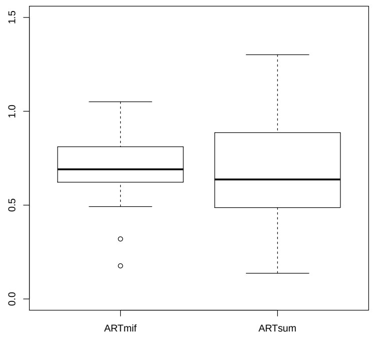
(k) `spline`



(l) `tcas`



(m) `totinfo`



(n) `uniq`

Figure 1: Boxplots of F-ratio distributions for ART techniques for each object program (continued)

Table 8: Number of Faults for Which the Technique on the Top Row Has a Lower (Better) F-measure Than the Technique on the Left

(a) `cal`

	RT	ARTmif	ARTsum
RT	N/A	**11**	**11**
ARTmif	**0**	N/A	8
ARTsum	**0**	3	N/A

(b) `comm`

	RT	ARTmif	ARTsum
RT	N/A	**27**	**27**
ARTmif	**0**	N/A	13
ARTsum	**0**	14	N/A

(c) `grep`

	RT	ARTmif	ARTsum
RT	N/A	**17**	**20**
ARTmif	**3**	N/A	11
ARTsum	**0**	9	N/A

(d) `look`

	RT	ARTmif	ARTsum
RT	N/A	**22**	**21**
ARTmif	**7**	N/A	13
ARTsum	**8**	16	N/A

(e) `printtokens`

	RT	ARTmif	ARTsum
RT	N/A	**7**	**7**
ARTmif	**0**	N/A	5
ARTsum	**0**	2	N/A

(f) `printtokens2`

	RT	ARTmif	ARTsum
RT	N/A	**10**	**9**
ARTmif	**0**	N/A	3
ARTsum	**1**	7	N/A

(g) `replace`

	RT	ARTmif	ARTsum
RT	N/A	18	14
ARTmif	13	N/A	**6**
ARTsum	17	**25**	N/A

(h) `schedule`

	RT	ARTmif	ARTsum
RT	N/A	4	4
ARTmif	5	N/A	8
ARTsum	5	1	N/A

(i) `schedule2`

	RT	ARTmif	ARTsum
RT	N/A	6	**8**
ARTmif	3	N/A	8
ARTsum	**1**	1	N/A

(j) `sort`

	RT	ARTmif	ARTsum
RT	N/A	**41**	**47**
ARTmif	**7**	N/A	31
ARTsum	**1**	17	N/A

(k) `spline`

	RT	ARTmif	ARTsum
RT	N/A	**14**	**13**
ARTmif	**2**	N/A	6
ARTsum	**3**	10	N/A

(l) `tcas`

	RT	ARTmif	ARTsum
RT	N/A	**8**	**7**
ARTmif	**33**	N/A	21
ARTsum	**34**	20	N/A

(m) `totinfo`

	RT	ARTmif	ARTsum
RT	N/A	**18**	17
ARTmif	**5**	N/A	**3**
ARTsum	6	**20**	N/A

(n) `uniq`

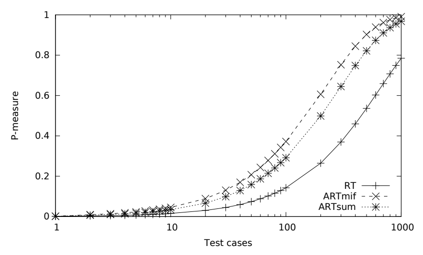
	RT	ARTmif	ARTsum
RT	N/A	**28**	**24**
ARTmif	**1**	N/A	12
ARTsum	**5**	17	N/A

ARTsum significantly outperformed RT in terms of the F-measure on 10 of the 14 object programs. For three of the remaining four programs, `replace`, `schedule`, and `totinfo`, there was no statistically significant difference in the performance of *ARTsum* and RT. On only one program, `tcas`, did RT significantly outperform *ARTsum*.

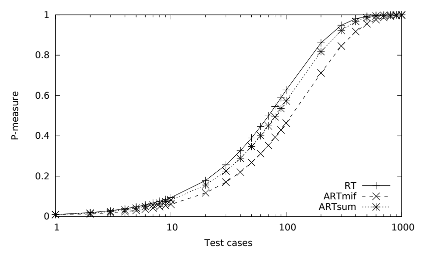
ARTmif displayed similar but not identical performances. *ARTmif* outperformed RT on 10 of the 14 object programs. There were no statistically significant differences in performance on three programs, `replace`, `schedule`, and `schedule2`. Again, only on `tcas`, did RT outperform *ARTmif*. The magnitude of the performance improvement varied between the programs. The best case, seen on a number of object programs, involved the median F-measure for the two ART techniques being more than 50% lower than for RT. The differences in effectiveness between *ARTsum* and *ARTmif* were small. There was a slight preponderance of results indicating that *ARTsum* may marginally outperform *ARTmif*, but these did generally not achieve statistical significance.

5.1.2 P-measure

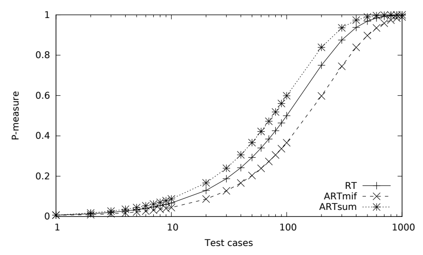
Presenting and comparing results for the P-measure when there are many faults with different failure rates is challenging. Nevertheless, we present both descriptive and inferential statistics comparing the three methods



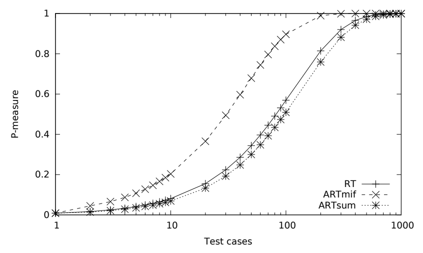
(a) `grep` fault #6



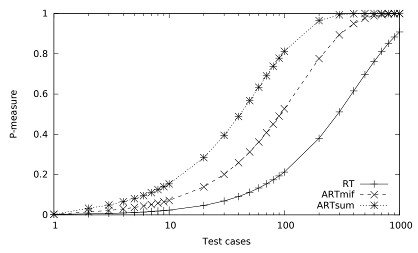
(b) `schedule` fault #7



(c) `sort` fault #43



(d) `totinfo` fault #2



(e) `uniq` fault #20

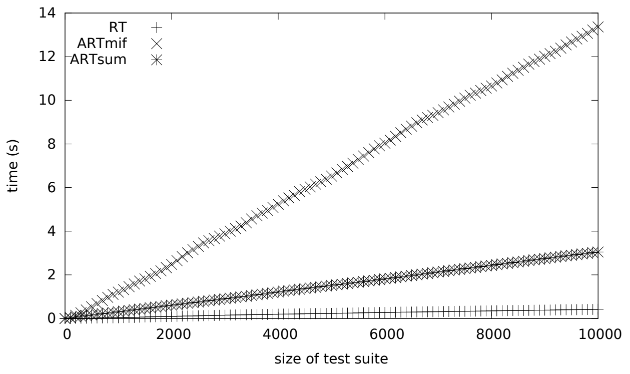
Figure 2: P-measure by technique for selected faults.

using the P-measure. We first present graphs (in Figure 2) showing the P-measure for the three techniques under test for a few selected faults on which the techniques achieved different relative rankings, to illustrate trends in the results. We used logarithmic scales on the x-axis in Figure 2 to represent the number of test cases.

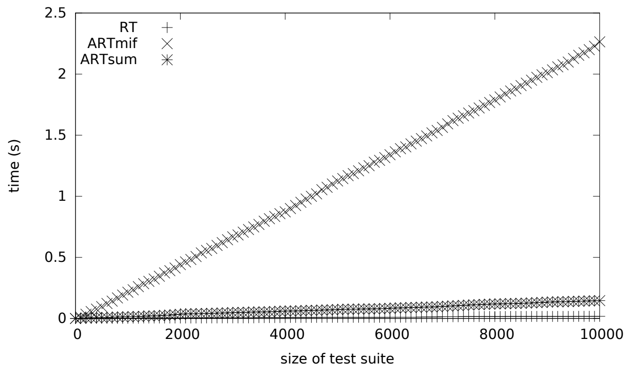
The values of P-measures depend not only on the program under test and the testing strategy used to generate the test suite, but also on the size of the test suite. Thus, simply examining individual P-measures on some specific test suite sizes may not provide a complete picture. Therefore, to enable the statistical analysis of the P-measure results, we performed some additional preliminary calculations to aggregate results from each fault. We used the aggregation of P-measures across various test suite sizes, as measured by the total area under the P-measure graph (the "P-measure area", which we abbreviate as "PMA"), to compare the effectiveness of testing techniques. For a given fault, if PMA is larger for a technique α than for another technique β, the performance of α is interpreted to be superior to β. In fact, Figure 2 clearly shows that for the selected faults, if one technique has a higher P-measure and is therefore more effective than another for a given test suite size, then it will be equal to or superior than the other for other test suite sizes. We found this pattern to hold for all faults examined.

We calculated the PMA for all faults in all programs and ranked the techniques for each fault in each program. Table 9 shows that the ranking of each technique by PMA was almost identical to that achieved using the F-measure, with a small difference in the case of the `replace` program. That is, in virtually all cases, if one technique demonstrated a superior (lower) F-measure than another for a specific fault in a specific program, that technique would have a superior (higher) PMA.

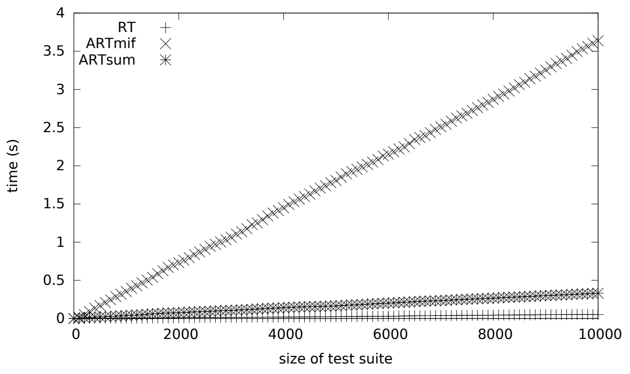
Friedman tests (applying a Holm-Bonferroni correction across all hypothesis tests, for both P-measures and F-measures) therefore showed that the differences that were significant for the P-measures and F-measures were identical.



