Spectral debugging: How much better can we do?

Lee Naish, Hua Jie Lee and Kotagiri Ramamohanarao
Department of Computing and Information Systems
The University of Melbourne, Victoria 3010, Australia
lee,kotagiri@unimelb.edu.au, huajie.lee@gmail.com

Abstract

This paper investigates software fault localization methods which are based on program spectra – data on execution profiles from passed and failed tests. We examine a standard method of spectral fault localization: for each statement we determine the number of passed and failed tests in which the statement was/wasn't executed and a function, or *metric*, of these four values is used to rank statements according to how likely they are to be buggy. Many different metrics have been used. Here our main focus is to determine how much improvement in performance could be achieved by finding better metrics. We define the cost of fault localization using a given metric and the *unavoidable cost*, which is independent of the choice of metric. We define a class of *strictly rational* metrics and argue that is reasonable to restrict attention to these metrics. We show that every *single bug optimal* metric performs as well as any strictly rational metric for single bug programs, and the resulting cost is the unavoidable cost. We also show how any metric can be adapted so it is single bug optimal, and give results of empirical experiments using single- and two-bug programs.

1 Introduction

Bugs are pervasive in software under development and tracking them down contributes greatly to the cost of software development. One of many useful sources of data to help diagnosis is the dynamic behaviour of software as it is executed over a set of test cases where it can be determined if each result is correct or not; each test case is said to *pass* or *fail*. Software can be instrumented automatically to gather data known as program spectra (Reps, Ball, Das & Larus 1997), such as the statements that are executed, for each test case. If a certain statement is executed in many failed tests but few passed tests we may conclude it is likely to be buggy. Typically the raw data is aggregated to get the numbers of passed and failed tests for which each statement is/isn't executed. Some function is applied to this aggregated data to rank the statements, from those most likely to be buggy to those least likely. We refer to such functions as *metrics*. A programmer can then use the ranking to help find a bug.

We make the following contributions:

- We define a class of metrics we call *strictly rational* metrics and argue why restricting attention to such metrics is reasonable.
- We define the *unavoidable cost* of bug localization, and show it is the minimum cost for any strictly rational metric.
- We show that a class of previously proposed metrics lead to the lowest possible cost of any strictly rational metrics for programs with single bugs.
- We show how any metric can be easily adapted so it it optimal for single bug programs.
- We evaluate several metrics for benchmark sets with one and two bugs.
- We perform additional experiments to help further understand the best known metric for two-bug programs.
- We suggest how test selection strategies can be improved to help performance of bug localization.

The rest of this paper is structured as follows. We first describe spectral fault localization and define the metrics we evaluate in this paper. Section 3 revisits metrics and defines (strictly) rational metrics. Section 4 describes how the cost of fault localization is measured in this paper and also introduces the idea of "unavoidable" cost. Section 5 discusses previous work on optimal metrics for single bug-programs and significantly extends those results. Section 6 shows how any metric can be adapted so it is optimal for single bug programs. Section 7 describes empirical experiments and their results and Section 9 concludes.

2 Background — Spectral fault localization

All spectral methods use a set of tests, each classified as failed or passed; this can be represented as a binary vector, where 1 indicates failed and 0 indicates passed. For statement spectra (Jones & Harrold 2005, Abreu, Zoeteweij & van Gemund 2006, Wong, Qi, Zhao & Cai 2007, Xie, Chen & Xu 2010, Naish, Lee & Kotagiri 2011, Lee 2011), which we use here, we gather data on whether each statement is executed or not for each test. This can be represented as a binary matrix with a row for each statement and a column for each test; 1 means executed and 0 means not executed. For each statement, four numbers are ultimately produced. They are the number of passed/failed test cases in which the statement was/wasn't executed. We adapt the notation from Abreu et al. (Abreu et al. 2006) — $\langle a_{np}, a_{nf}, a_{ep}, a_{ef} \rangle$. The first part of the subscript indicates whether the statement was executed (e) or not (n) and the second indicates whether the test passed (p) or failed (f). We use superscripts to indicate the statement number where appropriate. For example, a^1_{ep} is the number of passed tests that executed statement 1. We use F and P to denote the total number of tests which

A version of this paper with a different copyright notice is to appear at 35th Australasian Computer Science Conference (ACSC 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology, Vol. 122. Mark Reynolds and Bruce Thomas, Ed.

Table 1: Statement spectra with tests $T_1 \ldots T_5$

	T_1	T_2	T_3	T_4	T_5	a_{ef}	a_{ep}
S_1	1	0	0	1	0	1	1
S_2	1	1	0	1	0	2	1
S_3	1	1	1	0	1	2	2
				⋮			
Res.	1	1	0	0	0	$F = 2$	$P = 3$

fail and pass, respectively. Clearly, $a_{nf} = F - a_{ef}$ and $a_{np} = P - a_{ep}$. In most of this paper we avoid explicit use of a_{nf} and a_{np}; making F and P explicit suits our purposes better. Table 1 gives an example binary matrix of execution data and binary vector containing the test results. This data allows us to compute F, P and the a_{ij} values, $i \in \{n, e\}$ and $j \in \{p, f\}$.

