Evolving Human Competitive Spectra-Based Fault Localisation Techniques

Shin Yoo

University College London

Abstract. Spectra-Based Fault Localisation (SBFL) aims to assist debugging by applying risk evaluation formulæ (sometimes called suspiciousness metrics) to program spectra and ranking statements according to the predicted risk. Designing a risk evaluation formula is often an intuitive process done by human software engineer. This paper presents a Genetic Programming (GP) approach for evolving risk assessment formulæ. The empirical evaluation using 92 faults from four `Unix` utilities produces promising results[1]. Equations evolved by Genetic Programming can consistently outperform many of the human-designed formulæ, such as Tarantula, Ochiai, Jaccard, Ample, and Wong1/2, up to 6 times. More importantly, they can perform equally as well as Op2, which was recently proved to be optimal against `If-Then-Else-2` (`ITE2`) structure, or even outperform it against other program structures.

1 Introduction

Despite the advances in software testing techniques, faults still prevail in many software systems and debugging remains a hard task. Fault localisation aims to guide the programmer towards the program statement that contains the fault, using the information observed during test execution.

Spectra-Based Fault Localisation (SBFL) is a class of fault localisation techniques that uses program spectra (i.e., a summary of program's execution trace) to predict the likelihood of each program statement containing the fault [1,13,14]. The key element is a risk evaluation formula, or sometimes a suspiciousness metric, that converts the program spectra to relative risk value for each statement. SBFL subsequently ranks program statements according to the relative risk: the programmer can investigate the source code following the rank order. The intuition is that the faulty statement will be high in the ranking, reducing the number of statements the programmer has to check.

The performance of a SBFL technique depends mostly on the quality of the risk evaluation formula. The majority of the existing, widely studied formulæ are either inherited from other fields [12, 18] or designed by human intuition [5, 14, 15, 17, 24]: there is no guarantee that one formula is optimal for all classes of faults. Designing a risk evaluation formula that performs universally

[1] The program spectra data used in the paper, as well as the complete empirical results, are available from: `http://www.cs.ucl.ac.uk/staff/s.yoo/evolving-sbfl.html`.

G. Fraser (Ed.): SSBSE 2012, LNCS 7515, pp. 244–258, 2012.

well against all possible combination of various program structures, test suites, and potential locations of faults remains a difficult task for a human. The only available methodology is that of trial and error: to design intuitively and evaluate empirically. Recent work includes efforts to design a risk evaluation formula that can be proven to be *optimal*, but only with respect to the case that the fault is contained within a specific program structure [17].

We presents an alternative approach: to evolve risk evaluation formulæ from program spectra directly. Using program spectra from test executions and known fault locations, we use Genetic Programming (GP) to evolve risk evaluation formulæ. By using a non-biased sample of known faults as the training data for GP, we try to obtain formulæ that are effective against various program structures. It is true that the evolved formulæ will be only as good as the input data for the GP. However, compared to proving optimality of risk evaluation formulæ against all possible program structures, providing common program structures that contain fault is a significantly easier task. In fact, this bears a strong resonance to the mantra of Search Based Software Engineering (SBSE) [10], namely:

> It is easier to compare solutions and choose the better one than to design a perfect solution from the scratch.

This paper introduces an evolutionary approach to designing risk evaluation formulæ for SBFL. GP uses program spectra from four `Unix` utilities from Software Infrastructure Repository [6] and the location information of 92 injected faults. The contributions of this paper are as follows:

- The paper presents the first evolutionary approach to generating risk evaluation formulæ for SBFL. All existing formulæ have been manually designed, often relying only on intuition. The introduced approach is evaluated with empirical studies, using test spectra data from real world `Unix` utilities.
- The empirical evaluation shows that GP-generated risk evaluation formulæ can outperform those designed by human. GP-generated formulæ can outperform some of the widely studied formulæ. Moreover, GP-generated formulæ can perform equally well or even better than an existing formula that has been proven to be optimal against a specific program structure. The equal performance provides evidence that GP can match the human design efforts; the outperformance provides evidence that GP can produce formulæ that are very effective for structures against no proof of optimality is currently available.
- All data used for the empirical study in the paper have been made available online to encourage replication and further research.

The rest of the paper is structured as follows. Section 2 introduces the concept of Spectra-Based Fault Localisation and the role of risk evaluation formulæ. Section 3 explains how we formulate the design of risk evaluation formulæ using Genetic Programming. Section 4 describes the experimental setup. Section 5 presents and analyses the results from the empirical evaluation. Section 6 presents the related work. Section 7 concludes and discusses future work.

2 Spectra-Based Fault Localisation

2.1 Basic Concept

Fault location aims to reduce the cost of debugging by guiding the process of searching for the location of the fault in the program. Various techniques rely on different software artefact to aid the developer: delta debugging [27,28] uses the cause-effect chain between the test input and the failure to guide the developer to the specific part of test input that causes the failure. Program Dependence Graph (PDG) has been used to construct a causal inference model for the location of fault [4].

