Multiple Bug Spectral Fault Localization Using Genetic Programming

Lee Naish, Neelofar, Kotagiri Ramamohanarao
Department of Computing and Information Systems
The University of Melbourne, Victoria 3010, Australia
Email: {lee, kotagiri}@unimelb.edu.au, neelofar.eme@gmail.com

***Abstract*—Debugging is crucial for producing reliable software. One of the effective bug localization techniques is Spectral-Based Fault Localization (SBFL). It locates a buggy statement by applying an evaluation metric to program spectra and ranking program components on the basis of the score it computes. Recently, genetic programming has been proposed as a way to find good metrics. We have found that the huge search space for metrics can cause this approach to be slow and unreliable, even for relatively simple data sets. Here we propose a restricted class of "hyperbolic" metrics, with a small number of numeric parameters. This class of functions is based on past theoretical and empirical results. We show that genetic programming can reliably discover effective metrics over a wide range of data sets of program spectra. We evaluate the performance for both real programs and model programs with single bugs, multiple bugs, "deterministic" bugs and nondeterministic bugs.**

I. Introduction

Debugging software is a very important and resource intensive task in software engineering, with 50% to 80% of software development and maintenance costs attributed to bug fixes [1]. Debugging requires fault localization at the initial stage. This is a tedious process, requiring substantial manual work. Due to this cost, many researchers are studying and proposing effective approaches which involve automated tools to aid fault localization. There are many techniques proposed for fault localization and debugging in literature. Spectral based fault localization (SBFL) techniques [2], [3], [4], [5] have gained much popularity in last few years due to their simplicity. These methods extract program spectra, that is execution profiles of program components (statements, predicates, functions etc.) and information on whether tests pass or fail. The data is used with "risk evaluation" metrics to rank the program components according to how likely they are to be buggy.

Performance of SBFL methods critically depend on the metrics used. Over one hundred metrics have been developed manually and evaluated for SBFL. More recently, genetic programming has been used to automatically develop metrics [6], resulting in many thousands being evaluated. For programs with a single bug, SBFL is relatively easy and we now have a good theoretical understanding [5], [7]. The same is true for locating bugs which are "deterministic" (cause test case failure whenever they are executed) [8]. However, the general case where there are multiple bugs which may not be deterministic is much more complicated. Most SBFL research to date has had a strong bias towards single bug programs, partly due to the available data sets used for evaluation, and tackling the general case is an important priority.

In this paper we make the following contributions:

- We propose a new class of "hyperbolic" metrics which have a small number of numeric parameters whose values can be adjusted to vary the behaviour.
- Depending on the parameter values, hyperbolic metrics can be optimal for single bugs, optimal for deterministic bugs or similar to other metrics known to perform well for some multiple nondeterministic bug benchmarks.
- We describe how genetic programming can be used to find good parameter values for hyperbolic metrics, given training data.
- We use a range of model programs to show that the technique matches the performance of optimal metrics in the single bug and deterministic bug cases, and performs very well in intermediate cases.
- We use a data set from real multiple bug programs to show the technique can out-perform the best previously known metrics on average. Ten-fold cross validation is used to demonstrate the effectiveness of the learning.

The rest of the paper is structured as follows. Section II provides background information on SBFL (some material is based on [8] without additional citation), including theoretical results which motivate our approach, and gives the definitions of selected metrics from the literature that we use for comparison. Section III motivates and defines the class of hyperbolic metrics and Section IV briefly describes how we use genetic programming to learn parameter values. Section V describes experiments and their results, using spectral data from both model programs and real programs. Section VI briefly reviews some additional related work and Section VII concludes.

II. Background

SBFL 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 [9], [4], [10], [5], [11], [7], [8], 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 —

TABLE I
STATEMENT SPECTRA WITH TESTS $T_1 \ldots T_5$

	T_1	T_2	T_3	T_4	T_5	ef	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	$F=2$	$P=3$

TABLE II
FORMULAS FOR SEVERAL RISK EVALUATION METRICS

Metric	Formula
O^p [5]	$ef - \frac{ep}{ep+np+1}$
O^d [8]	$ep - \frac{ef}{ef+nf+1}$
Zoltar [13]	$\frac{ef}{ef+nf+ep+\frac{10000nf\times ep}{ef}}$
Kulczynski2 [11]	$\frac{1}{2}\left(\frac{ef}{ef+nf}+\frac{ef}{ef+ep}\right)$
Ochiai [4]	$\frac{ef}{\sqrt{(ef+nf)(ef+ep)}}$
Tarantula [2]	$\frac{\frac{ef}{ef+nf}}{\frac{ef}{ef+nf}+\frac{ep}{ep+np}}$