Metrics, which are numeric functions, can be used to rank the statements. Most commonly they are defined in terms of the four a_{ij} values. Statements with the highest metric values are considered the most likely to be buggy. We would expect buggy statements to generally have relatively high a_{ef} values and relatively low a_{ep}. In the example in Table 1, Statement 2 (S_2) is executed in both failed tests and only one passed test, which is the minimum for all statements, and thus would typically be ranked highest. The relative rank of statements 1 and 3 is not so clear cut, since statement 3 is executed in more failed tests but also more passed tests. One way of viewing the ranking is that rows of the matrix are ranked according to how "similar" they are to the vector of test results, or how similar the set of failed tests is to the set of tests which execute the statement. Measures of similarity are important in classification and machine learning, not just fault localization, and *many* different metrics have been proposed; Lee (2011) evaluates the fault localization performance of 50 metrics.

Programmers searching for a bug are expected to examine statements, starting from the highest-ranked statement, until a buggy statement is found. In reality, programmers are likely to modify the ranking due to their own understanding of whether the code is likely to be buggy, based on other information such as static analysis, the history of software changes, *et cetera*. Also, checking correctness generally cannot be done by a single statement at a time, or even one basic block at a time. Evaluation of different ranking methods, which we discuss in Section 4, generally ignores such refinement and just depends on where the bug(s) appear in the ranking.

Table 2 gives definitions of the metrics used here, all of which have been evaluated for fault localization in Naish et al. (2011) and their origins are discussed more there. Space prevents us from including all proposed metrics. Several were originally proposed for spectral fault localization: Tarantula (Jones & Harrold 2005), which is generally credited as being the first spectral fault localization system, Zoltar (Gonzalez 2007), Wong3, the best of several metrics proposed in Wong et al. (2007), and O and O^p (Naish et al. 2011). We also use a metric we refer to as Wong4, which was devised for fault localization. It is Heuristic III of Wong, Debroy & Choi (2010), with $\alpha = 0.0001$; we do not provide its definition here due to its complexity. Also, Ample is an adaptation, from Abreu et al. (2006), of a metric developed for the AMPLE system (Dallmeier, Lindig & Zeller 2005) and CBILog is an adaptation, from Naish et al. (2011), of a metric developed for the CBI system (Liblit, Naik, Zheng, Aiken & Jordan 2005). Jaccard (Jaccard 1901), the oldest metric here, originally used for classification of plants, has been used in the Pinpoint system (Chen, Kiciman, Fratkin, Fox & Brewer 2002). Ochiai (Ochiai 1957) and Russell (Russel & Rao 1940), both developed for other domains, were first evaluated for fault localization in Abreu et al. (2006) and Kulczynski2 (see Lourenco, Lobo & Bação (2004)) was first evaluated for fault localization in Naish et al. (2011). Kulczynski2 is the best metric we know of for two and three bug programs (see Naish, Lee & Kotagiri (2009), for example). We discuss some of these metrics in more detail later.

3 Ranking metrics, revisited

The formulas for ranking metrics are only used in constrained ways. We know that a_{ef}, a_{nf}, a_{ep} and a_{np} are all natural numbers. We can also assume there is at least one test case. Furthermore, for any given program and set of test cases F and P are fixed, and $a_{ef} \leq F$ and $a_{ep} \leq P$. Therefore it is possible to define metrics as follows (in Table 2 we can assume a_{nf} and a_{np} are defined in terms of the other values):

Definition 1 (Metric) *A* metric *is a partial function from four natural numbers, a_{ef}, a_{ep}, F and P to a real number. It is undefined if $a_{ef} > F$ or $a_{ep} > P$ or $F = P = 0$.*

Metrics are intended to measure *similarity* between the set of failed tests and the set of tests which execute a statement. The whole idea behind the approach is that *in most cases* there is a positive correlation between execution of buggy statements and failure, and a negative correlation between execution of correct statements and failure. Some metrics measure *dis-similarity* and produce very poor rankings. Typically the bugs are ranked towards the bottom rather than the top; we have observed such behaviour when we have incorrectly translated metrics which are described using different terminology from different application areas. Thus we can put additional constraints on what functions can sensibly, or rationally, be used as metrics.

Definition 2 ((strictly)rational metric) *A metric M is* rational *if it is monotonically increasing in a_{ef} and monotonically decreasing in a_{ep}: if $a'_{ef} > a_{ef}$ then $M(a'_{ef}, a_{ep}, F, P) \geq M(a_{ef}, a_{ep}, F, P)$ and if $a'_{ep} > a_{ep}$ then $M(a_{ef}, a'_{ep}, F, P) \leq M(a_{ef}, a_{ep}, F, P)$, for points where M is defined.*

A metric M is strictly rational *if $a'_{ef} > a_{ef}$ implies $M(a'_{ef}, a_{ep}, F, P) > M(a_{ef}, a_{ep}, F, P)$ and $a'_{ep} > a_{ep}$ implies $M(a_{ef}, a'_{ep}, F, P) < M(a_{ef}, a_{ep}, F, P)$, for points where M is defined.*

There are cases where the correlations are the opposite of what is expected, and some metric which is not rational will perform better than rational metrics. However, there is no way of knowing *a priori* whether we have such a case and *overall* rational metrics perform better. Thus we consider it reasonable to restrict attention to rational metrics in the search for good metrics and when assessing "ideal" performance. Almost all metrics previously used for fault localization are strictly rational. Ample is the only metric which is not rational, because "absolute value" is used. In Naish et al. (2011), a variation of this metric is defined (called "Ample2") which does not use absolute value, and it performs significantly better than Ample.