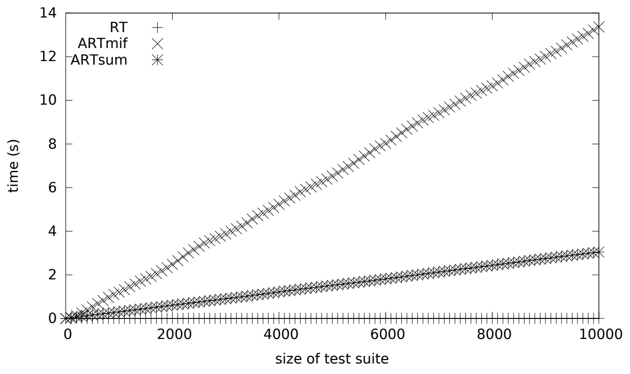
(a) **printtokens**



(b) **replace**



(c) **schedule**



(d) **grep**

Figure 3: Time required to generate test suites

Table 9: Number of Faults for Which the the Technique on the Top Row has a Higher (Better) PMA Than the Technique on the Left

(a) `cal`

	RT	ARTmif	ARTsum
RT	N/A	**11**	**11**
ARTmif	**0**	N/A	8
ARTsum	**0**	3	N/A

(b) `comm`

	RT	ARTmif	ARTsum
RT	N/A	**27**	**27**
ARTmif	**0**	N/A	13
ARTsum	**0**	14	N/A

(c) `grep`

	RT	ARTmif	ARTsum
RT	N/A	**17**	**20**
ARTmif	**3**	N/A	11
ARTsum	**0**	9	N/A

(d) `look`

	RT	ARTmif	ARTsum
RT	N/A	**22**	**21**
ARTmif	**7**	N/A	13
ARTsum	**8**	16	N/A

(e) `printtokens`

	RT	ARTmif	ARTsum
RT	N/A	**7**	**7**
ARTmif	**0**	N/A	5
ARTsum	**0**	2	N/A

(f) `printtokens2`

	RT	ARTmif	ARTsum
RT	N/A	**10**	**9**
ARTmif	**0**	N/A	3
ARTsum	**1**	7	N/A

(g) `replace`

	RT	ARTmif	ARTsum
RT	N/A	18	14
ARTmif	13	N/A	**5**
ARTsum	17	**26**	N/A

(h) `schedule`

	RT	ARTmif	ARTsum
RT	N/A	4	4
ARTmif	5	N/A	8
ARTsum	5	1	N/A

(i) `schedule2`

	RT	ARTmif	ARTsum
RT	N/A	6	**8**
ARTmif	3	N/A	8
ARTsum	**1**	1	N/A

(j) `sort`

	RT	ARTmif	ARTsum
RT	N/A	**41**	**47**
ARTmif	**7**	N/A	31
ARTsum	**1**	17	N/A

(k) `spline`

	RT	ARTmif	ARTsum
RT	N/A	**14**	**13**
ARTmif	**2**	N/A	6
ARTsum	**3**	10	N/A

(l) `tcas`

	RT	ARTmif	ARTsum
RT	N/A	**8**	**7**
ARTmif	**33**	N/A	21
ARTsum	**34**	20	N/A

(m) `totinfo`

	RT	ARTmif	ARTsum
RT	N/A	**18**	17
ARTmif	**5**	N/A	**3**
ARTsum	6	**20**	N/A

(n) `uniq`

	RT	ARTmif	ARTsum
RT	N/A	**28**	**24**
ARTmif	**1**	N/A	12
ARTsum	**5**	17	N/A

5.2 RQ2: Test suite generation time

Figure 3 shows the execution time required to generate test suites of various sizes for four of the 14 object programs using the three different techniques. Consistent with our theoretical analysis, all three techniques require time linear in the size of the generated test suite. The constant factors for *ARTsum*, however, are consistently lower than those for *ARTmif*.

Table 10 shows these constant factors by indicating the time required to generate 10,000 test cases using RT for all the input generators. Note that `schedule` and `schedule2` share the same input generator, as do `printtokens` and `printtokens2`, so only 12 input generators are listed. We also compare the relative time taken using the three different methods for each input generator. As can be seen, there is wide variation in the relative time costs of input generation depending on the program. The generation times using the *ARTsum* algorithm are within a range of 2.0 to 8.1 times that of RT, whereas *ARTmif* takes 4.7 to 2102.0 times longer than RT and takes 1.9 to 361.3 times longer than *ARTsum*.

Table 10: Comparison of Time Required to Generate 10,000 Random Inputs Using RT, *ARTmif* and *ARTsum*

	Generation time (s)			Relative generation time		
	RT	*ARTmif*	*ARTsum*	*ARTmif*/RT	*ARTsum*/RT	*ARTmif*/*ARTsum*
`cal`	0.032	0.15	0.065	4.7	2.0	2.3
`comm`	0.034	0.222	0.117	6.5	3.4	1.9
`grep`	0.011	23.122	0.064	2102.0	5.8	361.3
`look`	0.023	0.368	0.105	16.0	4.6	3.5
`printtokens`	0.421	13.365	3.046	31.7	7.2	4.4
`replace`	0.018	2.264	0.146	125.8	8.1	15.5
`schedule`	0.052	3.637	0.331	69.9	6.4	11.0
`spline`	0.022	0.622	0.116	28.3	5.3	5.4
`sort`	0.011	2.717	0.072	247.0	6.5	37.7
`tcas`	0.018	2.306	0.104	128.1	5.8	22.2
`totinfo`	0.045	1.464	0.3	32.5	6.7	4.9
`uniq`	0.025	0.839	0.1	33.6	4.0	8.4

6 Discussion

Overall, it is clear that in the cases we studied, *ARTsum* and *ARTmif* were each more effective than RT, as measured by both the F-measure and P-measure, on a majority of the object programs considered. Also, *ARTsum* had a much lower selection overhead than *ARTmif*, and its overhead was close to that of RT.

We examined the cases in which the ART techniques were not significantly more effective in terms of fault-detection than RT. This occurred on the object programs `replace`, `schedule`, and `totinfo`, where differences between *ARTsum* and RT were not statistically significant, and on `tcas`, where RT was more effective than ART. Our investigation revealed an interesting pattern related to the distribution of failure-revealing inputs in test frames for the different faulty versions of `replace`. We first examined the failure rate within failure-revealing "test frames" – that is, the subsets of the test pool that shared the same categories and choices, and contained at least one failure-causing input. We hypothesized that for faults on which ART performed poorly, the failure rate within the failure-revealing test frame would be lower. There did not appear to be any such systematic effect, so we then examined the distribution of the test frames containing failure-causing input in terms of their average "distance". We found that the "distance" between frames containing failures was *higher* for faults on which ART outperformed RT, and *lower* when RT outperformed ART. The faulty versions of `schedule` and `tcas`, on which ART exhibited comparatively poor performance, have similar distributions of failure-revealing inputs in test frames.

One potential explanation for this is that when a test case is executed in a test frame that contains a failure, but that test case does not reveal a failure, this reduces the chances of selecting nearby test cases. Thus, a technique will perform better if there are other failure-revealing test cases located far away, rather than close by. This suggests that our distance measure and selection criteria could still be improved. In the short-term, one obvious point is that to maximize testing effectiveness, non-homogeneous test frames should be avoided, and the best way to do this is to have fine-grained test frames that correspond to distinct program functionalities. In short, testers may be best advised to use a larger number of categories and choices to make as fine-grained a difference measure as possible, and choose them carefully to ensure that they align with the functionality of the program under test.

Given that `grep` was our largest program, despite the overall effectiveness of ART, it is worth considering effectiveness results on that program in some detail. For `grep`, *ARTmif* was inferior to RT for three faults: faults 4, 15, and 17 as listed in Table 25. We examined these cases in more detail, and found behavior similar to that occurring in the cases of `replace` and `schedule`. However, we also noted an additional factor: the inferior effectiveness was related to the non-uniform distribution of the test cases selected. Such biases have been observed in previous studies of ART. In fact, biases are almost inevitable; it is difficult to achieve an effective spreading of inputs without inducing *some* bias towards certain inputs. *ARTmif* preferentially selected inputs that had a large number of choices. For each of the three faults of `grep` on which *ARTmif* did not outperform RT, most of the *failure-causing inputs* had a very small number of choices. Hence, these failure-causing inputs were less likely to be selected by *ARTmif* than by random chance. This is related to the granularity problems of the distance measure as discussed in Section 3.2, but is not strictly the same.

We have shown that *ARTsum*, using our proposed distance measure, significantly outperformed RT on 10 of our 14 object programs. Our results showed that *ARTsum* and *ARTmif* had comparable performance, and that *ARTsum* slightly outperformed *ARTmif* particularly when there were a small number of categories. However, this is consistent with our view that the max-sum criterion handles a coarse distance measure better than the max-min criterion.

We have clearly shown that while both *ARTsum* and *ARTmif* are linear-time algorithms, in practice, *ARTsum* can incur a much smaller selection overhead than *ARTmif*. Therefore, given that *ARTsum* and *ARTmif* have comparable failure-detection effectiveness, the lower overhead suggests that *ARTsum* should be considerably more cost-effective overall.

Despite the satisfactory effectiveness demonstrated by the "forgetting" strategies employed in *ARTmif* in this study prior studies, the settings of their parameters seem to be arbitrary and are not rigorously justified. This arbitrariness does not occur for *ARTsum*, which in our view is a further point in favor its use of in preference to *ARTmif*.

7 Related Work

The extension of ART to non-numeric input domains has been of interest for some time. Ciupa et al. [21] demonstrated the application of ART to unit testing of object-oriented software. There are significant differences between their approach and ours. Ciupa et al.'s distance measure, which was specifically designed for unit testing of object-oriented software, is based on the structure of method inputs, and permits no tester discretion. Our distance measure, in contrast, allows testers to use their knowledge of the specification and/or the program structure to specify appropriate categories and choices. It is not restricted to object-oriented languages, and is applicable beyond the stage of unit testing. Ciupa et al.'s implementation uses FSCS-ART as the test case selection technique. This technique's quadratic selection overhead implies that overall, ART might not actually be cost effective compared to random testing. Our technique, in contrast, takes advantage of the properties of our distance measure to achieve linear-time selection overhead, addressing these cost-effectiveness issues.

There have been a number of attempts to reduce the selection overhead of ART, even before Arcuri and Briand [3] drew attention to the implications of this for the practical use of ART. Mayer and Schneckenburger [33] examined some earlier low-overhead ART algorithms. More recently, Shahbazi et al. [43] devised the RBCVT-Fast technique, a linear-time ART algorithm for d-dimensional real input domains. Our work is complementary to theirs in that it can be applied to non-numeric input domains.

Chen and Merkel [18] observed that if no assumptions are made about the location of "failure regions" within an input domain, the F-measure of any testing technique will (on average) be no lower than half that of RT. For a few object programs considered in our study, the average performance of the two ART algorithms was *better* than this effectiveness bound. This occurred because the algorithms proposed in this article implicitly make assumptions about failure locations, and tend to test some regions of programs more frequently than others due to variations in the size of the regions defined by the different category/choice combinations. While we made no attempt to specifically take advantage of these properties, this did mean that for some cases where the preference happened to align with the failure location the performance improvements did exceed the bound, while in some other cases, inferior performance resulted. However, our empirical work indicates that the random nature of ART ensures that the worst-case performance is tolerable and occurs relatively infrequently.

The code coverage characteristics of ART have been examined in detail in Chen et al. [11], for numerical software. An extension of this study to non-numeric software has been conducted using our approach involving categories and choices [12], the empirical work for which was conducted after the start of this project. Results of these two studies showed that in addition to higher failure-detection effectiveness, ART can also achieve higher code coverage than RT.

8 Conclusion

ART was proposed to enhance the failure-detection effectiveness of RT. In this work we have presented a linear-order ART algorithm, *ARTsum*, that makes use of a novel distance measure, and takes advantage of the properties of this distance measure to achieve a linear-order algorithm using the max-sum selection criterion. Our work is complementary to the new RBCVT-Fast algorithm [43], which is an innovative linear-order ART algorithm for numeric inputs.

We conducted an empirical study using a total of fourteen programs, comparing our *ARTsum* algorithm with RT and a baseline ART technique using the max-min criterion and the technique of "forgetting" to reduce selection overhead, namely *ARTmif*. These programs came from three sources: the SIR repository, several Unix utilities, and the larger GNU `grep` utility. Each of the ART algorithms significantly outperformed RT with respect to the F-measure for 10 of the 14 object programs, was significantly outperformed by RT for only one program, and had comparable performance with RT for the remaining three programs. An identical pattern was observed for the P-measure. Furthermore, the selection overhead of *ARTsum* was quite close to that of RT, and far lower than that of *ARTmif*.