One branch of fault localisation techniques that have attracted a significant amount of interest is Spectra-Based Fault Localisation (SBFL). Program spectra is a summary of a set of program executions [11]. For many of the SBFL techniques, we observe the execution of the test suite for System Under Test (SUT). Suppose SUT has n statements, and the test suite contains m test cases: the program spectrum for SBFL can be described as a matrix of n rows and 4 columns. Each row corresponds to individual statement of SUT, and contains four counters: (e_p, e_f, n_p, n_f). Counter e_p and e_f represent the number of times the corresponding program statement has been executed by tests, with pass and fail as a result respectively. Similarly, n_p and n_f represent the number of times the corresponding program statement has *not* been executed by tests, with pass and fail as a result respectively[2]. SBFL techniques subsequently use a risk evaluation formula, which is a formula based on the four counters, to predict the relative risk of each statement containing the fault. Compared to the case in which the developer investigates the structural elements in the order from s_1 to s_9, the ranking according to Tarantula produces 66.66% reduction in debugging effort (i.e. the developer will encounter s_7 6 elements earlier.

$$\text{Tarantula} = \frac{\frac{e_f}{e_f+n_f}}{\frac{e_p}{e_p+n_p} + \frac{e_f}{e_f+n_f}} \tag{1}$$

For example, Table 1 illustrates how the Tarantula metric [13], defined in Equation 1, can be applied to a small exemplar program spectrum. Suppose the structural element s_7 contains the fault. The coverage relationship between structural elements and the given test suite $T = \{t_1, t_2, t_3\}$ is given in the second column, with the corresponding test results. The Spectrum column contains the program spectrum data for T; the column Tarantula contains the resulting risk evaluation metric values. Finally, the column Rank contains the ranking of structural elements according to the Tarantula metric values. The faulty statement, s_7, is assigned with the highest Tarantula metric value, and therefore ends up in the first place.

2.2 Risk Evaluation formulæ

The effectiveness of a SBFL technique is determined by the risk evaluation formula, such as Equation 1. All existing formulæ are generated by human [17].

[2] The sum of e_p, e_f, n_p, and n_f should be m.

Table 1. Motivating Example: the faulty statement s_7 achieves the 1st place when ranked according to the Tarantula risk evaluation formula in Eq 1.

Structural Elements	Test t_1	Test t_2	Test t_3	Spectrum e_p	e_f	n_p	n_f	Tarantula	Rank
s_1	•			1	0	0	2	0.00	9
s_2	•			1	0	0	2	0.00	9
s_3	•			1	0	0	2	0.00	9
s_4	•			1	0	0	2	0.00	9
s_5	•			1	0	0	2	0.00	9
s_6	•		•	1	1	0	1	0.33	4
s_7 (faulty)		•	•	0	2	1	0	1.00	1
s_8	•	•		1	1	0	1	0.33	4
s_9	•	•	•	1	2	0	0	0.50	2
Result	P	F	F						

Table 2 contains several of the most widely studied formulæ. Interestingly, Jaccard [12] and Ochiai [18] were first studied in Botany and Zoology respectively but have been subsequently studied in the context of fault localisation [1, 17]. Tarantula was originally developed as a visualisation method [14, 15] but also increasingly considered as an SBFL risk evaluation formula independent from visualisation [13,19]. AMPLE [5] and three different versions of Wong metric [24] have been introduced specifically for fault localisation.

Op1 and Op2 metrics are recent additions to SBFL techniques that showed an interesting research direction: these metrics are proven to produce optimal ranking, as long as the fault is located in a specific program structure (two consecutive `If-Then-Else` blocks, called `ITE2`) [17]. Although the proof does not guarantee that Op1 and Op2 are optimal for all locations of faults (and not just limited to `ITE2`), the empirical evaluation showed that both Op1 and Op2 are very strong formulæ.

Table 2. Risk Evaluation formulæ

Name	Formula	Name	Formula
Jaccard [12]	$\frac{e_f}{e_f+n_f+e_p}$	Ochiai [18]	$\frac{e_f}{\sqrt{(e_f+n_f)\cdot(e_f+e_p)}}$
Tarantula [15]	$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_p}{e_p+n_p}+\frac{e_f}{e_f+n_f}}$	AMPLE [5]	$\lvert\frac{e_f}{e_f+n_f}-\frac{e_p}{e_p+n_p}\rvert$
Wong1 [24]	e_f	Wong2 [24]	$e_f - e_p$
Wong3 [24]	$e_f - h$, where $h = \begin{cases} e_p & \text{if } e_p \leq 2 \\ 2+0.1(e_p-2) & \text{if } 2 < e_p \leq 10 \\ 2.8+0.001(e_p-10) & \text{if } e_p > 10 \end{cases}$		
Op1 [17]	$\begin{cases} -1 & \text{if } n_f > 0 \\ n_p & \text{otherwise} \end{cases}$	Op2 [17]	$e_f - \frac{e_p}{e_p+n_p+1}$

2.3 Designing Risk Evaluation formulæ

This subsection discusses why Genetic Programming can be an ideal tool for designing risk evaluation formulæ.

Difficulties in Formal Approaches: Although the optimality proof of Naish et al. [17] presents a complete approach towards designing a risk evaluation formula, it will require significant human efforts to provide optimality proofs for a wider range of program structures. Moreover, SBFL can be applied to other testing criteria such as the existing work in concurrency testing [19], for which the possibility of optimality proof remains unknown.