The Russell metric does not strictly decrease as a_{ep} increases, so it is rational but not strictly rational, and similarly for O whenever $a_{ef} < F$. Metrics

Table 2: Definitions of ranking metrics used

Name	Formula	Name	Formula	Name	Formula
Kulczynski2	$\frac{1}{2}\left(\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ef}}{a_{ef}+a_{ep}}\right)$	Zoltar	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+\frac{10000\,a_{nf}\,a_{ep}}{a_{ef}}}$	Jaccard	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$	Russell	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$	Ample	$\left\lvert\frac{a_{ef}}{a_{ef}+a_{nf}}-\frac{a_{ep}}{a_{ep}+a_{np}}\right\rvert$
Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$	O	$\begin{cases}-1 & \text{if } a_{nf}>0\\ a_{np} & \text{otherwise}\end{cases}$	O^p	$a_{ef}-\frac{a_{ep}}{a_{ep}+a_{np}+1}$
CBILog	$\frac{2}{\frac{1}{c}+\frac{\log(a_{ef}+a_{nf})}{\log a_{ef}}}$, where $c=\frac{a_{ef}}{a_{ef}+a_{ep}}-\frac{a_{ef}+a_{nf}}{a_{ef}+a_{nf}+a_{np}+a_{ep}}$				
Wong3	$a_{ef}-h$, where $h=\begin{cases}a_{ep} & \text{if } a_{ep}\le 2\\ 2+0\cdot 1(a_{ep}-2) & \text{if } 2<a_{ep}\le 10\\ 2\cdot 8+0\cdot 001(a_{ep}-10) & \text{if } a_{ep}>10\end{cases}$				

which are rational but not strictly rational can generally be tweaked so they are strictly rational with no loss of performance in typical cases. O was designed specifically for the case of single bugs, where we know the bug is executed in all failed tests ($a_{ef} = F$ for the bug; we discuss this further in Section 5). If we modify O so it gives a small negative weight to a_{ep} when $a_{ef} < F$ it becomes strictly rational (it produced the same rankings as O^p). This does not affect performance at all for single bug programs and generally improves performance for multiple bug programs. Similarly, if we give a small negative weight to a_{ep} in the Russell metric it also becomes equivalent to O^p, which performs better in nearly all cases (see Naish et al. (2011)). Even if non-strict rational metrics remain of some practical benefit, considering only strict rational metrics leads us to additional theoretical insights.

4 Measuring performance

The most common way of performance measure for spectral fault localization is the rank of the highest-ranked bug, as a percentage of the total number statements; typically, only statements which are executed in at least one test case are counted (Abreu et al. 2006, Wong et al. 2007, Naish et al. 2011). This is often called the *rank percentage*. If a bug is ranked highest, which is the best case, the rank is 1. Here we give a slightly different definition, which we call *rank cost* to avoid confusion, where the best case is zero. This is more convenient for our work, and also when averaging the performance over several programs with different numbers of statements, which is normally done. When bugs and non-bugs are tied in the ranking we assume the bugs are ranked in the middle of all these equally ranked statements. We discuss this more in Section 7; there is some variation in how ties are handled in the literature.

Definition 3 (rank cost) *Given a ranking of S statements, the* rank cost *is*

$$\frac{GT + EQ/2}{S}$$

where GT is the number of correct statements ranked strictly higher than all bugs and EQ is the number of correct statements ranked equal to the highest ranked bug.

For most programs and sets of tests, we cannot expect a buggy statement to be ranked strictly higher than all correct statements, whatever metric is used to produce the ranking. For example, all statements in the same basic block as a bug will be be tied with the bug in the ranking, since they will have the same a_{ef} and a_{ep} values as the bug. Furthermore, there may be other statements with higher a_{ef} and lower a_{ep} values, which must be ranked higher than the bug for all strictly rational metrics. By explicitly considering such statements, we can determine how much of the cost of bug localization could potentially be avoided by choosing a different strictly rational metric, and how much is unavoidable. We define the *unavoidable cost* in a way which makes it easy to compare with the rank cost. We first introduce some additional notation concerning a partial order of statements based on their associated spectra.

Definition 4 ($=^s$, $\le^s$, $<^s$) *For two statements, x and y, with associated spectra:*

- $x =^s y$ *if* $a^x_{ef} = a^y_{ef} \wedge a^x_{ep} = a^y_{ep}$
- $x \le^s y$ *if* $a^x_{ef} \le a^y_{ef} \wedge a^x_{ep} \ge a^y_{ep}$
- $x <^s y$ *if* $x \le^s y \wedge \neg(a^x_{ef} =^s a^y_{ef})$

Definition 5 (unavoidable cost) *Given a set of S statements and corresponding spectra, the* unavoidable cost *is the minimum of UC_b, for all bugs b, where*

- $UC_b = \frac{GT'_b + EQ'_b/2}{S}$
- GT'_b *is the number of correct statements c, such that $b <^s c$,*
- EQ'_b, *the number of correct statements c, such that $b =^s c$.*

Proposition 1 *If $x \le^s y$, any rational metric will rank x below or equal to y. If $x <^s y$, any strictly rational metric will rank x below y.*

Proof Follows from definitions. □

Proposition 2 *For any set of statements, associated spectra and strictly rational ranking metric, the rank cost of the resulting ranking is at least the unavoidable cost.*

Proof If b is the highest ranked bug, the rank cost is UC_b, which is at least the unavoidable cost. □

Proposition 3 *For any set of statements and associated spectra there exists a strictly rational ranking metric such that the rank cost of the resulting ranking is the unavoidable cost.*