This work has demonstrated a feasible and computationally efficient scheme of linear order for applying ART to programs with non-numeric input types. We have shown that ART can be used to efficiently perform debug testing on several software programs with non-numeric input domains. In doing so, we address the cost-effectiveness issues raised by Arcuri and Briand [3], permitting both practical use and further investigation of the behavior of ART for programs with very low failure rates. In the present paper, the emphasis was on the presentation of a novel linear-order ART algorithm and the demonstration of its practicality, so the question of effectiveness with very low failure rates was not studied. Obviously, further work is now called for to examine this question. In our empirical study, the number of categories, and the number of choices per category, were small. This results in the granularity of our distance measure being quite coarse, particularly for the case of `grep`. Consistent with our expectations, *ARTsum* was far less affected by this problem than *ARTmif*. Intuitively speaking, our algorithm should perform better at finer granularity of the proposed distance measure. However, further investigation of the relationship between category and choice selection, and algorithm performance, is desirable.

Of the threats to validity identified for this study, the most significant is the limitation related to the object programs studied. Clearly, more studies will be needed to determine whether our results generalize.

A further examination of the effectiveness of the technique in multiple-fault scenarios is also called for.

Random testing is of course not the only way to automatically generate test data; alternatives in this broad field include search-based testing [1] and combinatorial methods [36]. However, direct comparisons between RT/ART methods and these techniques is not a straightforward matter. Orso and Rothermel [37] noted that many studies of search-based software testing did not attempt to compare their methods with existing approaches. Arcuri and Briand [4] formally compared RT and combinatorial methods and found that the overall cost-effectiveness of the two methods depended on a number of factors, not least the computation overhead of combinatorial testing. Given the relatively low overheads of *ARTsum*, some of their conclusions are likely to apply to our methods also. Having said that, it would be worthwhile to systematically compare ART with other state-of-the-art testing techniques, and this is one important direction for the future work. We also note that it is generally accepted that no single testing method is suitable for finding all faults or classes of faults; and that a variety of techniques should be used to ensure a high-quality system. How best to combine random, combinatorial, search-based, and other automated and manual methods to achieve the best overall testing cost-effectiveness is also a a challenging but important area for future work.

The selection of appropriate categories and choices, particularly for `grep`, was not a straightforward task, and relied substantially on the expertise and experience of the tester. However, to apply RT to a non-numeric input domain, testers already need to perform some analysis of the input domain. One useful method for doing so is by identifying categories and choices, just as has been done for `grep` in this paper. If such an approach has been taken, the additional effort by testers to apply ART over RT is quite small. There has already been some research on techniques for identifying appropriate categories and choices for the category-partition method [19]. Applying these techniques to generate categories and choices for an ART distance measure is a promising avenue for future work.

There is much scope for more advanced distance measures to take better account of more information about the characteristics of the software under test, to better predict similarity in failure behavior of inputs. Within the general paradigm of categories and choices, there are many potential refinements that could be attempted. For instance, some categories may be more fundamental, and thus presumably more important to the failure behavior of the software. It would be beneficial for the distance measure to take this information into account, for instance through a weighting scheme. Furthermore, within categories, the present scheme considers all disjoint choices as equally dissimilar. A more sophisticated scheme may treat some choices as more dissimilar than others. There are many other possibilities, and indeed there may be conceptually different schemes that work well in particular domains. We believe that finding appropriate distance measures for specific domains will prompt much future research.

References

[1] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2010.

[2] P. E. Ammann and J. C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425, 1988.

[3] A. Arcuri and L. Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 20th International Symposium on Software Testing and Analysis*, ISSTA '11, pages 265–275, 2011.

[4] A. Arcuri and L. Briand. Formal analysis of the probability of interaction fault detection using random testing. *IEEE Transactions on Software Engineering*, 38(5):1088–1099, 2012.

[5] A. Arcuri, M. Z. Iqbal, and L. Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *Proceedings of the 22nd IFIP WG 6.1 International Conference on Testing Software and Systems*, ICTSS '10, pages 95–110, 2010.

[6] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering*, 38(2):258–277, 2012.

[7] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna. A genetic approach for random testing of database systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1243–1251, 2007.

[8] P. G. Bishop. The variation of software survival time for different operational input profiles (or why you can wait a long time for a big bug to fail). In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, FTCS-23, pages 98–107, 1993.

[9] P. Bourque and R. E. D. Farley, editors. *SWEBOK 3.0: Guide to the Software Engineering Body of Knowledge*. IEEE, 3 edition, 2014.

[10] K.-P. Chan, T. Y. Chen, and D. Towey. Forgetting test cases. In *Proceedings of the 30th Annual International Computer Software and Applications Conference*, volume 1 of *COMPSAC '06*, pages 485–494, 2006.

[11] T. Y. Chen, F.-C. Kuo, H. Liu, and W. E. Wong. Does adaptive random testing deliver a higher confidence than random testing? In *Proceedings of the 8th International Conference on Quality Software*, QSIC '08, pages 145–154, 2008.

[12] T. Y. Chen, F.-C. Kuo, H. Liu, and W. E. Wong. Code coverage of adaptive random testing. *IEEE Transactions on Reliability*, 62(1):226–237, 2013.

[13] T. Y. Chen, F.-C. Kuo, and R. Merkel. On the statistical properties of testing effectiveness measures. *Journal of Systems and Software*, 79(5):591–601, 2006.

[14] T. Y. Chen, F.-C. Kuo, R. Merkel, and S. P. Ng. Mirror adaptive random testing. *Information & Software Technology*, 46(15):1001–1010, 2004.

[15] T. Y. Chen, F.-C. Kuo, and Z. Zhou. On favourable conditions for adaptive random testing. *International Journal of Software Engineering and Knowledge Engineering*, 17(6):805–825, 2007.

[16] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *Proceedings of the 9th Asian Computing Science Conference*, pages 320–329, 2004.

[17] T. Y. Chen and R. Merkel. Quasi-random testing. *IEEE Transactions on Reliability*, 56(3):562–568, 2007.

[18] T. Y. Chen and R. Merkel. An upper bound on software testing effectiveness. *ACM Transactions on Software Engineering and Methodology*, 17(3):16:1–16:27, 2008.

[19] T. Y. Chen, P.-L. Poon, S.-F. Tang, and T. H. Tse. On the identification of categories and choices for specification-based test case generation. *Information & Software Technology*, 46(13):887–898, 2004.

[20] T. Y. Chen, T. H. Tse, and Y. T. Yu. Proportional sampling strategy: A compendium and some insights. *Journal of Systems and Software*, 58(1):65–81, 2001.

[21] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: Adaptive random testing for object-oriented software. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 71–80, 2008.

[22] Codenomicon. Heartbleed.com, 2014.

[23] M. E. Delamaro and J. C. Maldonado. Proteum – a tool for the assessment of test adequacy for C programs. In *Proceedings of the Conference on Performability in Computing Systems*, PCS '96, pages 79–95, 1996.

[24] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[25] P. G. Frankl, R. G. Hamlet, B. Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering*, 24(8):586–601, 1998.

[26] A. Groce, G. J. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 621–631, 2007.

[27] S. Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6:65–70, 1979.

[28] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, 1994.

[29] F.-C. Kuo. *On Adaptive Random Testing.* PhD thesis, Faculty of Information and Communication Technologies, Swinburne University of Technology, 2006.

[30] S. Lipner. The trustworthy computing security development lifecycle. In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 2–13, 2004.

[31] Y. Liu and H. Zhu. An experimental evaluation of the reliability of adaptive random testing methods. In *Proceedings of the 2nd International Conference on Secure System Integration and Reliability Improvement*, SSIRI '08, pages 24–31, 2008.

[32] J. Mayer. Lattice-based adaptive random testing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 333–336, 2005.

[33] J. Mayer and C. Schneckenburger. An empirical analysis and comparison of random testing techniques. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 105–114, 2006.

[34] R. Merkel. *Analysis and Enhancements of Adaptive Random Testing.* PhD thesis, School of Information Technology, Swinburne University of Technology, 2005.

[35] G. J. Myers. *The Art of Software Testing.* John Wiley and Sons, second edition, 2004. Revised and Updated by T. Badgett and T. M. Thomas with C. Sandler.

[36] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11:1–11:29, 2011.

[37] A. Orso and G. Rothermel. Software testing: A research travelogue (2000-2014). In *Proceedings of Future of Software Engineering*, FOSE '14, pages 117–132, 2014.

[38] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[39] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, 2007.

[40] T. G. Project. Gnu c library reference manual. `http://www.gnu.org/software/libc/manual/html_node/index.html`, 2001.

[41] T. G. Project. Grep home page. `http://www.gnu.org/software/grep`, 2006.

[42] J. Regehr. Random testing of interrupt-driven software. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 290–298, 2005.

[43] A. Shahbazi, A. F. Tappenden, and J. Miller. Centroidal voronoi tessellations — a new approach to random testing. *IEEE Transactions on Software Engineering*, 39(2):163–183, 2013.

[44] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov. Testing container classes: Random or systematic? In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software*, FASE'11/ETAPS'11, pages 262–277, 2011.

[45] A. F. Tappenden and J. Miller. A novel evolutionary approach for adaptive random testing. *IEEE Transactions on Reliability*, 58(4):619–633, 2009.

[46] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6(3):247–257, 1980.

[47] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. *Software: Practice and Experience*, 28(4):347–369, 1998.

[48] T. Yoshikawa, K. Shimura, and T. Ozawa. Random program generator for Java JIT compiler test system. In *Proceedings of the 3rd International Conference on Quality Software*, QSIC '03, pages 20–24, 2003.

A Full category-choice description

Note that our categories and choices are mutually exclusive in terms of the inputs that our test case generator is able to actually produce. If the entire input domain of `grep` was to be considered, several categories in our category-choice definition (for instance, the category NormalChar) would then contain choices which are not mutually exclusive. If necessary, this issue can be resolved by defining an additional choice in each such category, representing the intersection of the two existing choices. For instance, for our category "NormalChar" in Table 13, a "both" choice could be defined. This choice applies to the situation where both the two existing choices "NormalAlNum" and "NormalPunct", would otherwise exist.