Data-driven Iteration: Barring the formal proof of optimality, the most intuitive process of designing a risk evaluation formula would be an iterative modification of a candidate formula, against as a wide range of spectra datasets as possible, until its performance reaches an acceptable level. Not only the amount of data will burden the human designer, but this process also is, in fact, how GP operates, i.e., a data-driven, systematic trial-and-error.

Providing Insights: The goal of using GP for designing risk evaluation formulæ does not have be to replace human designs completely. It can actually be a powerful tool that the human software engineer can use to explore the design space with, to identify building blocks of better formulæ, and to gain insights into the specific domain under consideration.

2.4 Research Questions

Based on the discussions in Section 2.3, this paper investigates the performance of GP-designed risk evaluation formulæ for structural SBFL.

- **RQ1. Effectiveness:** How much debugging effort can be reduced by the GP-generated risk evaluation formulæ compare to existing human-designs?
- **RQ2. Design Space:** How much diversity is observed among the GP-generated formulæ? Does GP re-discover human-designed formulæ? How much problem does GP-bloat cause?
- **RQ3. Insights:** Are there design insights we can obtain by analysing the GP-generated formulæ? Do more complex formulæ perform better? Are certain spectra elements more important than the others?

RQ1 directly concerns the performance of the GP-evolved risk evaluation formulæ. It will be answered by performing statistical hypothesis testing to the reduction of debugging effort produced by GP and human generated formulæ. **RQ2** aims to investigate how much diversity can be allowed in the design space. It will be answered by comparing the GP-generated formulæ, both the whole and its parts, to the existing ones. Finally, **RQ3** is about the design insights we can expect to learn by evolving risk evaluation formulæ using GP.

3 Genetic Programming for SBFL

3.1 Representation

We use a simple tree-based representation and a set of simple operators on the ground that they can sufficiently represent most of the existing risk evaluation formulæ. Table 3 presents the GP operators used in the paper. Addition

(`gp_add`), subtraction (`gp_sub`), and multiplication (`gp_mul`) do not require any treatment, because these operations cannot result in numerical exceptions. The division operator `gp_div` will return 1 when division by zero error is expected. Similarly, the square root operator `gp_sqrt` uses the absolute value of the given input. For terminal symbols, we use the program spectra data $\{e_p, e_f, n_p, n_f\}$, as well as one constant, 1.

Table 3. List of GP operators

Operator Node	Definition
`gp_add(a, b)`	`a + b`
`gp_sub(a, b)`	`a - b`
`gp_mul(a, b)`	`ab`
`gp_div(a, b)`	1 if $b = 0$, $\frac{a}{b}$ otherwise
`gp_sqrt(a)`	$\sqrt{\lvert a \rvert}$

3.2 Fitness Function

The aim of risk evaluation formula is not only to assign high risk value to the faulty statement, but also to ensure that the assigned high risk value results in a high ranking of the faulty statement. That is, the performance of a risk evaluation formula is measured by the relative position of the faulty statement when ranked by the formula.

In literature, this relative measurement is often referred to as the Expense metric [21], which is a normalised ranking of the faulty statement. Given a risk evaluation formula τ, a program p, and a fault b in p, the Expense metric E is calculated as in Equation 2:

$$E(\tau, p, b) = \frac{\text{Ranking of } b \text{ according to } \tau}{\text{Number of statements in } p} * 100 \tag{2}$$

Expense is an *a-posteriori*, evaluative metric: it can be calculated only when the faulty statement is known. Because we are evolving a risk evaluation formula from locations of the known faults, Expense can be used as a fitness function. To avoid over-fitting to the location of a specific fault, we calculate Expense metric for a candidate formula using multiple faults from different programs and take the average as the fitness function. For a set of n known faults $B = \{b_1, \ldots, b_n\}$ from corresponding n programs $P = \{p_1, \ldots, p_n\}$, the fitness value of a candidate risk evaluation formula τ is calculated as follows:

$$\text{fitness}(\tau, B, P) = \frac{1}{n} \sum_{i=1}^{n} E(\tau, p_i, b_i) \text{ (to be minimised)} \tag{3}$$

Depending on the risk evaluation formula, multiple statements may get assigned the same risk evaluation value and, thereby, tie in the ranking. Because it is not immediately clear what will be the appropriate tie-breaker for a candidate formula, we do not break ties and assign the most conservative ranking to all tied statements, which is equal to the sum of the number of the tied statements and the number of statements ranked before them [21,26]. In the context of the

fault localisation, this means that we assume the developer has to check all of the tied statements to locate the fault.

4 Experimental Setup

4.1 Subjects

Table 4 lists the subject programs whose faults are studied in the paper: `flex` (a lexical analyzer), `grep` (a text-search utility), `gzip` (a compression utility), and `sed` (a stream text editor). All four programs are obtained from Software Infrastructure Repository (SIR) [6] along with their test suites. Statement coverage information was collected using the GNU profiler, `gcov` version 4.3.2 on Linux version 2.6.27. We use the test suites provided by SIR.