Proof Let b be a bug which minimises UC_b. There is no buggy statement b' s.t. $b <^s b'$, otherwise the unavoidable cost would be lower. Consider the following definitions:

$$M(a_{ef}, a_{ep}, F, P) = f(a_{ef} - a^b_{ef}) + f(a^b_{ep} - a_{ep})$$

$$f(x) = \begin{cases} \epsilon x & \text{if } x < 0 \\ 1 + \epsilon x & \text{otherwise,} \end{cases}$$

$$\epsilon = 1/(F + P + 1)$$

M is a strictly rational metric. For the bug b (and all statements c s.t. $c =^s b$) it has value 2. The metric value of a statement c is greater than 2 if and only if $b <^s c$, and all such statements are correct. Thus the rank cost using M is the unavoidable cost. □

Proposition 4 *For any set of statements and associated spectra, the unavoidable cost is the minimum rank cost for any strictly rational metric.*

Proof Follows from Propositions 2 and 3. □

If we do not restrict the class of metrics, there will always be *some* metric which ranks the bug highest and the minimum cost, ignoring ties, will always be zero. Note that a metric which is rational but not *strictly* rational can also have a cost lower than the unavoidable cost. For example, with one passed and one failed test, all statements could be executed in the failed test but the passed test could execute only the buggy statement(s). The unavoidable cost is the highest possible cost, which is close to 1, whereas a rational metric could have all statements tied in the ranking, with a cost of around 0.5. Although theoretically interesting, such examples do not seem to provide strong practical motivation for using metrics which are not strictly rational.

We cannot necessarily expect to achieve a cost as low as the unavoidable cost in practice — it simply gives a lower bound on what we can reasonably expect using this approach to bug localization. If we can achieve the unavoidable cost or very close to it in all cases, we know there is no point in searching for better metrics. If there is a wide gap between the cost we achieve and the unavoidable cost for some buggy programs and sets of test cases we might be able to close the gap with different metrics, but we risk making the situation worse for other programs. There is no single metric which achieves the unavoidable cost for all programs — we cannot swap the order of quantifiers in Proposition 3. Also, the unavoidable cost does not give a lower bound on what can be achieved by other fault localization methods. However, the same methodology could potentially be applied. If, for example, a richer form of spectral data was used, we may be able to find an appropriate unavoidable cost definition for that method.

5 Optimality for single bug programs

In Naish et al. (2011), optimality of metrics is introduced and "single bug" programs are the focus. In order to establish any technical results, we must be clear as to what constitutes a bug, so it is clear if a program has a single bug. In Naish et al. (2011) a bug is defined to be "a statement that, when executed, has unintended behaviour". A programmer may make a single mistake which leads to multiple bugs according to this definition. For example, when coding various formulas which use logarithms, the programmer may use the wrong base for all the logarithms. Also a mistake in a single `#define` directive in a C program can lead to multiple bugs. The `#define` directive is not a statement which is executed and no spectral data is generated for it, but several statements which *use* the macro may behave incorrectly.

In order to understand the fault localization problem better, a very simple model program, with just two if-then-else statements and a single bug is proposed in Naish et al. (2011), along with a very simple way of measuring performance of a metric with a given set of test cases, based on whether the bug is ranked top, or equal top. A set of test cases corresponds to a multiset of execution paths through the program. Performance depends on the multiset, but *overall* performance for T tests is determined by the average performance over all possible multisets of T execution paths. Using a combinatorial argument, the O metric is shown to be "optimal": its overall performance is at least as good as any other metric, for any number of tests. Although O is not strictly rational, there are strictly rational metrics such as O^p which are also optimal, so restricting attention to strictly rational metrics does not reduce potential performance, at least in this case. There are two conditions for a metric to be optimal for this simple model and performance measure (here we show the definition has much wider utility):

Definition 6 (Single bug optimality) *A metric M is* single bug optimal *if*

1. *when $a_{ef} < F$, the value returned is always less than any value returned when $a_{ef} = F$, that is, $\forall F \forall P \forall a_{ep} \forall a'_{ep}$ if $a_{ef} < F$ then $M(a_{ef}, a_{ep}, F, P) < M(F, a'_{ep}, F, P)$, and*
2. *when $a_{ef} = F$, M is strictly decreasing in a_{ep}, that is, if $a'_{ep} > a_{ep}$ then $M(F, a'_{ep}, F, P) < M(F, a_{ep}, F, P)$.*

The first condition is motivated by the fact that for single bug programs, the bug must be executed in all failed tests. Since $a_{ef} = F$ for the bug, statements for which $a_{ef} < F$ are best ranked strictly lower. The second condition is motivated by the fact that the bug tends to have a lower a_{ep} value than correct statements, because some executions of the bug lead to failure, the model program is symmetric with respect to buggy and correct statements, and all possible multisets of executions paths are used to evaluate overall performance. In addition to proving optimality under these very artificial conditions, Naish et al. (2011) conjectured such metrics were optimal for a wider class of models. In addition, empirical experiments were conducted with real programs and the optimal metrics performed better than other proposed metrics when measured using rank percentages.

Here we give an optimality result for single bug programs which does not constrain us to a simplistic program structure or performance measure or a particular distribution of sets of test cases. Indeed, we prove optimal performance for *every* set of test cases, not just overall performance. We obtain this *much* more widely applicable technical result, which also has a much simpler proof, by restricting attention to strictly rational metrics.