Table 11: Definition of categories and choices for `cal`

#	**Category**	**Choice**
1	number of parameters	0
		1
		2
2	month	< 1 or > 12
		≥ 1 or ≤ 12
3	year	< 1 or > 9999
		leap year
		non-leap year

Table 12: Definition of categories and choices for `comm`

#	**Category**	**Choice**
1	number of parameters	< 3
		3
		> 3
2	option 1	exist
		not exist
3	option 2	exist
		not exist
4	option 3	exist
		not exist
5	bad option	exist
		not exist
6	file 1	not exist
		contents sorted
		contents unsorted
7	file 2	not exist
		contents sorted
		contents unsorted
8	common lines	exist
		not exist

Table 13: Definition of categories and choices for `grep`

#	**Category**	**Choice**
1	NormalChar - presence of any literal character	NormalAlNum - presence of any alphabetic or numerical literal (for instance "A", "z", or "5")
		NormalPunct - presence of any punctuation character (such as ":")
2	WordSymbol - presence of "word" or "non-word" metacharacters	YesWord - "\w" present
		NoWord - "\W" present
3	DigitSymbol - presence of "digit" or "non-digit" metacharacters	YesDigit - "\d" present
		NoDigit - "\D" present
4	SpaceSymbol - presence of any "whitespace" or "non-space" metacharacters	YesSpace - "\s" present
		NoSpace - "\S" present

Table 13 Definition of categories and choices for `grep` (continued)

#	**Category**	**Choice**
5	NamedSymbol - presence of a symbol from a character group	ALPHA - presence of [:ALPHA:]
		UPPER - presence of [:UPPER:]
		LOWER - presence of [:LOWER:]
		DIGIT - presence of [:DIGIT:]
		XDIGIT - presence of [:XDIGIT:]
		SPACE - presence of [:SPACE:]
		PUNCT - presence of [:PUNCT:]
		ALNUM - presence of [:ALNUM:]
		PRINT - presence of [:PRINT:]
		GRAPH - presence of [:GRAPH:]
		CNTRL - presence of [:CNTRL:]
		BLANK - presence of [:BLANK:]
6	AnyChar - presence of the "." metacharacter (matches any character)	Dot - dot (".") present
7	Range - presence of a pattern representing a character range	NumRange - number range present (for example "[1-7]")
		UpcaseRange - uppercase letter range present (for example "[C-G]")
		LowcaseRange - lowercase letter range present (such as "[s-w]")
8	Bracket - presence of patterns encompassed by [] or [ˆ]	NormalBracket - "[]" pattern present
		CaretBracket - [ˆ] pattern present
9	Iteration - presence of patterns that contain iterator symbols	Qmark - presence of the question mark metacharacter ("?"), which matches 0 or 1 iteration
		Star - presence of the star metacharacter ("*"), matching zero or more iterations
		Plus - presence of the plus metacharacter("+"), matching one or more iterations
		Repminmax - presence of min-max repetition form: for example, "{2, 3}" matches lines containing "aa" or "aaa"
10	Parentheses - used to group patterns for repetition, also "backreferencing"	NormParen - presence of a pattern surrounded by parentheses
		Backref - presence of a pattern with normal parentheses and a back reference
11	Line - presence of special characters relating to line boundaries	BegLine - presence of ("ˆ") (matches beginning of line)
		EndLine - presence of ("$") (matches end of line)
		BegEndLine - presence of ("ˆ"..."$") (matches beginning and end of line)
12	Word - presence of sequences that match word beginnings or ends	BegWord - presence of a ("\<") metacharacter (matches word beginning)
		EndWord - presence of a ("\>") metacharacter (matches word end)
		BegEndWord - presence of a ("\<" ... "\>") pattern (matches word end)
13	Edge - presence of sequences that match word boundaries	YesEdgeBeg - presence of a "\b" metacharacter (sequence must lie on a word edge at the beginning - for example "\babc" matches "abcde" but not "xabc")
		YesEdgeEnd - presence of the "\b" metacharacter (sequence must lie on a word edge at the end - for example "abc\b" matches "12abc" but not "abc12")
		YesEdgeBegEnd - presence of "\b" ... "\b" pattern - sequence must lie on a word edge at the beginning and the end (for example "\babc\b" matches "abc" only)
		NoEdgeBeg - presence of "\B" metacharacter - sequence must not lie on a word edge at the beginning (for example, "\Babc" matches "xabce" but not "abcde").
		NoEdgeEnd - presence of "\B" metacharacter - sequence must not lie on a word edge at the end (for example, "abc\B" matches "xabce" but not "xabc").
		NoEdgeBegEnd - presence of "\B" ... "\B" - sequence must not lie on a word edge at the beginning and the end (for example, "\Babc\B" matches "xabce" but not "abcdeabc").

Table 13 Definition of categories and choices for `grep` (continued)

#	Category	Choice
14	Combine - combining multiple patterns	Concatenation - presence of a sequence of tokens (which must all appear in sequence in the text to match - for example, ”ab” matches ”abx” or ”cab” but not ”aaa”, ”axb”, or ”bax”)
		Alternative - presence of two tokens separated by the “|” metacharacter (presence of either token will result in a match - for instance ”a|b” matches ”ast” or byz”)

Table 14: Definition of categories and choices for `look`

#	Category	Choice
1	number of parameters	0
		1
		2
		3
		> 3
2	input	default dictionary (input file name does not exist)
		input file exists
		invalid input file name
3	option d	exist
		not exist
4	option f	exist
		not exist
5	bad option	exist
		not exist
6	search string	exist and length < 250
		exist and length ≥ 250
		not exist
7	search string is found	yes
		no

Table 15: Definition of categories and choices for `printtokens` and `printtokens2`

#	Category	Choice
1	NumOfInputs - number of parameters of the input	Input=0 - an input has no parameters (an empty string input)
		Input=1 - an input has one parameter (input file name)
2	FileExist - presence of the file input	Yes - the file exists
		No - the file does not exist
3	HasEmptyString - presence of an empty string in the file input	Yes - an empty string present
		No - no empty string present
4	HasStringLength80 - presence of a string with length equal to 80 in the file input	Yes - a string with length equal to 80 present
		No - no string with length equal to 80 present
5	HasStringLengthLess80 - presence of a non-empty string with length less than 80 in the file input	Yes - a non-empty string with length less than 80 present
		No - no non-empty string with length less than or equal to 80 present
6	HasStringLengthGreater80 - presence of a string with length greater than 80 in the file input	Yes - a string with length greater than 80 present
		No - no string with length greater than 80 present
7	HasStringWithoutDoubleQuotes - presence of a string having no double quotes in the file input	Yes - a string having no double quotes present
		No - there are no strings without double quotes
8	HasStringWithEvenDoubleQuotes - presence of a string enclosed by a pair of double quotes in the file input	Yes - a string enclosed by a pair of double quotes present
		No - no strings enclosed by a pair of double quotes present
9	HasStringWithOddDoubleQuote - presence of a string not enclosed by a pair of double quotes in the file input	Yes - a string not enclosed by a pair of double quotes present
		No - no strings not enclosed with a pair of double quotes are present
10	BlankInsideEnclosedDoubleQuote - presence a blank string enclosed by a pair of double quotes in file input	Yes - a blank string enclosed by a pair of double quotes present
		No - no blank string enclosed by a pair of double quotes
11	Has# - presence of # in a string in the file input	Yes - # present in any string in the file input
		No - no # present in any string in the file input
12	HasCharAfter# - presence of any characters after # in the file input	Yes - a string with a character after # present
		No - no string with a character after # present
13	HasLambda - presence of keyword “lambda” in the file input	Yes - keyword “lambda” present
		No - keyword “lambda” not present

Table 15 Definition of categories and choices for printtokens and printtokens2 (continued)

#	Category	Choice
14	HasAnd - presence of keyword “and” in the file input	Yes - keyword “and” present
		No - keyword “and” not present
15	HasIf - presence of keyword “if” in the file input	Yes - keyword “if” present
		No - keyword “if” not present
16	HasOr - presence of keyword “or” in the file input	Yes - keyword “or” present
		No - keyword “or” not present
17	HasXor - presence of keyword “xor” in the file input	Yes - keyword “xor” present
		No - keyword “xor” not present
18	HasStandAloneAlphaNum - presence of alphanumeric outside double quotes and not after # in the file input	Yes - an alphanumeric character outside double quotes and before # is present
		No - no alphanumeric character outside double quotes and before a# is present
19	HasLParan - presence of left parenthesis in the file input	Yes - left parenthesis present
		No - left parenthesis not present
20	HasRParan - presence of right parenthesis in the file input	Yes - right parenthesis present
		No - right parenthesis not present
21	HasLBracket - presence of left bracket in the file input	Yes - left bracket present
		No - left bracket not present
22	HasRBracket - presence of right bracket in the file input	Yes - right bracket present
		No - right bracket not present
23	HasQuote - presence of single quote in the file input	Yes - single quote present
		No - single quote not present
24	HasBackQuote - presence of back quote in the file input	Yes- back quote bracket present
		No - back quote not present
25	HasComma - presence of comma in the file input	Yes - comma present
		No - comma not present
26	HasGreaterEqual - presence of (>=) in the file input	Yes - (>=) present
		No - (>=) not present
27	HasSpace - presence of space in the file input	Yes - space present
		No - space not present
28	HasOtherChar - presence of any characters in the file input not included in previous categories	Yes - other characters present
		No - no such characters present

Table 16: Definition of categories and choices for replace

#	Category	Choice
1	NumOfInputParameters - Number of parameters of the input	Input=0 - an input has no parameters (an empty string input)
		Input=1 - an input has one parameter (Regular Expression parameter)
		Input=2 - an input has two parameters (Regular Expression and Replacing String parameters)
		Input=3 - an input has three parameters (Regular Expression, Replacing String, and input file name (containing searched strings to be replaced) parameter)
2	RE_ESC- presence of escape symbol (@) in the regular expression parameter	HasESC - escape symbol present
		NoESC - escape symbol not present
3	RE_BOL - presence of Beginning of Line symbol (%) as a metacharacter	HasMetacharBOL - Beginning of Line symbol present as metacharacter
		NoMetacharBOL - Beginning of Line symbol not present as metacharacter
4	RE_EOL - presence of End of Line symbol ($) as a metacharacter	HasMetacharEOL - End of Line symbol present as metacharacter
		No MetacharEOL - End of Line symbol not present as metacharacter
5	RE_? - presence of symbol ? as a metacharacter	HasMetachar? - symbol ? present as metacharacter
		NoMetachar? - symbol ? not present as metacharacter
6	RE_* - presence of symbol * as a metacharacter	HasMetachar* - symbol * present as metacharacter
		NoMetachar* - symbol * not present as metacharacter
7	RE_EnumCharSet - presence of enumeration type of character set	HasEnumCharSet - enumeration type character set present
		NoEnumCharSet - enumeration type character set not present
8	RE_RangeCharSet - presence of range type of character set	HasRangeCharSet - range type character set present
		NoRangeCharSet - range type character set not present

Table 16 Definition of categories and choices for `replace` (continued)

#	Category	Choice
9	RE_MixCharSet - presence of both enumeration and range type of character set	HasMixCharSet - both enumeration and range type of character set present
		NoMixCharSet - enumeration and range type not both present
10	RE_MetacharNegate - presence of negate symbol [ˆ] as metacharacter	HasMetacharNegate - negate symbol present as metacharacter
		NoMetacharNegate - Negate symbol not present as metacharacter
11	RE_MetacharDash - presence of dash symbol [-] as metacharacter in the range enumeration set	HasMetacharDash - dash symbol [-] present as metacharacter in the range enumeration set
		NoMetacharDash - dash symbol [-] not present as metacharacter in the range enumeration set
12	RE_MetacharTab - presence of metacharacter tab symbol (@t) in the regular expression	HasMetacharTab - tab symbol present as metacharacter
		NoMetacharTab - tab symbol not present as metacharacter
13	RE_MetacharNewLine - presence of metacharacter new-line symbol (@n)	HasMetacharNewLine - new-line symbol present as metacharacter
		NoMetacharNewLine - new-line not present as metacharacter
14	RE_Length - determine the length of the regular expression	≤MAXSTR - the length of the non-empty regular expression is less or equal to a pre-determined constant MAXSTR
		>MAXSTR - the length of the non-empty regular expression is greater than a pre-determined constant MAXSTR
		= 0 - the length of the regular expression is 0 (empty string)
15	RS_Esc - presence of escape symbol (@) in the replacing string parameter	HasESC - escape symbol present
		NoESC - escape symbol not present
16	RS_& - presence of symbol & as a metacharacter in the replacing string parameter	HasMetachar& - symbol & present as metacharacter
		NoMetachar& - symbol & not present as metacharacter
17	RS_MetacharTab - presence of metacharacter tab symbol (@t) in the replacing string parameter	HasMetacharTab - tab symbol present as metacharacter
		NoMetacharTab - tab symbol not present as metacharacter
18	RS_MetacharNewLine - presence of metacharacter new line symbol (@n) in the replacing string parameter	HasMetacharNewLine - new line symbol present as metacharacter
		NoMetacharNewLine - new line symbol not present as metacharacter
19	RS_Length - Classifying the length of the replacing string parameter	≤MAXSTR - the length of the non-empty replacing string is less or equal to a pre-determined constant MAXSTR
		>MAXSTR - the length of the non-empty replacing string is greater than a pre-determined constant MAXSTR
		= 0 - the length of the replacing string is 0 (empty string)
20	F_EndStr - presence of end string character in the file referred by the third parameter of an input	HasEndStr - end string character present in the file
		NoEndStr - end string character not present in the file
21	F_NewLine - presence of new line character in the file referred by the third parameter of an input	HasEndStr - new line character present in the file
		NoEndStr - new line character not present in the file
22	F_String≤MAXSTR - presence of a string shorter than or equal in length to MAXSTR in the file referred by the third input parameter	HasString≤MAXSTR - at least a string with length less or equal to MAXSTR string present in the file
		NoString≤MAXSTR - no string with length less or equal to MAXSTR string present in the file
23	F_String>MAXSTR - presence a string longer than MAXSTR string in the file referred by the third parameter of an input	HasString>MAXSTR - at least a string with length greater than MAXSTR string present in the file
		NoString>MAXSTR - no string with length equal greater than MAXSTR string present in the file
24	F_EmptyString - presence of empty string in the file referred by the third parameter of an input	HasEmptyString - an empty string present in the file
		NoEmptyString - no empty string present in the file