Table 4. Subject Programs from SIR

Subject	Number of Tests	Lines of Code	Executable Lines of Code	Number of Faults
`flex`	567	12,407–14,244	3,393–3,965	47
`grep`	199	12,653–13,363	3,078–3,314	11
`gzip`	214	6,576– 7,996	1,705–1,993	18
`sed`	360	8,082–11,990	1,923–2,172	16

SIR provides a total of 219 (both real and seeded) faults across the five versions of the four subject programs [6]. We exclude 35 of these faults because these faults were unreachable when compiled for the experimental environment, and additional 92 faults because these are not detected by the chosen test suites. This leaves 92 faults, the distribution of which are listed in Table 4.

4.2 Implementation and Configuration

We use `pyevolve` [20] version 0.6 to implement the Genetic Programming. The algorithm was executed using Python runtime version 2.7.3. The population size was iteratively configured to 40. The initialisation uses the ramping method with the maximum tree depth of 4: the maximum tree depth was chosen to be able to express the most of the existing formalæ. The stopping criterion is a fixed run of 100 generations. The GP is configured with a rank selection operator, a single point crossover operator with the rate of 1.0, and a subtree replacement mutation operator with the rate of 0.08.

4.3 Evaluation

The Genetic Programming algorithm was repeated 30 times to cater for its stochastic nature. Each individual run of the GP uses a random sample of 20 faults out of 92 to evolve a risk evaluation formula; the remaining 72 faults are reserved for evaluation purposes.

We use Vargha & Delaney's A-test [22] to compare the Expense metric values of GP-evolved formulæ to those of existing ones. Vargha & Delaney's A-test is a non-parametric statistical test for determining stochastic superiority/inferiority of one sample X over another sample Y: the value of A is the probability that a

Table 5. Comparison of mean Expense for 72 faults in evaluation sets. Rows in bold correspond to GP-results that perform as well as or better than any human-designed formulæ.

ID	GP	Op1	Op2	Ochiai	AMPLE	Jacc'd	Tarant.	Wong1	Wong2	Wong3
GP01	5.73	9.20	5.30	32.66	10.96	6.10	15.06	22.24	17.10	6.63
GP02	12.04	9.67	5.72	32.60	11.91	6.63	14.92	23.45	19.49	8.92
GP03	14.46	11.35	6.11	29.99	12.18	6.99	15.68	23.55	18.55	8.85
GP04	7.80	9.70	4.46	30.98	8.83	5.03	13.88	22.62	14.64	6.33
GP05	9.35	11.04	5.80	29.95	10.63	6.42	14.46	23.15	18.54	8.53
GP06	12.15	11.11	5.87	28.02	12.51	6.79	15.35	23.12	16.70	7.01
GP07	8.93	11.18	5.94	29.53	12.19	6.85	14.81	23.88	19.74	8.68
GP08	**6.32**	**10.23**	**6.34**	**30.91**	**11.67**	**7.04**	**16.21**	**23.54**	**19.94**	**9.05**
GP09	9.66	10.58	5.33	31.56	11.40	6.17	14.06	22.58	18.31	8.20
GP10	**6.31**	**11.55**	**6.31**	**29.83**	**12.51**	**7.16**	**15.79**	**22.99**	**19.74**	**8.56**
GP11	**5.83**	**11.07**	**5.83**	**33.52**	**12.12**	**6.69**	**16.77**	**22.05**	**18.16**	**6.96**
GP12	12.09	8.84	6.23	32.15	11.65	7.02	16.65	22.91	19.42	9.09
GP13	**5.11**	**9.05**	**5.11**	**31.67**	**10.27**	**5.90**	**15.92**	**22.03**	**17.00**	**6.69**
GP14	9.91	8.52	5.91	31.69	11.10	6.55	15.88	23.15	18.10	8.65
GP15	5.62	9.54	5.59	33.02	10.23	6.19	15.16	23.85	17.17	8.44
GP16	6.79	8.32	5.71	30.52	10.74	6.41	14.60	23.06	18.36	8.42
GP17	7.67	11.46	6.22	33.62	12.06	6.98	16.85	22.44	17.94	8.59
GP18	9.42	10.78	5.54	34.17	11.46	6.33	15.45	22.17	17.46	8.14
GP19	6.42	9.01	5.11	31.28	10.18	5.78	15.03	22.84	15.26	7.79
GP20	**5.69**	**10.93**	**5.69**	**29.34**	**10.88**	**6.38**	**15.23**	**23.41**	**19.30**	**8.42**
GP21	10.17	10.13	6.24	29.82	10.86	6.89	15.70	23.01	19.85	9.43
GP22	7.58	8.50	5.91	28.06	10.46	6.60	13.67	23.25	18.60	8.63
GP23	6.14	10.76	5.52	30.86	10.57	6.16	14.69	21.77	16.90	7.25
GP24	9.18	10.15	6.21	28.74	12.53	7.10	15.76	23.41	20.16	8.35
GP25	9.34	10.19	6.29	32.56	12.36	7.18	17.59	22.63	20.19	9.48
GP26	**6.38**	**11.62**	**6.38**	**32.83**	**12.27**	**7.25**	**18.28**	**23.77**	**16.18**	**7.69**
GP27	9.75	8.53	5.89	33.28	12.01	6.85	16.42	22.99	19.23	7.81
GP28	5.56	9.18	5.25	30.02	11.18	6.15	13.52	22.86	17.17	6.85
GP29	7.16	10.12	6.17	34.17	12.83	7.14	17.00	22.94	20.18	8.88
GP30	10.68	9.10	5.14	30.02	10.17	5.78	14.49	22.79	17.09	8.34

single subject taken randomly from group X has higher/lower value than another single case randomly taken from group Y. For $A(X > Y)$, the value of A closer to 1 represents a higher probability of $X > Y$, 0 a higher probability of $X < Y$, and 0.5 no effect (i.e., $X = Y$).