$\langle ef, ep, nf, np \rangle$, where the first letter indicates whether the statement was executed (e) or not (n) and the second indicates whether the test passed (p) or failed (f). We use F and P to denote the total number of tests which fail and pass, respectively. Clearly, $nf = F - ef$ and $np = P - ep$ and it is sometimes convenient to use F, P, ef (or nf) and ep (or np) rather than all four ij values. Table I gives an example binary matrix of execution data and binary vector containing the test results. This data allows us to compute F, P and all the ij values.

Metrics, which are numeric functions, can be used to rank the statements. Most commonly they are defined in terms of the four 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 ef values and relatively low ep. Table II gives definitions of the established metrics used here and cites where they were first used for SBFL. More than 150 metrics are evaluated in [12]; our small selection is justified below. 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 may well modify the ranking due to other considerations, and checking correctness generally cannot be done by a single statement at a time. Evaluation of different ranking methods generally ignores such refinement and just depends on where the bug(s) appear in the ranking. Here we use a common measure, "rank percentage" which is the rank of the top-ranked bug, expressed as a percentage of the program size. Results are averaged over all buggy programs in the benchmark set. Statements which are not executed in any test are ignored.

Most evaluation of SBFL has a strong bias to programs with a single bug. This problem is now reasonably well understood. O^p was proposed in [5] and proven optimal with respect to a single bug "model" program. This optimality result was strengthened in [7] to arbitrary single bug programs and test sets by restricting attention to "strictly rational" metrics, which are those whose value strictly increases in ef when ep is fixed and strictly decreases in ep (or increases in np) when ef is fixed. Metrics such as O^p which rank statements primarily on their ef value and use np to break ties when the ef values are equal are single bug optimal. In [8] it was shown that metrics such as O^d which rank statement primarily on np and break ties using ef are optimal for programs with only deterministic bugs (which cause failure whenever they are executed).

Between these two extreme cases we may have multiple bugs, some of which are not deterministic, which is the norm in large software systems. There is little theorectical understanding of this more general case and constructing metrics which perform well seems to be a very challenging problem. Any given bug is typically not executed in all failed tests and not all tests in which it is executed will fail. Metrics which are derived for or perform well for the single bug (or deterministic bug) case often do not perform well in the general case. It was argued in [8] that no single metric will perform well in all situations, but using information such as the likely number of bugs and how consistently their execution causes failure, we may be able to construct specialised metrics which perform well. Unfortunately, there is a paucity of good data sets to evaluate performance in the general case.

In our experiments here we include optimal metrics for the two extreme cases, O^p and O^d. Zoltar is close to optimal for deterministic bugs and also performs well for multiple bug programs [11]. Kulczynski2 and Ochiai perform extremely well for the multiple-bug data sets of [11] which we use here. A version of Ochiai is proved to be the best on benchmarks from the Software-artifact Infrastructure Repository(SIR) [12]. Tarantula was the first metric used for SBFL and though it performs relatively poorly for many data sets a small adjustment makes it optimal for deterministic bugs [8] and it performs reasonably well towards this extreme.

There has been some recent success using genetic programming to construct metrics. Metrics which are nearly optimal for the single bug case have been constructed automatically [6]. However, in our experiments we have failed to reproduce this behaviour reliably and to our knowledge, standard ways of assessing machine learning techniques such as 10-fold cross validation have not been applied and learning good metric for the general case is significantly harder.

III. THE HYPERBOLIC METRIC CLASS

This section motivates and defines the class of "hyperbolic" metrics we propose. Rather than a single fixed formula, we use a formula with several additional numeric parameters. The general idea is to be able pick different parameter values so the resulting formula performs well for different data sets. Later we describe how we use genetic programming to choose the parameter values. Using machine learning to find a small number of numeric parameters rather than a complete formula