Table 17: Definition of categories and choices for `schedule` and `schedule2`

#	Category	Choice
1	CorrectNumberOfInputParameters - number of parameters of the input	Input=3 - an input has three parameters
		Input≠3 - an input does not have three parameters
2	TotalNumberInitialJobsIn -AllPrioQueues - the total number of initial processes	Tot=0 - the total is zero
		Tot≠0 - the total is not zero
3	InvalidInputInitialJobsInFirstPrioQueue - presence of an invalid input in the first parameter	True - There is an invalid input in the first parameter
		False - There is no invalid input in the first parameter
4	NumberOfInitialJobsInFirstPrioQueue - the number of processes in the first parameter	Num=0 - the number of processes in the first parameter is 0
		Num>0 - the number of processes in the first parameter is > 0
		Num<0 - the number of processes in the first parameter is < 0

Table 17 Definition of categories and choices for `schedule` and `schedule2` (continued)

#	Category	Choice
5	InvalidInputInitialJobsInSecondPrioQueue - presence of an invalid input in the second parameter	True - There is an invalid input in the second parameter
		False - There is no invalid input in the second parameter
6	NumberOfInitialJobsInSecondPrioQueue - the number of processes in the second parameter	Num=0 - the number of processes in the second parameter is 0
		Num>0 - the number of processes in the second parameter is > 0
		Num<0 - the number of processes in the second parameter is < 0
7	InvalidInputInitialJobsInThirdPrioQueue - presence of an invalid input in the third parameter	True - There is an invalid input in the third parameter
		False - There is no invalid input in the third parameter
8	NumberOfInitialJobsInThirdPrioQueue - the number of processes in the third parameter	Num=0 - the number of processes in the third parameter is 0
		Num>0 - the number of processes in the third parameter is $>$ 0
		Num<0 - the number of processes in the third parameter is $<$ 0
9	FileExist - presence of the input file	True - The file is present
		False - The file is not present
10	NumberOfJobCommandsGivenInFile - the number of commands listed in the input file	Num=0 - the number of job commands is 0
		Num>0 - the number of job commands is > 0
11	InvalidContent - presence of invalid contents in the input file	True - There is at least an invalid content
		False - There is no invalid content
12	ContainNewJob - presence of NEW JOB command in the input file	True - The NEW JOB command is present in the input file
		False - The NEW JOB command is not present in the input file
13	ContainUpgradePrio - presence of UPGRADE PRIO command in the input file	True - The UPGRADE PRIO command is present in the input file
		False - The UPGRADE PRIO command is not present in the input file
14	ContainBlock - presence of BLOCK command in the input file	True - The BLOCK command is present in the input file
		False - The BLOCK command is not present in the input file
15	ContainUnBlock - presence of UNBLOCK command in the input file	True - The UNBLOCK command is present in the input file
		False - The UNBLOCK command is not present in the input file
16	ContainQuantumExpire - presence of QUANTUM EXPIRE command in the input file	True - The QUANTUM EXPIRE command is present in the input file
		False - The QUANTUM EXPIRE command is not present in the input file
17	ContainFinish - presence of FINISH command in the input file	True - The FINISH command is present in the input file
		False - The FINISH command is not present in the input file
18	ContainFlush - presence of FLUSH command in the input file	True - The FLUSH command is present in the input file
		False - The FLUSH command is not present in the input file
19	ContainNewJobWithoutPrio - presence of NEW JOB without priority parameter in the input file	True - The NEW JOB command without priority parameter is present in the input file
		False - The NEW JOB command without priority parameter is not present in the input file
20	ContainNewJobWithPrio $>$ MAXPRIO - presence of NEW JOB with priority parameter $>$ MAXPRIO in the input file	True - The NEW JOB command with priority parameter $>$ MAXPRIO is present in the input file
		False - The NEW JOB command with priority parameter $>$ MAXPRIO is not present in the input file
21	ContainNewJobWith0 $<$ Prio $\leq$ MAXPRIO - presence of NEW JOB with priority parameter > 0 and $\leq$ MAXPRIO in the input file	True - The NEW JOB command with priority parameter > 0 and $\leq$ MAXPRIO is present in the input file
		False - The NEW JOB command with priority parameter $>$ 0 and $\leq$ MAXPRIO is not present in the input file
22	ContainNewJobWithPrio ≤ 0 - presence of NEW JOB with priority parameter ≤ 0 in the input file	True - The NEW JOB command with priority parameter ≤ 0 is present in the input file
		False - The NEW JOB command with priority parameter $\leq$ 0 is not present in the input file
23	ContainUpgradePrioWithoutPrio - presence of UPGRADE PRIO without priority parameter in the input file	True - The UPGRADE PRIO command without priority parameter is present in the input file
		False - The UPGRADE PRIO command without priority parameter is not present in the input file
24	ContainUpgradePrioWithPrio $>$ MAXPRIO - presence of UPGRADE PRIO with priority parameter $>$ MAXPRIO in the input file	True - The UPGRADE PRIO command with priority parameter $>$ MAXPRIO is present in the input file
		False - The UPGRADE PRIO command with priority parameter $>$ MAXPRIO is not present in the input file

Table 17 Definition of categories and choices for `schedule` and `schedule2` (continued)

#	Category	Choice
25	ContainUpgradePrioWith0 $<$ Prio $\leq$ MAXPRIO - presence of UPGRADE PRIO with priority parameter $>$ 0 and $\leq$ MAXPRIO in the input file	True - The UPGRADE PRIO command with priority parameter $>$ 0 and $\leq$ MAXPRIO is present in the input file
		False - The UPGRADE PRIO command with priority parameter $>$ 0 and $\leq$ MAXPRIO is not present in the input file
26	ContainUpgradePrioWithPrio $\leq$ 0 - presence of UPGRADE PRIO with priority parameter $\leq$ 0 in the input file	True - The UPGRADE PRIO command with priority parameter $\leq$ 0 is present in the input file
		False - The UPGRADE PRIO command with priority parameter $\leq$ 0 is not present in the input file
27	ContainUpgradePrioWithoutRatio - presence of UPGRADE PRIO without ratio parameter in the input file	True - The UPGRADE PRIO command without ratio parameter is present in the input file
		False - The UPGRADE PRIO command without ratio parameter is not present in the input file
28	ContainUpgradePrioWithRatio $>$ 1 - presence of UPGRADE PRIO with ratio parameter $>$ 1 in the input file	True - The UPGRADE PRIO command with ratio parameter $>$ 1 is present in the input file
		False - The UPGRADE PRIO command with ratio parameter $>$ 1 is not present in the input file
29	ContainUpgradePrioWith0 $<$ Ratio $\leq$ 1 - presence of UPGRADE PRIO with ratio parameter $>$ 0 and $\leq$ 1 in the input file	True - The UPGRADE PRIO command with ratio parameter $>$ 0 and $\leq$ 1 is present in the input file
		False - The UPGRADE PRIO command with ratio parameter $>$ 0 and $\leq$ 1 is not present in the input file
30	ContainUpgradePrioWithRatio$\leq$0 - presence of UPGRADE PRIO with ratio parameter $\leq$ 0 in the input file	True - The UPGRADE PRIO command with ratio parameter $\leq$ 0 is present in the input file
		False - The UPGRADE PRIO command with ratio parameter $\leq$ 0 is not present in the input file
31	ContainUnblockWithoutRatio - presence of UNBLOCK without ratio parameter in the input file	True - The UNBLOCK command without ratio parameter is present in the input file
		False - The UNBLOCK command without ratio parameter is not present in the input file
32	ContainUnblockWithRatio>1 - presence of UNBLOCK with ratio parameter $>$ 1 in the input file	True - The UNBLOCK command with ratio parameter $>$ 1 is present in the input file
		False - The UNBLOCK command with ratio parameter $>$ 1 is not present in the input file
33	ContainUnblockWith0<Ratio$\leq$1 - presence of UNBLOCK with ratio parameter $>$ 0 and $\leq$ 1 in the input file	True - The UNBLOCK command with ratio parameter $>$ 0 and $\leq$ 1 is present in the input file
		False - The UNBLOCK command with ratio parameter $>$ 0 and $\leq$ 1 is not present in the input file
34	ContainUnblockWithRatio$\leq$0 - presence of UNBLOCK with ratio parameter $\leq$ 0 in the input file	True - The UNBLOCK command with ratio parameter $\leq$ 0 is present in the input file
		False - The UNBLOCK command with ratio parameter $\leq$ 0 is not present in the input file

Table 18: Definition of categories and choices for `sort`

#	Category	Choice
1	number of parameters	0
		$\geq$ 1
2	valid input file	exist
		not exist
3	invalid input file	exist
		not exist
4	option b	exist
		not exist
5	option d	exist
		not exist
6	option f	exist
		not exist
7	option i	exist
		not exist
8	option c	exist
		not exist
9	option m	exist
		not exist
10	option n	exist
		not exist

Table 18 Definition of categories and choices for `sort` (continued)

#	Category	Choice
11	option o	exist
		not exist
12	option t	exist
		not exist
13	option T	exist
		not exist
14	option r	exist
		not exist
15	option u	exist
		not exist
16	option . (DOT)	exist
		not exist
17	bad option	exist
		not exist
18	start position	exist
		not exist
19	end position	exist
		not exist
20	the number of keys	< 10
		≥ 10
21	line longer than 2048	exist
		not exist

Table 19: Definition of categories and choices for `spline`

#	Category	Choice
1	number of parameters	0
		≥ 1
2	input	input from screen
		input file exists
		invalid input
3	option a	exist
		not exist
4	option k	exist
		not exist
5	option n	exist
		not exist
6	option p	exist
		not exist
7	option x	exist
		not exist
8	bad option	exist
		not exist
9	number of input data	< 3
		≥ 3 and ≤ 1000
		> 1000
10	input data are monotonic	yes
		no

Table 20: Definition of categories and choices for `tcas`

#	Category	Choice
1	Correct_Number_of_Input_Parameters	$= 12$
		$\neq 12$
2	Invalid_Cur_Vertical_Sep_INPUT	TRUE
		FALSE
3	Vertical_Sep_Degree	$>$ MAXALTDIFF
		$\leq$ MAXALTDIFF and $\geq$ MINSEP
		$<$ MINSEP
4	Invalid_High_Confidence_INPUT	TRUE
		FALSE

Table 20 Definition of categories and choices for `tcas` (continued)