However, the statistical interpretation of the results should be treated with caution. There is no guarantee that the studied programs and faults are representative of all possible programs and faults and, therefore, it is not clear whether they are legitimate *samples* of the entire group. On the other hand, if the cost of designing risk evaluation formulæ is significantly reduced by the use of GP, the possibility of project-specific formalæ should not be entirely ruled out.

5 Results and Analysis

5.1 Effectiveness

Table 5 contains the mean Expense values for all 30 GP-evolved formulæ and human-designed formulæ in Table 7[3]. Each row reports the mean Expense values

[3] The complete results for individual faults are available from: `http://www.cs.ucl.ac.uk/staff/s.yoo/evolving-sbfl.html`.

from 72 faults in corresponding evaluation set. Note that the evaluation set differs between GP runs, as the training set is sampled randomly to avoid bias.

Rows in bold typefaces represent the GP runs that produced formulæ that performed as well as or better than all of the human-designed formulæ: this was observed 6 times out of 30 runs. The human-designed formula that performs the best is Op2; its relative performance confirms the trend observed in the previous work [17]. In 5 runs out of the aforementioned 6 (GP10, GP11, GP13, GP20, and GP26), GP-evolved formulæ always produce the same ranking, and subsequently the same Expense value, as Op2 and outperforms all other human-designed formulæ. In GP8, the remaining one run, the GP-evolved formula does not completely agree with Op2, but the mean Expense value from GP-evolved formula is lower than that from Op2.

The biggest improvement over human-designed formula is found in GP13 between GP and Ochiai: the expense from GP-evolved formula is less than one sixth of that from Ochiai. In fact, Ochiai, Tarantula, Wong1, and Wong2 are outperformed by GP in all runs. Based on this observation, we focus our comparative statistical analysis to the better performing formulæ: Op1, Op2, Ample, Jaccard, and Wong3. Table 6 presents the statistical analysis of the comparison between GP-evolved formulæ and the five human-designed formulæthat can produce Expense value below 10. Column A contains Varghar & Delaney's A test results, with which we test whether GP-based Expenses are lower than those based on existing formulæ. Column Count contains a tuple $(x/y/z)$: x is the number of faults for which GP produces lower Expense than the corresponding human-designed formula, y is the number of faults for which the Expense values are equal, and finally z is the number of faults for which GP produces higher Expense[4]. Combined with the A-test, these numbers provide a summary of how GP-evolved formulæ compare to existing ones.

The overall trend in Table 6 is that the results from A-test are mostly close to 0.5, suggesting that there is no overall difference in Expense values produced by GP and other formulæ overall. This confirms the results in Table 5: GP-evolved formulæ perform as equally well as human-designed formulæ. However, observing the details in Column Count reveals that there exist faults for which GP outperforms existing formulæ and vice versa. Figure 1 provides a scatterplot with fault-by-fault comparison between some of GP-evolved formulæ and other metrics[5]. GP08 produces lower Expense values for only 3 faults and higher values for 10, but the mean Expense of GP08 is still lower(Table 5). GP11 performs exactly as well as Op2 (i.e., the rankings are identical). For GP15 and GP27, the story is mixed: GP15 comfortably outperforms Tarantula, but GP27 produces Expense values significantly higher than those from Jacard for a few faults.

Considering that the aim of our approach is to *design* a formula that will be repeatedly used, we argue that it is not unrealistic to apply GP to existing program spectra data repeatedly and choose the best performing outcome: the cost of multiple GP execution will be amortised over the saved effort in fault

[4] Therefore $x + y + z$ is equal to 72, i.e., the size of the evaluation set.

[5] Scatterplot comparisons for all GP-evolved formulæ are also available online.

Table 6. Vargha & Delaney's A-test between GP and the better performing formulæ. Rows in bold correspond to GP-results that perform as well as or better than any human-designed formulæ.