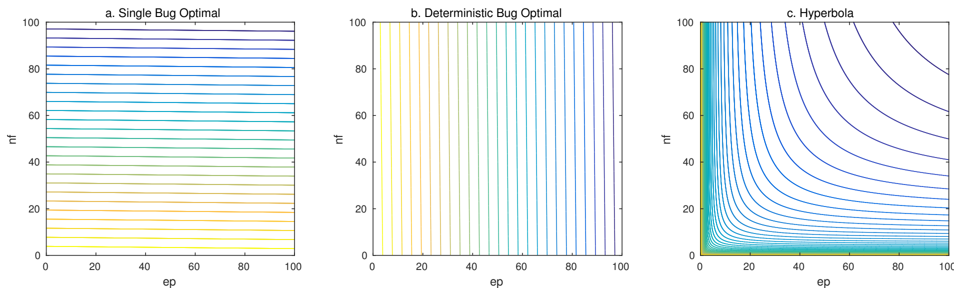

Fig. 1. Contour plots for O^p (a), O^d(b) and $\frac{1}{nf} + \frac{1}{ep}$ (c).

makes the learning task much simpler and, we have found, more efficient and reliable.

A. Motivation for Hyperbolic Metrics

Since F and P are fixed for a given set of test outcomes, a metric defined in terms of F, P, ef (or nf) and ep (or np) can be viewed as a surface in three dimensions. In [8] it was noted that plotting the contours of a metric over its rectangular domain ($F \times P$) gives useful insights. There is a 1:1 relationship between contours and the rankings produced by a metric. If we plot metric contours using ep and nf as the axes then strictly rational metrics have contours with finite strictly negative gradients, with the contours close to $(0, 0)$ corresponding to the highest metric values. If all gradients are sufficiently close to zero the metric is single bug optminal, like O^p, whereas if all gradients are sufficiently large negative numbers the metric is deterministic bug optminal, like O^d — see Figure 1(a),(b). Figure 1(c) shows contours which are hyperbolas with negative gradients. The contours at the bottom right of this plot are like those of O^p whereas those at the top left are like those of O^d. Thus by selecting part of the domain, hyperbolas can be optimal for either of the extremes.

Between these extremes, the gradients of the hyperbolic contours increase (get closer to zero) as ep increases. At the bottom left there is a very sharp increase and as we proceed up and right the increase is more gradual. If we "zoom in" sufficiently in the top right the contours are close to straight lines with a gradient of -1. In experiments on multiple bug programs, metrics such as Kulczynski2 and Ochiai have performed best overall [11]. The contours of these metrics also have decreasing gradients and are similar to the hyperbolic contours for selected parts of the domain — see Figure 2(a), (b) and (c). Note that the contours at the bottom left correspond to the top of the ranking and are thus the most important for performance of a metric. Thus the motivation for using metrics with hyperbolic contours is they can be optimal in the two extreme cases we have theoretical results for, and similar to the best metrics we know of in other cases.

B. Hyperbolic Metrics

We now define the class of metrics used. There are four adjustments to the simple formula. The first is to scale both the nf and ep values to the range 0–1 (the aim of this is to help limit the range of values for the parameters introduced in the next steps). There are several ways scaling can be done: divide by F and P respectively, divide by $F+P$ or divide by $nf+np$ and $ef+ep$ respectively. Our choice is based on performance for one data set rather than theoretical analysis. The second adjustment is to add a parameter K_1 to the scaled nf value. A positive K_1 value translates the domain upwards and a large enough value results in a deterministic bug optimal metric if other things remain unchanged. The third adjustment is to add a parameter K_2 to the scaled ep value. A positive K_2 value translates the domain to the right and a large enough value results in a single bug optimal metric if other things remain unchanged (that is, K_1 is relatively small). If K_1 and K_2 are small, contour gradients change abruptly whereas if they are both large the gradients change slowly. Lastly, we multiply the ep term by parameter K_3. This allows us to effectively stretch or compress the domain in a horizontal direction. With large identical K_1 and K_2 values the contours are close to straight lines, but their gradient can be adjusted using K_3. The overall formula for our class of hyperbolic metrics is as follows:

$$\frac{1}{K_1 + \frac{nf}{nf+ef}} + \frac{K_3}{K_2 + \frac{ep}{ep+ef}}$$

IV. Finding Parameter Values Using Genetic Programming

Yoo el al. propose to use genetic programming to generate metrics directly from spectral data [6]. In our efforts to reproduce the results, we found that the results are not reliably good, especially in the case of multiple bugs. In some runs good metrics were found but in many cases they were not. The search space is huge and the choice of GP operators often makes it hard to gradually improve the metrics. For example, subtracting a good metric from another good metric often results in a very bad metric. Even our attempts to constrain the search to more sensible metrics, such as those which are strictly rational, were not particularly successful. In contrast, learning three numeric parameter values is a much simpler task, and a small change in parameter values generally results in a relatively small change in metric performance — the objective function is more "smooth" in some sense. We re-used the genetic programming software to learn the hyperbolic metric parameter values K_1, K_2 and K_3 from training data and applied the resulting metric to localize bugs.