Proposition 5 *Given any program with a single bug, any set of test cases and any single bug optimal metric M used to rank the statements, the rank cost equals the unavoidable cost.*

Proof The rank cost is $\frac{GT+EQ/2}{S}$. Since there is a single bug, b, the unavoidable cost is $\frac{GT'_b+EQ'_b/2}{S}$ and $a^b_{ef} = F$. M is single bug optimal so any statement c ranked strictly higher than the bug must have the same a_{ef} value and a strictly lower a_{ep} value (so $b <^s c$) and any statement ranked equal to the bug must have the same a_{ef} value and the same a_{ep} value as the

bug (so $b =^s c$). Thus $EQ = EQ'_b$ and $GT = GT'_b$. □

Proposition 6 *Given any program with a single bug, any set of test cases and any single bug optimal metric M used to rank the statement, the rank cost using M is no more than the rank cost using any other strictly rational metric.*

Proof Follows from Propositions 2 and 5. □

The rank cost is not necessarily the best measure of performance. However, a consequence of this proposition is that for *any* cost measure which is monotonic in the rank cost, single bug optimal metrics have a lower or equal cost than any other rational metric. For example, optimality applies with respect to rank percentages and the simple cost measure of Naish et al. (2011) which only examines the statement(s) ranked (equal) top. Alternatively, a more complex non-linear cost function could be considered desirable, since the time spent finding a bug typically grows more than linearly in the number of different lines of code examined.

Drawing definitive conclusions from empirical experiments such as those of Naish et al. (2011) is normally impossible because the results may be dependent on the set of benchmark programs used, or the sets of test cases, or the details of the performance evaluation method. However, in this case we can use Proposition 6 to remove any doubt that the "optimal" metrics are indeed better than other metrics for single bug programs. Interestingly, there was one case found in Naish et al. (2011) where the optimal metrics did not perform the best overall — when sets of test cases were selected so that the buggy statement in the model program was executed in nearly every test. The only better metric was Russell, which benefits from a large number of ties in such cases, as discussed earlier. It was also noted that Russell performed better than the optimal metrics for some of the empirical benchmark programs, though its overall performance was worse. From Proposition 6 we know such behaviour can only occur for metrics which are not strictly rational.

6 Optimizing metrics for single bugs

O^p was proposed as a metric which was single bug optimal and also expected to perform rather better than O for multiple bug programs. While this is true, experiments have shown that O^p does not perform particularly well for multiple bug programs (Naish et al. 2009). The two conditions for single bug optimality of a metric place no constraint on the relative ordering of statements for which $a_{ef} < F$. *Any* metric can thus be adapted so it becomes optimal for single bug by adding a special case for $a_{ef} = F$ — we just need to ensure it is decreasing in a_{ep} and larger than any other value possible with the same F and P.

Definition 7 (Optimal single bug version) *The* optimal single bug version *of a metric M, denoted $O1(M)$ is defined as follows.*

$$O1(M)(a_{ef}, a_{ep}, F, P) = \begin{cases} K + 1 + P - a_{ep} & \text{if } a_{ef} = F \\ M(a_{ef}, a_{ep}, F, P) & \text{otherwise,} \end{cases}$$

where K is the maximum of $\{M(x, y, F, P) | x < F \wedge y \leq P\}$.

Proposition 7 *$O1(M)$ is single bug optimal for all metrics M.*

Table 3: Description of Siemens + Unix benchmarks

Program	1 Bug	2 Bugs	LOC	Tests
tcas	37	604	173	1608
schedule	8	—	410	2650
schedule2	9	27	307	2710
print_tok	6	—	563	4130
print_tok2	10	10	508	4115
tot_info	23	245	406	1052
replace	29	34	563	5542
Col	28	147	308	156
Cal	18	115	202	162
Uniq	14	14	143	431
Spline	13	20	338	700
Checkeq	18	56	102	332
Tr	11	17	137	870

Proof When $a_{ef} = F$, $O1(M)$ is clearly strictly decreasing in a_{ep} and the value returned is at least $K+1$, since $a_{ep} \leq P$, so it is greater than any value returned when $a_{ef} < F$. □

In practice, many metrics range between 0 and 1 so we could just choose $K = 1$ in all cases for these metrics. A larger fixed value, such as $K = 999999$ is sufficient for all metrics proposed to date unless there are a very large number of test cases.

If $a_{ef} < F$ for a large proportion of statements, $O1(M)$ will produce a similar ranking to M and if M works very well for multiple bug programs, we would expect $O1(M)$ to also work well. Of course, $O1(M)$ will also work as well as any other rational metric for single bug programs.

7 Experimental results

We performed empirical evaluation using a collection of small C programs: the Siemens Test Suite (STS), from the Software Information Repository (Do, Elbaum & Rothermel 2005), plus several small Unix utilities, from Wong, Horgan, London & Mathur (1998). These, particularly STS, are widely used for evaluating spectral ranking methods. Table 3 gives the names of the programs (the first seven are from STS), and the numbers of single bug and two-bug versions, lines of code (LOC) and test cases. A small number of programs in the repository were not used because there was more than one bug according to our definition (for example, a `#define` was incorrect) or we could not extract programs spectra. We used the `gcov` tool, part of the `gcc` compiler suite, and it cannot extract spectra from programs with runtime errors. We generated the two-bug versions from pairs of single-bug versions, eliminating resulting programs if they encountered runtime errors, as in Naish et al. (2009). This collection of two-bug programs is far from ideal. However, most collections of buggy programs have a very strong bias towards single bugs or have only a relatively small number of program versions. Obtaining better benchmarks is clearly a priority.