ID	Op1		Op2		AMPLE		Jaccard		Wong3	
	A	Count	A	Count	A	Count	A	Count	A	Count
GP01	0.51	3/63/6	0.50	2/64/6	0.53	25/46/1	0.51	22/47/3	0.50	7/60/5
GP02	0.38	9/16/47	0.35	8/16/48	0.39	22/8/42	0.36	19/10/43	0.39	13/15/44
GP03	0.45	4/52/16	0.42	0/56/16	0.45	21/33/18	0.42	20/33/19	0.44	5/54/13
GP04	0.37	11/9/52	0.34	7/9/56	0.37	16/9/47	0.34	10/9/53	0.37	9/9/54
GP05	0.49	6/53/13	0.47	4/53/15	0.49	19/42/11	0.47	15/41/16	0.50	10/51/11
GP06	0.49	4/48/20	0.47	3/48/21	0.50	6/56/10	0.47	5/48/19	0.48	6/46/20
GP07	0.46	6/38/28	0.44	2/42/28	0.47	19/30/23	0.44	14/31/27	0.46	7/38/27
GP08	**0.51**	**3/59/10**	**0.50**	**3/59/10**	**0.54**	**25/47/0**	**0.51**	**26/46/0**	**0.52**	**9/54/9**
GP09	0.50	6/51/15	0.48	2/55/15	0.50	17/43/12	0.48	17/42/13	0.50	4/53/15
GP10	**0.52**	**4/67/1**	**0.50**	**0/71/1**	**0.53**	**23/45/4**	**0.50**	**24/44/4**	**0.51**	**8/63/1**
GP11	**0.52**	**4/68/0**	**0.50**	**0/72/0**	**0.53**	**24/45/3**	**0.50**	**23/46/3**	**0.52**	**5/67/0**
GP12	0.48	2/53/17	0.47	2/53/17	0.50	19/46/7	0.48	19/45/8	0.49	2/55/15
GP13	**0.51**	**3/69/0**	**0.50**	**0/72/0**	**0.52**	**23/47/2**	**0.50**	**22/48/2**	**0.50**	**6/66/0**
GP14	0.50	2/59/11	0.49	2/59/11	0.52	20/49/3	0.49	18/49/5	0.50	5/56/11
GP15	0.51	3/63/6	0.50	3/63/6	0.51	21/48/3	0.50	21/48/3	0.52	10/56/6
GP16	0.50	2/58/12	0.49	2/58/12	0.53	22/47/3	0.50	17/50/5	0.52	10/53/9
GP17	0.48	5/50/17	0.45	1/53/18	0.49	22/33/17	0.46	18/35/19	0.48	8/49/15
GP18	0.50	4/61/7	0.48	0/65/7	0.50	21/42/9	0.48	20/43/9	0.50	2/64/6
GP19	0.50	4/49/19	0.49	3/49/20	0.52	20/46/6	0.50	16/46/10	0.51	8/49/15
GP20	**0.52**	**4/68/0**	**0.50**	**0/72/0**	**0.52**	**23/46/3**	**0.50**	**23/46/3**	**0.53**	**9/63/0**
GP21	0.50	3/61/8	0.49	3/61/8	0.51	22/46/4	0.49	20/46/6	0.51	9/55/8
GP22	0.50	2/67/3	0.49	0/69/3	0.52	22/47/3	0.50	20/49/3	0.52	5/65/2
GP23	0.52	4/63/5	0.50	0/67/5	0.52	23/45/4	0.50	19/47/6	0.52	5/64/3
GP24	0.51	3/56/13	0.50	3/56/13	0.52	20/50/2	0.50	19/49/4	0.51	6/54/12
GP25	0.48	11/46/15	0.47	8/47/17	0.50	17/37/18	0.48	18/36/18	0.50	12/43/17
GP26	**0.52**	**4/68/0**	**0.50**	**0/72/0**	**0.52**	**23/46/3**	**0.50**	**22/47/3**	**0.51**	**5/67/0**
GP27	0.51	2/58/12	0.50	2/58/12	0.52	21/51/0	0.50	11/51/10	0.51	6/54/12
GP28	0.52	3/60/9	0.51	3/60/9	0.53	22/50/0	0.51	21/49/2	0.52	8/57/7
GP29	0.51	6/45/21	0.49	5/45/22	0.52	19/41/12	0.50	18/39/15	0.52	11/42/19
GP30	0.50	3/60/9	0.49	1/62/9	0.50	18/46/8	0.49	17/46/9	0.51	4/59/9

localisation. Therefore, we answer **RQ1** positively: GP-evolved risk evaluation formulæ can reduce debugging effort more effectively than many of human-designed formulæ, sometimes over 6 times. For many faults, GP-evolved formulæ perform as equally well as the best known formula, Op2. Finally, for some faults, GP-evolved formulæ can outperform even Op2.

5.2 Design Space

Table 7 contains the GP-evolved formulæ in their refined forms. The original solutions were refined by removing syntactic bloats (such as $n_f - n_f$) and improving readability. Explicit bloats were only observed only twice among the 30 formulæ. Since we are evolving formulæ rather than programs, GP-trees do not contain non-reachable nodes. Therefore, it is not clear whether any subcomponents of evolved formulæ can be definitely labelled as bloats, apart from the explicit, syntactic ones.