#	Category	Choice
5	High_Confidence	TRUE
		FALSE
6	Invalid_Two_of_Three_Reports_Valid_INPUT	TRUE
		FALSE
7	Is_Report_Valid	TRUE
		FALSE
8	Invalid_Own&Other_Tracked_Alt_INPUT	TRUE
		FALSE
9	Above_or_Below_Treat	Own_Tracked_Alt < Other_Tracked_Alt
		Own_Tracked_Alt > Other_Tracked_Alt
10	Invalid_Own_Tracked_Alt_Rate_INPUT	TRUE
		FALSE
11	Tracked_Alt	$\leq$ OLEV
		> OLEV
12	Invalid_Alt_Layer_Value_INPUT	TRUE
		FALSE
13	Adequate_Separation_Level	< 0
		= 0
		= 1
		= 2
		= 3
		> 3
14	Invalid_Up_Separation_INPUT	TRUE
		FALSE
15	Up_Separation_Threshold	< 400
		$\geq$ 400 and < 500
		$\geq$ 500 and < 640
		$\geq$ 640 and < 740
		$\geq$ 740
16	Invalid_Down_Separation_INPUT	TRUE
		FALSE
17	Down_Separation_threshold	< 400
		$\geq$ 400 and < 500
		$\geq$ 500 and < 640
		$\geq$ 640 and < 740
		$\geq$ 740
18	Up_Preference	Down_Separation < Up_Separation
		Up_Separation $\leq$ Down_Separation < Up_Separation + NOZCROSS
		Down_Separation $\geq$ Up_Separation + NOZCROSS
19	Invalid_Other_RAC_INPUT	TRUE
		FALSE
20	Clear_Intention	= NO_INTENT
		$\neq$ NO_INTENT
21	Invalid_Other_Capability_INPUT	TRUE
		FALSE
22	TCAS_Equipped	TRUE
		FALSE
23	Invalid_Climb_Inhibit_INPUT	TRUE
		FALSE
24	Climb_Inhibit	TRUE
		FALSE

Table 21: Definition of categories and choices for `totinfo`

#	Category	Choice
1	Correct_number_of_input_parameters	= 0
		$\geq$ 1
2	File_Contain_BlankLine	Yes
		No
3	File_Contain_Comment	Yes
		No
4	File_Contain_Invalid_r_Input	Yes
		No

Table 21 Definition of categories and choices for `totinfo` (continued)

#	Category	Choice
5	File_Contain_Invalid_c_Input	Yes
		No
6	File_Contain_r×c>MAXTBL	Yes
		No
7	File_Contain_r×c≤MAXTBL	Yes
		No
8	File_Contain_r_Extremely_Big	Yes
		No
9	File_Contain_r>1	Yes
		No
10	File_Contain_r≤1	Yes
		No
11	File_Contain_c_Extremely_Big	Yes
		No
12	File_Contain_c>1	Yes
		No
13	File_Contain_c≤1	Yes
		No
14	File_Contain_Table(s)_without_Input_r	Yes
		No
15	File_Contain_Table(s)_without_Input_c	Yes
		No
16	File_Contain_Table(s)_Size_Not_Equal_r×c	Yes
		No
17	File_Contain_Table(s)_Size_Equal_r×c	Yes
		No
18	File_Contain_Table(s)_with_Invalid_Cell(s)	Yes
		No
19	File_Contain_Table(s)_with_All_Cells_Valid	Yes
		No
20	File_Contain_Table(s)_with_Negative_Cell(s)	Yes
		No
21	File_Contain_Table(s)_with_All_Cells_Zero	Yes
		No

Table 22: Definition of categories and choices for `uniq`

#	Category	Choice
1	input	input from screen
		input file exists
		invalid input
2	option	u
		d
		c
		not exist
3	input contents sorted	yes
		no
4	fields	exist
		not exist
5	letters	exist
		not exist
6	duplicate lines	exist
		not exist
7	blank lines	exist
		not exist

B Full experimental results of F-measures

In this appendix, we include the complete results of F-measures for RT, *ARTmif*, and *ARTsum* on all 14 object programs. In the tables below, F-ratio refers to the ratio between the F-measures of ART and RT, and "sDev" denotes the sample standard deviation for the F-measures.

Table 23: F-measure data on `cal`

M-	RT		*ARTmif*			*ARTsum*		
ID	F-measure	sDev	F-measure	F-ratio	sDev	F-measure	F-ratio	sDev
1	163.92	159.60	39.50	24.10%	38.60	21.23	12.95%	16.70
2	41.72	39.66	11.19	26.82%	9.27	10.51	25.19%	8.74
3	15.91	15.85	8.75	55.02%	7.69	9.26	58.18%	7.42
4	159.13	158.54	94.32	59.27%	91.36	95.73	60.16%	96.15
5	10.13	9.88	6.35	62.74%	4.71	6.89	68.09%	5.99
6	27.41	26.97	9.05	33.02%	8.07	6.95	25.37%	5.05
7	159.15	165.91	59.78	37.56%	59.69	35.38	22.23%	33.34
8	23.56	23.19	7.30	31.00%	5.59	5.78	24.54%	3.95
9	20.45	19.81	13.30	65.00%	13.01	9.94	48.62%	8.76
10	23.69	23.25	15.53	65.54%	15.13	11.25	47.48%	10.14
11	23.34	22.22	11.95	51.21%	10.80	11.19	47.95%	9.84

Table 24: F-measure data on `comm`

M-	RT		*ARTmif*			*ARTsum*		
ID	**F-measure**	**sDev**	**F-measure**	**F-ratio**	**sDev**	**F-measure**	**F-ratio**	**sDev**
1	125.55	123.89	46.37	36.93%	43.80	47.94	38.19%	38.94
2	31.22	31.09	7.97	25.52%	7.19	6.40	20.51%	5.24
3	186.74	181.12	81.65	43.72%	77.98	81.65	43.72%	77.98
4	18.15	17.43	5.01	27.61%	4.01	4.49	24.72%	3.19
5	12.70	11.81	4.59	36.14%	3.51	4.73	37.27%	3.88
6	19.45	19.54	4.84	24.89%	3.56	4.48	23.03%	2.92
7	10.66	10.41	4.21	39.51%	3.11	4.50	42.17%	3.62
8	21.26	20.44	5.65	26.58%	4.83	5.08	23.91%	3.90
9	13.04	12.30	4.82	36.98%	3.76	5.03	38.53%	4.17
10	63.71	62.17	26.67	41.86%	26.22	26.73	41.96%	26.35
11	93.37	90.24	40.08	42.92%	38.09	40.08	42.93%	38.08
12	35.62	34.28	10.06	28.24%	9.07	8.35	23.44%	6.86
13	194.45	190.52	56.72	29.17%	55.56	40.03	20.59%	37.35
14	42.59	41.20	12.70	29.82%	12.32	10.19	23.93%	8.52
15	149.35	147.47	40.80	27.32%	39.90	32.25	21.59%	30.00
16	26.37	26.28	10.15	38.48%	9.05	14.01	53.13%	14.58
17	45.94	44.15	11.03	24.00%	9.47	10.80	23.51%	8.92
18	36.69	36.21	12.91	35.18%	11.73	15.04	40.98%	15.35
19	75.13	73.67	27.37	36.43%	26.01	45.67	60.79%	47.69
20	11.52	10.66	4.23	36.68%	3.05	4.22	36.63%	3.35
21	147.43	150.74	24.77	16.80%	23.01	17.51	11.88%	15.20
22	143.67	148.86	22.63	15.75%	22.13	16.89	11.75%	15.31
23	10.41	10.06	4.23	40.58%	3.10	4.55	43.71%	3.59
24	12.05	11.08	4.41	36.59%	3.30	4.64	38.51%	3.73
25	26.59	26.28	9.65	36.28%	9.31	10.09	37.94%	9.59
26	10.45	9.48	3.55	33.95%	2.39	3.55	33.95%	2.37
27	73.98	77.57	12.18	16.47%	10.90	9.54	12.89%	7.83

Table 25: F-measure data on `grep`

M-	RT		*ARTmif*			*ARTsum*		
ID	**F-measure**	**sDev**	**F-measure**	**F-ratio**	**sDev**	**F-measure**	**F-ratio**	**sDev**
1	49.54	48.51	17.72	35.77%	15.05	21.03	42.46%	20.27
2	14.85	14.28	6.18	41.63%	4.58	6.17	41.54%	5.05
3	207.31	209.05	78.01	37.63%	75.23	85.38	41.19%	82.45
4	858.96	862.44	1844.73	214.76%	1163.69	235.39	27.40%	224.52
5	474.26	469.57	154.28	32.53%	147.54	200.21	42.22%	202.59
6	650.99	663.96	215.30	33.07%	211.32	290.47	44.62%	286.39
7	14.40	14.01	6.13	42.55%	4.55	6.00	41.65%	4.79
8	277.52	269.52	94.73	34.14%	87.00	121.81	43.89%	118.95
9	14.89	14.34	6.20	41.65%	4.60	6.19	41.57%	5.03
10	463.90	459.89	156.62	33.76%	148.51	206.06	44.42%	209.24
11	35.75	36.97	16.44	45.98%	14.35	15.56	43.54%	14.48
12	22.20	21.62	19.20	86.47%	20.34	9.58	43.16%	8.86
13	15.34	14.72	6.64	43.27%	5.19	6.47	42.18%	5.20
14	14.88	14.26	6.21	41.72%	4.59	6.19	41.59%	4.94
15	46.46	44.69	49.70	106.98%	52.52	25.39	54.65%	26.72
16	36.47	35.50	13.95	38.26%	12.35	16.03	43.96%	14.77
17	34.90	34.19	46.41	132.95%	52.68	15.85	45.40%	15.15
18	34.18	33.27	13.07	38.25%	10.88	14.24	41.66%	12.79
19	59.68	58.03	22.25	37.29%	20.07	25.58	42.85%	24.93
20	2.23	1.66	2.03	91.42%	1.35	1.93	86.86%	1.19

Table 26: F-measure data on `look`

M-	RT		*ARTmif*			*ARTsum*		
ID	**F-measure**	**sDev**	**F-measure**	**F-ratio**	**sDev**	**F-measure**	**F-ratio**	**sDev**
1	10.01	9.87	5.49	54.83%	3.78	5.81	58.08%	3.89
2	12.93	12.29	6.05	46.78%	4.04	6.22	48.11%	4.13
3	13.78	13.03	6.23	45.18%	4.16	6.22	45.13%	4.08
4	47.37	47.22	79.45	167.70%	82.65	49.08	103.61%	48.09
5	11.24	10.68	5.83	51.83%	4.13	5.90	52.50%	3.89
6	15.89	15.68	8.98	56.53%	7.34	8.77	55.18%	6.98
7	12.00	11.46	5.90	49.16%	4.08	6.10	50.80%	4.16
8	10.81	10.06	5.23	48.40%	3.21	13.33	123.31%	9.53
9	11.94	11.62	5.89	49.30%	4.17	6.21	51.98%	4.16
10	10.67	10.38	5.66	52.99%	3.86	5.81	54.48%	3.91
11	12.78	12.17	6.02	47.11%	4.14	6.21	48.59%	4.15
12	14.87	14.81	8.87	59.65%	7.22	8.70	58.49%	6.96
13	38.47	38.21	22.66	58.90%	21.77	18.56	48.24%	15.04
14	17.70	17.02	8.21	46.40%	6.17	8.59	48.53%	6.56
15	45.94	46.58	84.34	183.59%	85.76	50.35	109.60%	49.23
16	14.91	14.15	6.20	41.60%	4.02	6.44	43.22%	4.13
17	13.71	13.76	8.56	62.41%	6.82	8.54	62.27%	6.89
18	63.03	66.53	116.55	184.92%	120.60	77.18	122.46%	76.29
19	48.07	46.98	23.62	49.14%	22.22	20.11	41.83%	16.16
20	37.71	38.62	70.71	187.53%	71.99	64.08	169.94%	66.09
21	46.63	48.60	87.96	188.65%	87.25	71.15	152.60%	74.05
22	192.94	193.43	320.25	165.98%	320.13	370.06	191.80%	383.64
23	21.63	21.02	9.67	44.68%	8.08	9.32	43.08%	7.15
24	10.72	9.84	7.67	71.54%	6.37	7.84	73.19%	6.29
25	194.06	203.12	380.96	196.31%	391.12	453.28	233.58%	487.30
26	10.01	9.75	5.58	55.78%	3.83	5.65	56.45%	3.81
27	11.41	10.87	5.74	50.32%	3.97	5.90	51.68%	3.95
28	26.59	26.47	20.52	77.18%	18.89	16.85	63.36%	13.97
29	17.41	16.64	7.46	42.86%	5.41	8.36	48.04%	6.45