We conducted experiments to compute the average unavoidable cost and the rank cost for each metric and the single bug optimal version of the metric, for both one and two bug benchmark sets. Table 4 gives the results. The unavoidable costs are given in the second row. The last column of figures uses the optimal single bug version of the metrics. For the single bug benchmark, the optimal single bug version of each metric gives a rank cost of 16.87, which is the unavoidable cost, so we omit these from the table. The original versions of the metrics range in perfor-

Table 4: Unavoidable and rank costs

Benchmark	1 Bug	2 Bug	2 Bug
Unavoidable	16.87	11.72	$O1(\ldots)$
O	16.87	23.75	23.75
O^p	16.87	21.64	21.64
Wong3	17.20	21.34	21.56
Zoltar	17.24	19.32	21.42
Kulczynski2	18.07	18.32	21.24
Ochiai	20.63	18.95	21.18
Wong4	21.23	21.51	21.60
Jaccard	22.65	19.87	21.20
CBILog	25.23	21.04	21.56
Tarantula	26.10	21.91	21.54
Ample	29.17	23.26	21.75
Russell	29.02	30.88	21.82

Table 5: Two bug rank cost wrt $a_{ef} = F$ category

$a_{ef} = F$ for ...	2 Bugs	1 Bug same	1 Bug inverted	No Bug
% of Cases	44	35	11	10
% $a_{ef} = F$	67	51	44	37
Unavoidable	17.55	9.53	2.43	3.25
Kul2	18.62	21.91	4.52	17.80
O1(Kul2)	17.55	20.88	17.51	42.43
O^p	17.55	20.88	17.51	46.50
O	17.55	20.88	17.51	68.04
Russell	32.96	25.40	21.83	48.40

mance for the single bug benchmark set, with O and O^p being the best, and equal to the unavoidable cost, as expected.

For the two bug benchmark set, the unavoidable cost, 11.72, is significantly lower. This is to be expected since it is essentially the minimum of the unavoidable costs of two bugs. However, the rank costs are higher for most metrics, and the best rank cost, 18.32, for Kulczynski2, is significantly higher than the unavoidable cost. Although it cannot be guaranteed, it seems likely better metrics to exist. The optimal single bug versions of the metrics show much less variation in rank cost and, unfortunately, perform significantly worse than the best metrics. We conducted additional experiments to better understand the performance of Kulczynski2 and the single bug optimal metrics.

Table 5 summarises the results. It breaks down the 2-bug benchmark set into four categories: the programs for which $a_{ef} = F$ for both bugs, the programs for which $a_{ef} = F$ for just one bug where O^p and Kulczynski2 rank the bugs in the same order, the programs for which $a_{ef} = F$ for just one bug where O^p and Kulczynski2 rank the bugs in the opposite (inverted) order, and the programs for which $a_{ef} = F$ for no bug. The first row of figures gives the percentages of cases for these four categories. Overall, in 90% of cases at least one bug is executed in all failed tests, which is generally helpful for the single bug optimal metrics. The second row gives the percentages of correct statements for which $a_{ef} = F$ in the four categories. On average, 57% of correct statements are used in all failed tests, so the "special case" in $O1$ actually applies to most statements, and $O1$ affects the ranking more than expected for this benchmark set. This is the main reason why $O1$ performs more poorly than anticipated. The second row gives the average unavoidable cost for each category. It shows significant variation in unavoidable cost and we discuss this further below. The following lines of Table 5 give the average rank cost (percentage) for the different metrics; Kulczynski2 is abbreviated to Kul2.

Table 6: Two bug rank cost wrt $\%a_{ef} = F$

$\%a_{ef} = F$	<20	20–40	40–60	60–80	≥80
%of Cases	10.6	8.1	19.0	57.9	4.2
Kul2	7.05	10.99	17.67	21.33	19.65
O1(Kul2)	5.47	16.08	21.51	24.51	21.89
O^p	6.24	19.17	21.64	24.58	21.75
O	16.70	23.39	23.57	24.96	22.37
Russell	8.35	22.59	26.11	36.26	43.96

The three single bug optimal metrics have equal rank cost for the first three categories since the top-ranked bug has $a_{ef} = F$ and the ranking of all such statements is the same with these metrics. The difference between these metrics and Russell indicates the usefulness of a_{ep} — Russell ranks according to a_{ef} and essentially ignores a_{ep}. In the first category, our treatment of ties is (arguably) unfair to Russell — 67% of statements, including both bugs, are tied at the top of the ranking. Some researchers report the "worst case" (67%) in such situations and assume the bugs are ranked at the bottom of this range (Chilimbi, Liblit, Mehra, Nori & Vaswani 2009); this over-estimation of cost is discussed in Naish, Lee & Kotagiri (2010). Some researchers report both the "best case" (0%) and the "worst case" (Wong et al. 2007). Here we assume both bugs are ranked in the middle of the range, but even this leads to some over-estimation. If both bugs were to appear at a random point amongst these ties, the top-most bug would have an average cost of around 22% rather than 33%. Similarly, in the fourth category, the O metric has both bugs tied with 63% of the correct statements, at the bottom of the ranking, and a fairer treatment of ties would give an average cost of 59% rather than 68%. With the better metrics there are far fewer ties and thus the over-estimation of cost is much less and we doubt it affects any overall conclusions. Our treatment of ties was motivated by much simpler analysis and could be refined further.

Kulczynski2 performs slightly worse (around 1.1%) than O1(Kulczynski2) for the first two categories, which cover the majority (79%) of cases. This is because around 1.1% of correct statements have lower a_{ef} and significantly lower a_{ep} values than the bugs, and they overtake both bugs in the ranking. However, in the third category, Kulczynski2 performs extremely well. Almost half the statements have $a_{ef} = F$ and when a bug with a lower a_{ef} and a_{ep} is placed higher in the ranking, it overtakes nearly all these statements. Although this category account for only 11% of cases, it more than compensates for the cases when correct statements overtake the bugs. Kulczynski2 also performs significantly better than the other metrics in the last category. Russell and all the single bug optimal metrics rank the statements with $a_{ef} = F$ highest, so the rank cost must be at least 37%, whereas Kulczynski2 does even better than its overall performance.