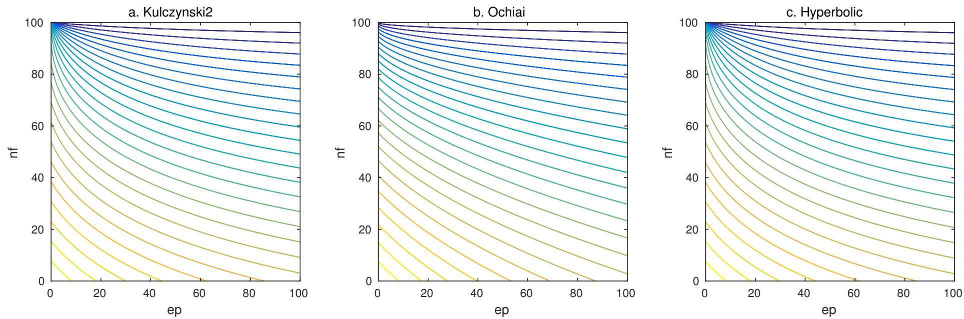

Fig. 2. Contour plots for Kulczynski2 (a), Ochiai (b) and hyperbolic metrics(c).

In our experiments we used JGAP — an open source Java based framework for genetic programming. There are many ways in which the learning can be adjusted. For the results presented here we used a population size of 1000 and stopped after 100 generations. GP is configured with a mutation operator with the rate of 0.9. The three parameters of the hyperbolic metrics were used as terminal symbols and we fixed the range of K_1 and K_2 to be 0–100 and K_3 to be 0–2. The GP operators increased and decreased the parameters within these ranges. We found that for different runs on the same data there was quite a wide variation of parameter values found, though the ratio of K_1 and K_2 was more stable.

V. Experimental Evaluation

To evaluate our technique we used data from model programs as described in [5] and real data from the Siemens Test Suite [14] and small Unix programs. We describe the experiments using models first.

A. Model program experiments

The model-based approach is explained in detail in [5]. An advantage of using models is that large data sets with precisely controlled parameters can be generated. Here we use six different model programs, each with four statements, where execution of correct statements is statistically independent of test case failure but execution of buggy statements is correlated to varying degrees. In the first model, only the first statement is a bug (so O^p is optimal) and execution of the bug leads to test case failure 20% of the time. In models 2–6 the first two statements are buggy, each of which cause failure in 20%, 40%, 60%, 80% and 100% of cases where they are executed, respectively (thus the last model has only deterministic bugs and O^d is optimal). The first two statements are modelled using ten execution paths, five of which execute the statement and a number of those lead to failure, dependent on the model. The other two statements are each modelled using just two execution paths, one of which executes the statement. The models are designed so the relative discrimination of ef versus ep drops as we go from model 1–6, and this affects what metric is best to use for each of these models. For our experiments we used 100000000 test sets, each with 15 passed and 5 failed tests. Learning was done on 50000 instances.

Table III shows the comparison of average rank percentage for learned hyperbolic metrics and previously established metrics on model data explained above. The best results for each line are shown in bold. As expected, O^p and O^d perform best for the first and last models, respectively — these are known to be optimal. The hyperbolic metrics match this optimal performance. They also do better than all other metrics evaluated for the models with 40% and 60% bug consistencies. For the fifth model the performance is very good but not the best and for the second model it is not particularly good. The best performance for these two models are given by O^d and O^p, respectively. Hyperbolic metrics can mimic these two metrics by appropriate choice of parameter values, so we can conclude the less than best performance is due to the learning method rather than the hyperbolic metric class itself. We believe that by making adjustments to the learning method it should be possible to learn good hyperbolic metrics for these two model, to make hyperbolic metrics achieve the best performance in all the models examined.

B. Real program experiments

For Siemens test suite and Unix, we used the data of [11], where versions of the programs with two bugs were created by combining two single bug versions. We used total of 1262 two bug versions of Cal, Checkeq, Col, replace, print_tokens, Spline, tcas, tot_info, Tr and Uniq for our experiments.