The GP-evolved formulæ show strong diversity. There is only one formula that is evolved twice by the GP: both GP14 and GP24 evolved $e_f + \sqrt{n_p}$. The same subcomponent is found in GP02, GP22, and GP28. Finally, a similar pattern,

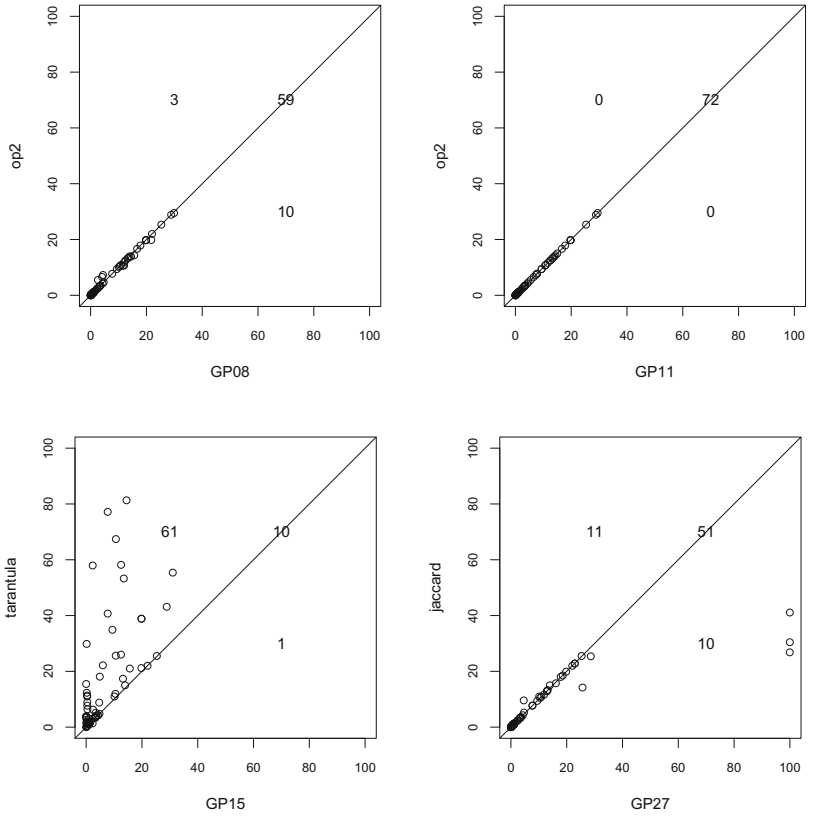

Fig. 1. Scatterplot comparisons of Expense for faults in evaluation set. Each dot represents a fault: the x-axis represents Expense produced by GP-evolved formula, and the y-axis by the specified formula. The solid line represents $y = x$: dots above the line correspond to faults which GP-evolved formulæ can rank higher. The upper two plots show that GP can perform equally or better than Op2. The lower left plot shows that GP can outperform Tarantula for most of the studied faults; the lower right plot shows a mixed results for GP against Jaccard.

$(ae_f^x + bn_p^y)$, where $a, b \in \mathbf{I}, x, y \in \{\frac{1}{2}, 1, \frac{3}{2}, 2, 3\}$, is also frequently observed as in GP01/09/12/21 (which contain $e_f + n_p$), **GP08** $(2e_f + 3n_p)$, **GP11**/22/25/**26** $(ae_f^2 + \sqrt{n_p})$, GP16 $(e_f^{\frac{3}{2}} + n_p)$, and GP18 $(e_f^3 + 2n_p)$. Interestingly, both $e_f + n_p$ and $\sqrt{e_f + n_p}$ are studied in existing literature [17]. However, GP did not rediscover these two metrics in their exact forms; rather, GP evolved variations of these formulæ as parts of larger formulæ. Apart from this, GP did not rediscover any of the existing formulæ.

To answer **RQ2**, the level of diversity observed in GP-evolved formulæ suggests the possibility that there may exist risk evaluation formulæ that are different from, but at least as effective as, the existing formulæ designed by the human. The observation made in Section 5.1, i.e., the fact that some GP-evolved formulæ can

outperform existing ones for certain faults, provides further evidence that there may exist more effective formulæ for various program structures other than `ITE2`. However, the existence of common subcomponents suggest that a hybrid design approach may be even more successful: such an approach would introduce existing formulæ or partially-designed subcomponents into the GP population to assist the evolution.

5.3 Insights

Analysis of GP-evolved formulæ in Table 7 suggests that the most significant program spectra element, with respect to the faults we have studied, is e_f, i.e., the number of times a statement has been executed by failing tests. In all of the 8 GP-evolved formulæ that are equally as effective as Op2 in Table 5, e_p is the element that is either the only component proportional to the risk evaluation value, or the component that is the most dominant. The discussion of common subcomponent in Section 5.2 suggests that n_p is perhaps the second most significant element. Similarly, the least significant element appears to be n_f.

These observations do confirm our intuitions about the relationship between program spectra elements and fault localisation. A statement that contains fault will display a relatively higher e_f value (i.e., frequently covered by failing tests) and a relatively lower n_p value (i.e., less frequently covered by passing tests). In fact, human-designed formulæ such as Wong1/2/3 and Op2 are also designed to translate higher e_f and lower n_p values to higher rankings.

However, there are also some new design insights that can be gained by observing GP-evolved formulæ, which provide answers to **RQ3**. Most interestingly, it appears that ratio-type subcomponents (such as the ratio of a statement being covered by failing tests in Tarantula formula, $\frac{e_f}{e_f+n_f}$) are not necessarily required for a well performing formula: polynomials of spectra elements often seem to be sufficient. Similarly, the results achieved by polynomials of spectra elements suggests that specific constants, such as those found in Wong3, may not be necessary for designing a well performing formula.