The unavoidable cost figures also underscore the importance of the number of statements which are executed in all failed tests. In the first category, where both bugs are executed in all failed tests, we may intuitively expect bug localization to be easiest and the good metrics should achieve their best performance. However, it actually has the highest unavoidable cost, by a large margin. As well as both bugs being executed in all failed tests, on average, two thirds of correct statements are also executed in all failed tests. Table 6 gives an alternative breakdown of the two-bug performance figures, based on

the percentage of statements which are executed in all failed tests. Most cases fall into the 60–80% range for this benchmark set. Performance for all metrics drops as the percentage increases, except when the percentage is very high. When the percentage is less than 20%, O1(Kulczynski2) performs better than all other metrics.

The good overall performance of Kulczynski2 compared to O1(Kulczynski2) and other single bug optimal metrics is thus strongly linked to the number of statements with $a_{ef} = F$. For larger programs we would expect this proportion to be significantly smaller. We know from Naish et al. (2011) that for the Space benchmark (around 9000 LOC) the rank percentages for the better metrics is around one tenth that of the Siemens Test Suite; this is partly due to a smaller percentage of statements with $a_{ef} = F$. We could also improve overall performance by initially computing this percentage using the spectra for all statements, then using it to select either Kulczynski2, if it is relatively large, or O1(Kulczynski2), for example.

We also note that the tests suites are designed primarily to detect the *existence* of bugs, not find the *location* of bugs. There is a desire for a large coverage of statements. For example the STS tests were designed with the aim of having every statement executed by at least 30 tests. Tests which execute a large percentage of the code are generally better for detecting the existence of bugs but are worse for locating bugs. We are hopeful that with better test selection strategies and larger programs, single bug optimal metrics can be of significant practical benefit.

8 Other related work

In Section 2 we referred to several papers which introduced new metrics for spectral fault localization, or evaluated metrics which had previously been introduced for other domains. Here we briefly review other related work. There are a couple of approaches which post-process the ranking produced which are equivalent to adjusting the metric, similar to our $O1$ function. The post-ranking method of Xie et al. (2010) essentially drops any statement which is not executed in any failed test to the bottom of the ranking. That of Debroy, Wong, Xu & Choi (2010) ranks primarily on the a_{ef} value and secondarily on the original rank. Thus if the original ranking is done with a strictly rational metric, the resulting ranking is the same as that produced by O^p.

Other variations on the statement spectra ranking method described in this paper attempt to use additional and/or different information from the program executions. Execution frequency counts for statements, rather than binary numbers, are used in Lee, Naish & Kotagiri (2010) to weight the different a_{ij} values and in Naish et al. (2009) aggregates of the columns of the matrix are used to adjust the weights of different failed tests. The RAPID system (Hsu, Jones & Orso 2008) uses the Tarantula metric but uses branch spectra rather than statement spectra.

The CBI (Liblit et al. 2005) and SOBER (Liu, Yan, Fei, Han & Midkiff 2005) systems use predicate spectra: predicates such as conditions of if-then-else statements are instrumented and data is gathered on whether control flow ever reaches that point and, if it does, whether the predicate is ever true. CBI uses sampling to reduce overheads but aggregates the data so there are four numbers for each predicate, which are ranked in a similar way to how statements are ranked using statement spectra. SOBER uses frequency counts and a different form of statistical ranking method. The Holmes system (Chilimbi et al. 2009) uses path spectra: data is collected on which acyclic paths through single functions are executed or "reached ", meaning the first statement is executed but not the whole path, and the paths are ranked in a similar way to predicate ranking in CBI. Statement and predicate spectra are compared in Naish et al. (2010), and it is shown that the aggregate data used in predicate spectra methods is more expressive than that used for statements spectra and modest gains in theoretical performance are demonstrated. The data collected for path spectra contains even more information and thus could potentially be used to improve performance further.

9 Conclusion

Spectra-based techniques are a promising approach to software fault localization. Here we have used one of the simplest and most popular variants: ranking statements according to some metric, a function of the numbers of passed and failed tests in which the statement is/isn't executed. We have identified the class of *strictly rational* metrics, which are strictly increasing in the number of failed test executions and strictly decreasing in the number of passed test executions. We have argued that it is reasonable to restrict attention to this class of metrics, and there is no apparent evidence that doing so reduces fault localization performance. Having made this restriction, we can put a lower bound on the cost of fault localization — the "unavoidable cost". No strictly rational metric can achieve a lower cost.

We have shown that *single bug optimal* metrics perform at least as well as any other strictly rational metric, for various reasonable measures of performance, for all programs with a single bug and all sets of test cases. This significantly extends a previous theoretical result and shows that we cannot do any better with this variant of spectral fault localization for single-bug programs. We also showed how any metric can be adapted so it becomes single bug optimal.