Table IV shows comparison of average rank percentages of various metrics on the Siemens Test Suite and subset of Unix. The first ten rows show the results for 10-fold validation. For each validation run we randomly divided the 1262 programs into training and testing partitions with 90% and 10% of the programs, respectively. We used the training partition to find values of hyperbolic metric parameters and used the testing partition to find the average rank percentages for the trained hyperbolic metric and the other metrics. This approach is commonly used to evaluate machine learning techniques. The testing data is separated from the training data to avoid “over fitting” and there are multiple runs to evaluate consistency.

The hyperbolic metrics outperform other metrics in most of the runs and on average. Note that [11] evaluated performance of over 80 metrics for two and three bug versions of the Siemens test suite and Unix benchmarks and Kulczynski2 and Ochiai were the best performing metrics. Thus beating these

TABLE III
AVERAGE RANK PERCENTAGES FOR MODELS WITH DIFFERENT BUG CONSISTENCIES

# Bugs	Consistency	O^p	O^d	Ochiai	Zoltar	Kul2	Tarantula	Hyperbolic
1	20%	**26.12**	48.84	26.64	26.13	26.20	30.22	**26.12**
2	20%	**28.97**	34.94	29.00	**28.97**	**28.97**	30.13	29.50
2	40%	28.48	30.36	28.12	28.43	28.36	28.31	**28.01**
2	60%	28.06	27.04	27.08	27.83	27.60	26.77	**26.53**
2	80%	27.92	**25.57**	26.01	27.01	26.50	25.82	25.62
2	100%	29.52	**26.74**	26.75	**26.74**	**26.74**	30.50	**26.74**

TABLE IV
VALIDATION OF LEARNING HYPERBOLIC METRICS FOR REAL TWO-BUG PROGRAMS

Run #	O^p	O^d	Ochiai	Zoltar	Kul2	Tarantula	Hyperbolic
1	8.5589	12.9314	8.2348	8.2368	7.8830	9.3710	**7.5796**
2	8.2981	12.7456	7.4348	7.3441	**7.0153**	8.9626	7.1798
3	8.7780	12.4552	8.7744	7.9651	8.0189	9.1145	**7.4434**
4	9.8355	11.9914	7.9703	8.1220	7.9094	8.4635	**7.4619**
5	8.9519	11.1911	6.8386	7.7156	7.2334	7.8279	**6.8245**
6	9.2180	10.7376	6.5247	7.3386	6.5262	7.6621	**6.4787**
7	8.7144	12.2496	8.0000	7.5995	**7.2740**	9.1989	7.4502
8	9.1017	11.9229	8.0081	8.7738	8.0727	8.4518	**7.6873**
9	7.3316	12.0082	6.6867	7.0915	6.7641	7.9456	**6.5009**
10	8.5498	11.2670	**7.3264**	7.8023	7.4553	8.3404	7.4905
Average	8.7338	11.9500	7.5799	7.7989	7.4152	8.5338	**7.2097**
P Value	0.0020	0.0020	0.0098	0.0020	0.0488	0.0020	-

metrics for this data set is an impressive achievement. Each run evaluates only 10% of the data set and results therefore vary between runs. The last line of the table shows the p-value for the Wilcoxon signed rank test, comparing the results for the hyperbolic metrics with the other metrics. Its shows we can be confident that the learned hyperbolic metrics perform better than all the other metrics except Kulczynski2 and Ochiai. For these two metrics, more runs would be required in order to achieve a p-value less than 0.05 (the normal standard for statistical significance), but we can say with high confidence the method achieves excellent results on real data. In other experiments we have used all the data for both training and testing, and found hyperbolic metrics which perform better than Kulczynski2 and Ochiai for the whole data set. For such experiments there is no question of statistical significance, but from the machine learning perspective one can question both reliability and the possibility of over fitting.