Table 27: F-measure data on `printtokens`

M-	**RT**		***ARTmif***			***ARTsum***		
ID	**F-measure**	**sDev**	**F-measure**	**F-ratio**	**sDev**	**F-measure**	**F-ratio**	**sDev**
1	456.11	450.18	220.03	48.24%	219.36	161.49	35.41%	162.05
2	55.20	53.64	27.70	50.17%	28.26	21.27	38.53%	20.47
3	73.28	75.70	39.95	54.52%	38.78	32.93	44.93%	30.08
4	91.41	91.87	56.83	62.17%	55.69	50.93	55.72%	48.90
5	7.16	6.55	4.11	57.42%	3.20	4.47	62.39%	3.08
6	14.95	14.91	6.04	40.42%	5.18	6.10	40.79%	4.36
7	97.35	98.07	47.11	48.39%	46.36	41.35	42.48%	40.10

Table 28: F-measure data on `printtokens2`

M-	**RT**		***ARTmif***			***ARTsum***		
ID	**F-measure**	**sDev**	**F-measure**	**F-ratio**	**sDev**	**F-measure**	**F-ratio**	**sDev**
1	10.69	10.75	5.25	49.16%	4.42	5.45	50.97%	3.93
2	10.32	10.37	5.02	48.70%	4.18	5.28	51.23%	3.79
3	246.72	250.76	197.19	79.93%	195.32	290.36	117.69%	300.13
4	8.21	8.06	3.95	48.09%	2.99	4.26	51.84%	2.86
5	16.25	16.12	10.45	64.32%	9.98	10.19	62.74%	8.72
6	5.33	4.89	3.13	58.77%	2.19	3.46	64.99%	2.25
7	12.73	12.78	6.09	47.86%	5.06	6.21	48.75%	4.64
8	10.85	10.07	4.84	44.60%	3.82	5.11	47.12%	3.53
9	44.14	43.57	18.48	41.88%	18.27	15.29	34.64%	12.97
10	16.25	16.12	10.45	64.32%	9.98	10.19	62.74%	8.72

Table 29: F-measure data on `replace`

M-	**RT**		***ARTmif***			***ARTsum***		
ID	**F-measure**	**sDev**	**F-measure**	**F-ratio**	**sDev**	**F-measure**	**F-ratio**	**sDev**
1	146.86	150.89	103.12	70.22%	100.26	118.25	80.52%	117.21
2	208.22	215.89	186.20	89.43%	188.82	198.78	95.47%	207.34
3	87.33	90.86	85.93	98.40%	82.80	86.56	99.12%	86.90
4	43.60	44.15	39.31	90.16%	37.30	67.78	155.47%	66.97
5	57.17	56.23	67.06	117.30%	68.81	90.43	158.17%	95.90
6	51.76	53.02	90.13	174.12%	92.45	200.28	386.92%	213.42
7	63.93	64.47	62.29	97.43%	64.12	80.75	126.32%	84.48
8	104.42	101.85	69.60	66.65%	65.65	90.73	86.89%	93.30
9	248.82	246.24	510.04	204.98%	523.93	1019.27	409.64%	1036.60
10	233.26	225.35	417.57	179.01%	434.28	770.69	330.39%	774.09
11	248.82	246.24	510.04	204.98%	523.93	1019.27	409.64%	1036.60
12	18.13	17.37	14.07	77.63%	13.40	12.60	69.50%	11.37
13	41.02	40.46	38.05	92.76%	35.72	60.05	146.39%	62.18
14	36.05	36.07	54.84	152.11%	55.31	93.87	260.36%	102.70
15	91.21	88.20	69.79	76.51%	58.78	58.21	63.82%	54.36
16	63.93	64.47	62.29	97.43%	64.12	80.75	126.32%	84.48
17	467.79	475.68	611.58	130.74%	630.08	611.47	130.71%	630.15
18	27.81	27.78	45.41	163.28%	46.80	95.95	345.03%	109.27
19	2604.75	2623.05	907.56	34.84%	964.92	789.75	30.32%	808.45
20	518.01	521.94	660.65	127.54%	655.59	660.63	127.53%	655.60
21	2142.99	2111.79	842.66	39.32%	832.82	758.64	35.40%	780.90
22	137.80	132.78	210.80	152.97%	207.70	376.50	273.22%	385.52
23	418.51	417.10	473.04	113.03%	492.49	748.57	178.87%	756.10
24	51.73	50.61	33.34	64.44%	33.15	40.59	78.47%	40.04
25	2385.25	2321.38	1650.46	69.19%	1692.29	2383.61	99.93%	2378.74
26	50.95	50.57	71.80	140.91%	72.93	129.01	253.19%	141.93
27	42.21	40.78	27.20	64.44%	25.72	32.81	77.72%	32.64
28	40.34	40.25	20.70	51.31%	18.14	28.83	71.47%	27.20
29	82.82	81.89	48.71	58.81%	46.45	56.21	67.87%	55.98
30	19.91	19.65	12.41	62.32%	11.11	15.37	77.23%	14.32
31	27.34	27.36	45.84	167.66%	48.02	93.63	342.43%	106.26

Table 30: F-measure data on `schedule`

M-	RT		*ARTmif*			*ARTsum*		
ID	**F-measure**	**sDev**	**F-measure**	**F-ratio**	**sDev**	**F-measure**	**F-ratio**	**sDev**
1	396.32	395.60	56.22	14.19%	56.15	48.18	12.16%	44.96
2	12.96	12.57	47.21	364.26%	45.88	30.27	233.57%	33.62
3	16.00	15.54	57.00	356.36%	56.93	57.21	357.63%	63.52
4	9.65	9.14	48.60	503.38%	49.71	34.52	357.53%	40.07
5	77.75	78.83	51.28	65.95%	46.10	36.96	47.54%	32.51
6	396.32	395.60	56.22	14.19%	56.15	48.18	12.16%	44.96
7	101.78	102.47	161.14	158.32%	161.47	117.84	115.78%	117.16
8	104.75	109.12	523.72	499.96%	524.82	330.23	315.25%	345.64
9	121.24	123.53	24.23	19.98%	19.79	19.14	15.78%	15.02

Table 31: F-measure data on `schedule2`

M-	RT		*ARTmif*			*ARTsum*		
ID	**F-measure**	**sDev**	**F-measure**	**F-ratio**	**sDev**	**F-measure**	**F-ratio**	**sDev**
1	88.34	87.11	65.41	74.04%	64.23	34.69	39.27%	30.92
2	104.68	104.80	105.78	101.04%	99.38	95.81	91.52%	96.66
3	111.45	111.98	48.48	43.50%	42.99	33.09	29.69%	27.84
4	33.92	46.02	836.20	2465.22%	833.37	518.08	1527.36%	507.73
5	163.30	160.99	45.67	27.97%	44.34	34.87	21.35%	29.12
6	447.98	449.46	88.28	19.71%	83.39	105.99	23.66%	97.82
7	104.68	104.80	105.78	101.04%	99.38	95.81	91.52%	96.66
8	66.64	64.42	22.18	33.29%	19.08	16.74	25.11%	12.39
9	68.50	66.75	23.22	33.90%	19.85	17.12	24.99%	12.81

Table 32: F-measure data on `sort`

M-ID	RT F-measure	RT sDev	*ARTmif* F-measure	*ARTmif* F-ratio	*ARTmif* sDev	*ARTsum* F-measure	*ARTsum* F-ratio	*ARTsum* sDev
1	97.46	94.12	96.58	99.10%	102.41	16.88	17.32%	13.39
2	22.60	22.20	9.79	43.33%	8.35	10.31	45.62%	8.84
3	81.83	78.85	96.32	117.71%	103.49	16.79	20.52%	13.23
4	168.96	165.34	75.46	44.66%	72.01	54.23	32.10%	50.21
5	10.17	9.20	12.84	126.22%	12.87	5.87	57.70%	4.34
6	26.37	25.64	29.98	113.72%	28.91	15.93	60.43%	14.02
7	37.70	37.35	14.96	39.69%	13.92	12.18	32.30%	10.66
8	37.89	37.38	14.87	39.25%	13.90	12.34	32.56%	10.78
9	241.25	243.33	80.23	33.26%	74.84	213.80	88.62%	210.13
10	334.95	336.92	187.04	55.84%	182.25	165.69	49.47%	155.96
11	111.14	106.75	105.65	95.06%	112.24	17.55	15.79%	14.12
12	89.35	87.68	71.16	79.64%	74.59	17.45	19.53%	14.11
13	333.80	331.32	185.86	55.68%	175.50	160.95	48.22%	148.52
14	334.08	346.59	218.64	65.44%	217.10	159.51	47.75%	149.17
15	506.63	500.51	436.00	86.06%	425.10	370.78	73.18%	346.82
16	244.03	252.56	114.22	46.80%	110.60	132.83	54.43%	124.16
17	241.42	235.40	69.80	28.91%	64.33	129.37	53.59%	124.64
18	240.46	249.31	105.96	44.06%	100.55	133.34	55.45%	123.25
19	326.57	332.04	121.44	37.19%	112.42	134.04	41.04%	124.01
20	244.52	251.90	80.99	33.12%	75.54	116.97	47.83%	111.03
21	256.44	257.36	210.46	82.07%	203.62	146.64	57.18%	141.64
22	243.53	247.01	103.97	42.69%	100.87	132.47	54.40%	124.76
23	160.44	162.71	66.34	41.35%	63.64	116.97	72.90%	112.75
24	343.90	357.48	239.16	69.54%	237.26	163.54	47.56%	153.34
25	337.09	341.08	114.12	33.86%	107.35	55.71	16.53%	50.78
26	36.15	35.99	14.96	41.40%	14.00	12.11	33.51%	10.58
27	10.80	10.25	12.10	112.07%	12.58	6.47	59.95%	5.04
28	329.97	327.03	81.87	24.81%	76.91	130.32	39.49%	125.85
29	136.26	137.70	132.42	97.18%	124.15	160.57	117.84%	151.76
30	241.74	241.56	244.28	101.05%	240.35	83.29	34.45%	74.62
31	249.34	260.05	96.44	38.68%	95.88	134.04	53.76%	124.63
32	248.46	249.51	211.00	84.92%	199.82	93.62	37.68%	84.40
33	163.92	170.49	63.59	38.80%	58.61	114.08	69.59%	111.74
34	138.65	139.79	56.67	40.87%	51.04	102.47	73.90%	99.48
35	164.54	169.61	76.51	46.50%	75.46	130.63	79.39%	126.69
36	246.50	241.94	162.66	65.99%	150.14	98.08	39.79%	87.15
37	249.23	244.80	186.58	74.86%	186.08	98.08	39.36%	87.10
38	247.45	254.25	116.82	47.21%	114.45	133.32	53.88%	124.17
39	242.51	233.71	203.43	83.88%	196.51	98.43	40.59%	88.52
40	58.78	58.97	31.48	53.55%	25.28	9.60	16.32%	7.01
41	51.64	52.02	31.09	60.20%	25.18	9.53	18.45%	7.01
42	140.61	141.81	152.92	108.76%	151.48	80.90	57.54%	73.83
43	144.77	141.68	220.07	152.01%	218.26	110.01	75.99%	105.74
44	12.79	12.37	7.62	59.58%	6.17	6.76	52.81%	5.32
45	245.06	245.41	207.33	84.60%	206.37	97.45	39.77%	87.17
46	247.95	247.58	107.50	43.35%	101.47	143.02	57.68%	140.97
47	340.13	353.07	271.26	79.75%	259.96	144.45	42.47%	133.73
48	250.36	251.62	156.78	62.62%	152.88	97.45	38.93%	87.11