Performance of spectral fault localization on multiple-bug programs is much less well understood. We have performed empirical experiments with a variety of metrics on a benchmark set of small two-bug programs. All metrics resulted in costs significantly greater than the unavoidable cost on average. Also, the single bug optimal metrics had significantly greater cost than the best metrics overall. We have identified one reason for this: typically a large proportion of statements are executed in every failed test. We know that this proportion is smaller in benchmarks with larger programs. Also, it may be practical to reduce it further by careful creation and selection of test cases. For the subset of our two-bug benchmark set where less than 20% of statements were executed in all failed tests, the best performance was achieved by the single bug optimal version of a metric known to perform well on multiple bug programs. Overall, there seem reasonable prospects for improving performance when there is a mixture of single- and multiple-bug programs, which is what real fault localization tools are faced with.

References

Abreu, R., Zoeteweij, P. & van Gemund, A. (2006), 'An evaluation of similarity coefficients for software fault localization', *PRDC'06* pp. 39–46.

Chen, M., Kiciman, E., Fratkin, E., Fox, A. & Brewer, E. (2002), 'Pinpoint: Problem determination in

large, dynamic internet services', *Proceedings of the DSN* pp. 595–604.

Chilimbi, T., Liblit, B., Mehra, K., Nori, A. & Vaswani, K. (2009), HOLMES: Effective statistical debugging via efficient path profiling, *in* 'Proceedings of the 2009 IEEE 31st International Conference on Software Engineering', IEEE Computer Society, pp. 34–44.

Dallmeier, V., Lindig, C. & Zeller, A. (2005), Lightweight bug localization with AMPLE, *in* 'Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging', ACM, pp. 99–104.

Debroy, V., Wong, W., Xu, X. & Choi, B. (2010), A Grouping-Based Strategy to Improve the Effectiveness of Fault Localization Techniques, *in* '10th International Conference on Quality Software , 2010. QSIC 2010'.

Do, H., Elbaum, S. & Rothermel, G. (2005), 'Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact', *Empirical Software Engineering* **10**(4), 405–435.

Gonzalez, A. (2007), Automatic Error Detection Techniques based on Dynamic Invariants, Master's thesis, Delft University of Technology, The Netherlands.

Hsu, H., Jones, J. & Orso, A. (2008), RAPID: Identifying bug signatures to support debugging activities, *in* '23rd IEEE/ACM International Conference on Automated Software Enginering, 2008. ASE 2008', pp. 439–442.

Jaccard, P. (1901), 'Étude comparative de la distribution florale dans une portion des Alpes et des Jura', *Bull. Soc. Vaudoise Sci. Nat* **37**, 547–579.

Jones, J. & Harrold, M. (2005), 'Empirical evaluation of the tarantula automatic fault-localization technique', *Proceedings of the 20th ASE* pp. 273–282.

Lee, H. J. (2011), Software Debugging Using Program Spectra , PhD thesis, University of Melbourne.

Lee, H. J., Naish, L. & Kotagiri, R. (2010), Effective Software Bug Localization Using Spectral Frequency Weighting Function, *in* 'Proceedings of the 2010 34th Annual IEEE Computer Software and Applications Conference', IEEE Computer Society, pp. 218–227.

Liblit, B., Naik, M., Zheng, A., Aiken, A. & Jordan, M. (2005), 'Scalable statistical bug isolation', *Proceedings of the 2005 ACM SIGPLAN* **40**(6), 15–26.

Liu, C., Yan, X., Fei, L., Han, J. & Midkiff, S. P. (2005), 'Sober: statistical model-based bug localization', *SIGSOFT Softw. Eng. Notes* **30**(5), 286–295.

Lourenco, F., Lobo, V. & Bação, F. (2004), 'Binary-based similarity measures for categorical data and their application in Self-Organizing Maps', *JOCLAD* .

Naish, L., Lee, H. J. & Kotagiri, R. (2009), Spectral debugging with weights and incremental ranking, *in* '16th Asia-Pacific Software Engineering Conference, APSEC 2009', IEEE, pp. 168–175.

Naish, L., Lee, H. J. & Kotagiri, R. (2010), Statements versus predicates in spectral bug localization, *in* 'Proceedings of the 2010 Asia Pacific Software Engineering Conference', IEEE, pp. 375–384.

Naish, L., Lee, H. J. & Kotagiri, R. (2011), 'A model for spectra-based software diagnosis', *ACM Transactions on software engineering and methodology (TOSEM)* **20**(3).

Ochiai, A. (1957), 'Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions', *Bull. Jpn. Soc. Sci. Fish* **22**, 526–530.

Reps, T., Ball, T., Das, M. & Larus, J. (1997), The use of program profiling for software maintenance with applications to the year 2000 problem, *in* 'Proceedings of the 6th European Conference held jointly with the 5th ACM SIGSOFT', Springer-Verlag New York, Inc. New York, New York, USA, pp. 432–449.

Russel, P. & Rao, T. (1940), 'On habitat and association of species of Anopheline larvae in south-eastern Madras', *J. Malar. Inst. India* **3**, 153–178.

Wong, W. E., Debroy, V. & Choi, B. (2010), 'A family of code coverage-based heuristics for effective fault localization', *Journal of Systems and Software* **83**(2).

Wong, W. E., Qi, Y., Zhao, L. & Cai, K. (2007), 'Effective Fault Localization using Code Coverage', *Proceedings of the 31st Annual IEEE Computer Software and Applications Conference* pp. 449–456.

Wong, W., Horgan, J., London, S. & Mathur, A. (1998), 'Effect of Test Set Minimization on Fault Detection Effectiveness', *Software-Practice and Experience* **28**(4), 347–369.

Xie, X., Chen, T. Y. & Xu, B. (2010), Isolating Suspiciousness from Spectrum-Based Fault Localization Techniques, *in* '10th International Conference on Quality Software , 2010. QSIC 2010'.