VI. OTHER RELATED WORK

Landsberg et al. [12] evaluate 157 different similarity metrics from the literature for SBFL and show how many are equivalent for ranking purposes. The Pattern-Similarity metric [**?**], and a simpler equivalent metric, PattSim2 = $-nf \times ep$, have hyperbolic contours identical to those in Figure 1(c). Furthermore, in all these metrics small "prior constants" are added to the spectral values to avoid division by zero etc. These are not needed for PattSim2 since there is no division, but the resulting metric is of the form $-(nf+K_1)(ep+K_2)$. These two constants effectively translate the contours vertically and horizontally in the same way as K_1 and K_2 in our hyperbolic metrics, and make the metric strictly rational. By manually selecting the two constant values, this metric was found to perform better than all others from the literature for a benchmark set which included the Siemens programs plus several significantly larger programs with multiple bugs. Our work differs in that we have three parameters rather than two, we scale the spectral values to the range 0–1 and we find the constants automatically, using machine learning techniques. Landsberg et al. also experimented with adapting metrics so they are single bug optimal, in the same way as proposed in [7]. The single bug optimal version of Ochiai performed the best of all metrics considered.

Slicing and Dicing are considered as one of the oldest techniques in debugging and fault localization. Slicing refers to the piece of program code that affects the value of any variable while dicing is the part of program, which appears in one slice but not in another. These approaches narrow down the program part, which is more likely to be buggy so that developer can concentrate on a small part of the code for fault localization [11][15].

Mutant based fault localization (MUSE) identifies the fault or bug by utilizing the information obtained by mutating the faulty and correct statement [16]. The technique uses the intuition that if a faulty statement is mutated, it will cause more tests to pass than average and if a correct statement is mutated, it will cause more tests to fail then average. These intuitions are the basis of MUSE [17][18][19].

State based approaches aim to localize the bugs by observing the changing state of the program and identifying the failure inducing circumstances [20][21][22]. Failure Inducing Circumstances refer to the input to the test cases which causes it to fail e.g. program input (particular URL input fails the web browser), User Interaction (keystroke of the user causing the program to fail) and changes to the program code.

There are many studies found that test reduction and test selection could help in improving the performance of fault localization technique. Recently, many machine learning and

data mining approaches are proposed in this area which can be used together with spectra-based approaches. Most of the work is done in test case clustering in this domain [23][24][25].

Most of the formulas or metrics used in Spectral based fault localization are not designed specifically for debugging. Jaccard for example used for biological classification for the first time and Ochiai used in marine zoology [26]. Tarantula was the first metric designed for spectral debugging [9]. Few other widely used metrics are Ample [27], Zoltar [13], Wong [10] etc.

Naish et al. propose optimal metrics for single bug [5] and deterministic bug programs [8] which are empirically proved to be the best metrics in these areas. They, however, do not perform equally well on multiple bug data.

There are few techniques available in literature for multiple bug problem which are combination of spectral based with machine learning or model based approaches. Some of these are given below.

James et al. present a clustering approach for debugging in parallel in presence of multiple bugs. Using fault localization information from program execution and behaviour models, they develop a technique that automatically partitions the failing test cases into clusters that target different faults. These clusters are called fault focusing clusters. Each fault focusing cluster is then combined with all the passing test cases to get a specialized test suite that targets a single fault. These specialized test suits can then be assigned to different developers who can work in parallel for debugging and localizing bugs [23].

Abreu el al. present a multi fault localization technique called BARINELL by combining spectral based fault localization and model based reasoning. Model based approaches are more accurate as compared to spectral fault localization but due to their computational complexity they are very expensive for large applications. BARINELL, however, uses effective candidate selection process that reduces it's complexity and make it better candidate for large programs as well [28].

Wong et al. propose a crosstab-based statistical fault localization technique(CBT). The technique uses statement based coverage information. A comparison has been made between CBT and Tarantula and results prove CBT better then tarantula. The technique is claimed to be effectively applicable for multiple bug programs [29].

VII. Conclusion and Future work

Performance of SBFL is strongly influenced by the choice of metric used to estimate the likelihood that a given program component is buggy. We have proposed the class of "hyperbolic" metrics, named after the shape of the contours in plots of the metrics. The class has a small number of numeric parameters, the values of which can be determined by machine learning from training data. We have shown that learned hyperbolic metrics can perform as well or better than previously discovered metrics in a wide range of situations. Using model programs we have demonstrated that these metrics can achieve optimal performance for programs with a single bug and with deterministic bugs (where bug execution always leads to test case failure) — the two extreme cases we have theoretical results for. In a range of other model programs, with two bugs which cause failure with varying consistency, the results are also good but indicate the machine learning method we use (a form of genetic programming) has potential for improvement. Using data from small real programs seeded with two bugs the learned hyperbolic metrics out-performed the best previously know metrics on average.