Table 33: F-measure data on `spline`

M-	**RT**		***ARTmif***			***ARTsum***		
ID	**F-measure**	**sDev**	**F-measure**	**F-ratio**	**sDev**	**F-measure**	**F-ratio**	**sDev**
1	17.36	16.76	7.01	40.40%	5.38	13.19	75.97%	13.09
2	27.63	27.40	8.31	30.09%	6.26	9.59	34.70%	7.82
3	13.17	13.00	6.87	52.16%	5.47	6.06	46.04%	4.68
4	78.61	79.37	15.97	20.32%	14.05	16.72	21.27%	14.56
5	28.53	27.84	9.04	31.68%	7.04	10.57	37.05%	9.73
6	10.94	10.90	5.22	47.76%	3.82	5.04	46.08%	3.70
7	26.53	25.69	13.08	49.30%	11.89	23.41	88.27%	23.76
8	44.07	42.29	71.19	161.52%	71.10	135.49	307.42%	131.06
9	47.49	45.20	31.51	66.34%	30.06	31.53	66.38%	30.06
10	43.50	42.56	19.91	45.77%	18.44	22.98	52.84%	21.85
11	12.92	12.44	5.96	46.14%	4.54	6.17	47.71%	4.98
12	77.60	76.57	24.22	31.21%	21.93	21.99	28.33%	19.27
13	45.37	45.15	30.92	68.16%	28.40	77.87	171.63%	92.80
14	86.13	87.35	47.13	54.73%	45.80	37.77	43.86%	36.82
15	32.97	33.73	19.22	58.30%	17.57	15.19	46.07%	13.46
16	15.26	14.54	29.52	193.51%	30.71	27.50	180.24%	25.74

Table 34: F-measure data on `tcas`

M-	RT		*ARTmif*			*ARTsum*		
ID	**F-measure**	**sDev**	**F-measure**	**F-ratio**	**sDev**	**F-measure**	**F-ratio**	**sDev**
1	21.77	21.60	43.48	199.70%	42.69	45.57	209.31%	54.51
2	34.07	33.89	67.62	198.44%	65.06	62.65	183.87%	58.72
3	54.55	55.40	62.91	115.33%	56.39	66.60	122.10%	57.05
4	46.03	45.71	56.89	123.59%	53.05	71.91	156.22%	55.89
5	92.85	90.62	53.47	57.59%	48.35	44.81	48.26%	41.71
6	216.22	216.29	415.54	192.18%	398.35	395.97	183.13%	563.45
7	55.08	58.10	91.02	165.26%	92.91	115.59	209.87%	89.46
8	730.75	745.14	928.62	127.08%	893.77	844.43	115.56%	582.84
9	272.46	261.88	431.67	158.43%	442.12	405.95	149.00%	287.28
10	216.22	216.29	401.88	185.87%	384.49	400.08	185.03%	524.35
11	210.46	211.03	406.38	193.09%	391.19	446.85	212.32%	482.28
12	19.20	18.21	15.82	82.42%	14.54	14.85	77.34%	13.40
13	240.58	241.49	121.84	50.64%	119.35	82.42	34.26%	77.85
14	40.93	40.65	78.77	192.45%	78.00	87.61	214.05%	88.88
15	92.85	90.62	53.47	57.59%	48.35	44.81	48.26%	41.71
16	41.17	41.96	82.02	199.20%	83.77	64.31	156.19%	78.52
17	74.34	77.70	146.16	196.61%	146.40	170.29	229.07%	182.19
18	95.67	92.10	155.89	162.96%	150.03	192.89	201.63%	180.64
19	144.43	143.32	311.78	215.87%	317.91	226.89	157.09%	306.89
20	158.97	158.54	231.35	145.53%	220.31	226.73	142.63%	176.88
21	131.05	127.01	123.91	94.55%	115.70	149.10	113.77%	118.65
22	149.06	144.90	176.01	118.08%	164.11	217.89	146.18%	181.20
23	61.79	62.54	77.84	125.98%	75.40	67.85	109.81%	58.41
24	148.11	144.33	191.06	129.00%	188.96	232.44	156.93%	178.78
25	344.63	343.06	606.44	175.97%	606.43	623.85	181.02%	459.39
26	132.24	133.98	67.03	50.69%	61.19	46.16	34.91%	39.93
27	92.85	90.62	53.47	57.59%	48.35	44.81	48.26%	41.71
28	26.98	27.09	32.18	119.29%	29.60	31.30	116.03%	27.24
29	74.10	73.85	90.65	122.33%	86.79	106.62	143.88%	88.72
30	41.59	42.02	49.08	118.00%	45.48	49.50	119.02%	42.74
31	216.22	216.29	401.88	185.87%	384.49	380.11	175.80%	532.96
32	1049.66	1013.08	1717.96	163.67%	1747.59	1202.44	114.56%	984.95
33	18.26	18.08	35.36	193.69%	34.53	40.22	220.26%	42.17
34	18.67	17.78	13.36	71.54%	11.58	11.81	63.28%	10.35
35	26.98	27.09	32.18	119.29%	29.60	31.30	116.03%	27.24
36	17.94	17.37	26.03	145.10%	24.29	31.18	173.84%	28.80
37	15.56	15.13	26.01	167.10%	24.56	26.59	170.84%	27.19
38	25.92	25.38	43.70	168.60%	40.74	47.61	183.69%	43.97
39	344.63	343.06	606.44	175.97%	606.43	623.85	181.02%	462.46
40	17.94	17.37	26.03	145.10%	24.29	31.18	173.84%	28.80
41	46.03	45.71	57.64	125.23%	54.45	56.42	122.57%	43.80

Table 35: F-measure data on `totinfo`

M-	RT		*ARTmif*			*ARTsum*		
ID	**F-measure**	**sDev**	**F-measure**	**F-ratio**	**sDev**	**F-measure**	**F-ratio**	**sDev**
1	5.19	4.75	3.03	58.26%	1.77	3.32	64.01%	1.87
2	119.13	115.90	44.50	37.36%	43.74	140.79	118.18%	159.43
3	276.96	286.66	666.26	240.56%	640.63	835.75	301.76%	619.45
4	23.51	22.46	18.55	78.88%	17.93	18.82	80.04%	17.68
5	7.82	7.54	18.79	240.31%	16.59	25.15	321.64%	28.33
6	16.09	15.45	14.88	92.52%	14.36	16.84	104.69%	17.47
7	6.62	6.11	5.41	81.80%	4.28	5.82	87.91%	4.92
8	4.07	3.55	3.34	81.93%	2.41	3.34	81.98%	2.39
9	7.24	7.00	16.22	224.09%	14.23	26.40	364.70%	28.88
10	105.46	104.56	48.89	46.36%	43.49	56.95	54.00%	52.91
11	4.07	3.55	3.34	81.93%	2.41	3.34	81.98%	2.39
12	23.39	22.91	21.85	93.41%	21.91	22.28	95.25%	23.30
13	6.35	5.89	5.21	82.03%	4.00	5.49	86.41%	4.61
14	373.69	390.93	961.06	257.18%	968.38	795.14	212.78%	657.78
15	4.07	3.55	3.34	81.93%	2.41	3.34	81.98%	2.39
16	4.81	4.22	3.94	81.92%	3.02	4.00	83.06%	3.09
17	17.29	16.44	15.32	88.66%	14.42	17.15	99.19%	16.60
18	6.97	6.62	3.80	54.47%	2.66	4.02	57.64%	2.50
19	8.62	8.30	8.70	100.91%	8.14	8.59	99.70%	8.25
20	9.83	9.61	5.16	52.45%	3.41	5.39	54.79%	3.60
21	6.72	6.33	3.14	46.69%	2.01	3.38	50.26%	1.68
22	33.65	34.01	32.27	95.90%	27.49	29.27	86.97%	24.51
23	11.01	10.23	8.34	75.80%	7.06	9.15	83.12%	8.50

Table 36: F-measure data on `uniq`

M-	RT		*ARTmif*			*ARTsum*		
ID	**F-measure**	**sDev**	**F-measure**	**F-ratio**	**sDev**	**F-measure**	**F-ratio**	**sDev**
1	11.46	10.98	6.52	56.93%	5.13	6.82	59.53%	5.53
2	15.87	15.70	10.96	69.06%	9.72	8.31	52.37%	6.49
3	13.03	12.87	13.69	105.06%	13.03	13.75	105.54%	13.19
4	16.42	15.48	12.90	78.55%	11.33	13.27	80.82%	12.24
5	32.93	32.70	29.91	90.83%	27.16	35.66	108.27%	32.80
6	40.24	40.79	27.70	68.85%	25.34	35.67	88.64%	37.20
7	13.64	13.18	12.88	94.42%	11.84	17.75	130.14%	18.17
8	26.34	25.38	22.96	87.16%	21.45	23.18	88.00%	23.56
9	25.02	23.83	20.22	80.80%	18.70	20.45	81.72%	21.31
10	35.34	36.14	28.66	81.11%	26.27	38.46	108.84%	35.79
11	16.91	16.20	13.24	78.29%	11.73	13.61	80.50%	12.33
12	11.47	10.95	10.65	92.84%	9.64	10.83	94.43%	9.90
13	38.47	39.86	35.20	91.49%	33.90	38.46	99.98%	35.37
14	43.10	44.56	26.81	62.21%	25.59	17.74	41.16%	14.74
15	71.86	71.97	12.68	17.65%	10.65	9.84	13.69%	7.23
16	11.81	11.10	10.75	91.06%	9.26	12.95	109.64%	12.53
17	39.45	40.38	24.70	62.61%	24.34	16.22	41.12%	14.03
18	17.27	17.27	10.78	62.40%	10.05	7.92	45.83%	6.53
19	13.02	12.42	7.40	56.83%	5.47	8.33	63.98%	6.94
20	419.13	419.45	134.04	31.98%	133.97	60.19	14.36%	55.20
21	30.01	29.38	20.39	67.93%	18.75	21.87	72.88%	21.89
22	21.79	21.41	13.50	61.96%	11.27	13.87	63.67%	12.31
23	23.68	22.83	17.43	73.61%	15.97	11.55	48.79%	9.86
24	60.49	59.48	41.08	67.92%	40.95	30.24	50.00%	30.32
25	22.29	21.33	16.23	72.82%	14.96	10.81	48.48%	8.96
26	10.70	10.45	5.27	49.22%	4.37	5.21	48.67%	4.13
27	11.99	11.31	6.61	55.10%	4.87	7.37	61.49%	6.00
28	25.79	25.60	18.81	72.94%	17.13	13.36	51.80%	12.13
29	23.65	22.76	16.30	68.93%	15.00	11.29	47.75%	9.40