One clear area for further improvement is in the learning component. We plan to experiment with different genetic programming parameters and applying other machine learning techniques. Given that we are only learning a small number of numeric values, the power of genetic programming is probably not needed. We are confident that the learning can be made more accurate, reliable and efficient. Validation of the approach on other data sets is also a priority, particularly programs which are larger and have more than two bugs.

We further plan to add parameters to Ochiai and Kulczynski2 and see if we can use learning to improve these metrics.

Acknowledgements

We would like to thank Jason (Hua Jie) Lee, who generated the real program data set we used and continues to be involved in discussions on this research area.

References

[1] J. S. Collofello and S. N. Woodfield, "Evaluating the effectiveness of reliability-assurance techniques," *Journal of systems and software*, vol. 9, no. 3, pp. 191–195, 1989.

[2] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th international conference on Software engineering*. ACM, 2002, pp. 467–477.

[3] B. Liblit, "Cooperative Bug Isolation," Ph.D. dissertation, University of California, 2004.

[4] R. Abreu, P. Zoeteweij, and A. van Gemund, "An evaluation of similarity coefficients for software fault localization," *PRDC'06*, pp. 39–46, 2006.

[5] L. Naish, H. J. Lee, and R. Kotagiri, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, August 2011.

[6] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *Search Based Software Engineering*. Springer, 2012, pp. 244–258.

[7] L. Naish, H. J. Lee, and R. Kotagiri, "Spectral debugging: How much better can we do?" in *35th Australasian Computer Science Conference (ACSC 2012), CRPIT Vol. 122*. CRPIT, 2012.

[8] L. Naish and H. J. Lee, "Duals in spectral fault localization," in *Proceedings of ASWEC 2013*. IEEE Press, 2013.

[9] J. Jones and M. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," *Proceedings of the 20th ASE*, pp. 273–282, 2005.

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

[11] H. J. Lee, "Software Debugging Using Program Spectra," Ph.D. dissertation, University of Melbourne, 2011.

[12] D. Landsberg, H. Chockler, D. Kroening, and M. Lewis, "Evaluation of measures for statistical fault localisation and an optimising scheme," in *Fundamental Approaches to Software Engineering*. Springer, 2015, pp. 115–129.

[13] A. Gonzalez, "Automatic Error Detection Techniques based on Dynamic Invariants," Master's thesis, Delft University of Technology, The Netherlands, 2007.

[14] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[15] H. Agrawal, J. Horgan, S. London, and W. Wong, "Fault localization using execution slices and dataflow tests," *Software Reliability Engineering*, pp. 143–151, 1995.

[16] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 153–162.

[17] M. Papadakis and Y. Le Traon, "Using mutants to locate" unknown" faults," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 691–700.

[18] V. Debroy and W. E. Wong, "Combining mutation and fault localization for automated program debugging," *Journal of Systems and Software*, vol. 90, pp. 45–60, 2014.

[19] M. Papadakis and Y. Le Traon, "Effective fault localization via mutation analysis: a selective mutation approach," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 1293–1300.

[20] A. Zeller, "Isolating cause-effect chains from computer programs," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, p. 10, 2002.

[21] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *Software Engineering, IEEE Transactions on*, vol. 28, no. 2, pp. 183–200, 2002.

[22] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 272–281.

[23] J. Jones, J. Bowring, and M. Harrold, "Debugging in parallel," *Proceedings of the ISSTA*, pp. 16–26, 2007.

[24] H. Hsu, J. Jones, and A. Orso, "RAPID: Identifying bug signatures to support debugging activities," in *23rd IEEE/ACM International Conference on Automated Software Engineering, 2008. ASE 2008*, 2008, pp. 439–442.

[25] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*. IEEE, 2007, pp. 137–146.

[26] A. Ochiai, "Zoogeographic studies on the soleoid fishes found in japan and its neighbouring regions," *Bull. Jpn. Soc. Sci. Fish*, vol. 22, no. 9, pp. 526–530, 1957.

[27] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight bug localization with ample," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM, 2005, pp. 99–104.

[28] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "Spectrum-based multiple fault localization," in *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*. IEEE, 2009, pp. 88–99.

[29] W. E. Wong, V. Debroy, and D. Xu, "Towards better fault localization: A crosstab-based statistical approach," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 42, no. 3, pp. 378–396, 2012.