6 Related Work

Various Spectra-Based Fault Localisation techniques have been developed to reduce the cost of debugging. One of the most widely studied risk evaluation formula, Tarantula, was initially developed as a visualisation aid for debugging process [14, 15]: subsequently, it has been studied independently from the visualisation [13, 19, 26]. Other notable formulæ include the family of Wong metrics [24], Statistical Bug Isolation (SBI) [16], and AMPLE [5]. Recently, Naish et al. provided an optimality proof against a specific program structure (`ITE2`: two consecutive `If-Then-Else` blocks) for their proposed metrics, Op1 and Op2 [17]. Naish et al. also provides an empirical evaluation of their metrics against a wide range of other formulæ, albeit using a set of relatively small subject programs. All existing metrics have been designed by human; this paper present the first

Table 7. GP-evolved risk evaluation formulæ. Trivial bloats, such as $n_f - n_f$, were removed.

ID	Refined Formula	ID	Refined Formula
GP01	$e_f(n_p + e_f(1 + \sqrt{e_f}))$	GP16	$\sqrt{e_f^{\frac{3}{2}} + n_p}$
GP02	$2(e_f + \sqrt{n_p}) + \sqrt{e_p}$	GP17	$\frac{2e_f+n_f}{e_f-n_p} + \frac{n_p}{\sqrt{e_f}} - e_f - e_f^2$
GP03	$\sqrt{\lvert e_f^2 - \sqrt{e_p}\rvert}$	GP18	$e_f^3 + 2n_p$
GP04	$\sqrt{\lvert\frac{n_p}{e_p-n_p} - e_f\rvert}$	GP19	$e_f\sqrt{\lvert e_p - e_f + n_f - n_p\rvert}$
GP05	$\frac{(e_f+n_p)\sqrt{e_f}}{(e_f+e_p)(n_p n_f+\sqrt{e_p})(e_p+n_p)\sqrt{\lvert e_p-n_p\rvert}}$	**GP20**	$2(e_f + \frac{n_p}{e_p+n_p})$
GP06	$e_f n_p$	GP21	$\sqrt{e_f + \sqrt{e_f + n_p}}$
GP07	$2e_f(1 + e_f + \frac{1}{2n_p}) + (1 + \sqrt{2})\sqrt{n_p}$	GP22	$e_f^2 + e_f + \sqrt{n_p}$
GP08	$e_f^2(2e_p + 2e_f + 3n_p)$	GP23	$\sqrt{e_f}(e_f^2 + \frac{n_p}{e_f} + \sqrt{n_p} + n_f + n_p)$
GP09	$\frac{e_f\sqrt{n_p}}{n_p+n_p} + n_p + e_f + e_f^3$	GP24	$e_f + \sqrt{n_p}$
GP10	$\sqrt{\lvert e_f - \frac{1}{n_p}\rvert}$	GP25	$e_f^2 + \sqrt{n_p} + \frac{\sqrt{e_f}}{\sqrt{\lvert e_p-n_p\rvert}} + \frac{n_p}{(e_f-n_p)}$
GP11	$e_f^2(e_f^2 + \sqrt{n_p})$	**GP26**	$2e_f^2 + \sqrt{n_p}$
GP12	$\sqrt{e_p + e_f + n_p - \sqrt{e_p}}$	GP27	$\frac{n_p\sqrt{(n_p n_f - e_f)}}{e_f+n_p n_f}$
GP13	$e_f(1 + \frac{1}{2e_p+e_f})$	GP28	$e_f(e_f + \sqrt{n_p} + 1)$
GP14	$e_f + \sqrt{n_p}$	GP29	$e_f(2e_f^2 + e_f + n_p) + \frac{(e_f-n_p)\sqrt{n_p e_f}}{e_p-n_p}$
GP15	$e_f + \sqrt{n_f + \sqrt{n_p}}$	GP30	$\sqrt{\lvert e_f - \frac{n_f-n_p}{e_f+n_f}\rvert}$

GP-based approach to the design of risk evaluation formulæ, reformulating it as a predictive modelling based on GP. Machine learning techniques have been also applied to fault localisation work, but the aim was to classify failing tests together rather than to identify the location of the fault directly [23].

Although SBFL originally started as a debugging aid for human developers, the technique is increasingly used to enable other automated Search-Based Software Engineering (SBSE) techniques. Goues et al. use SBFL to identify the parts of a program that needs to be automatically patched [7]. Yoo et al. use SBFL to measure the Shannon entropy of fault locality, so that the test suite can be prioritised for faster fault localisation [25]. GP may be able to help these techniques by evolving SBFL techniques with a specific set of characteristics, improving the synergy between predictive modelling and SBSE even further [9].

Other approaches towards fault localisation include slicing [2], consideration of test similarity [3,8], delta debugging [27,28], and causal inference [4]. While this paper only concerns the spectra-based approach, the positive results suggest that GP may be successfully employed to evolve a wider range of fault localisation techniques.

7 Conclusion

This paper reports the first application of Genetic Programming to evolving risk evaluation formulæ for Spectra-Based Fault Localisation. We use a simple tree-based GP to evolve risk evaluation formulæ that take program spectra elements as terminals. Empirical evaluation based on 92 different faults from four `Unix` utilities shows three important findings. First, GP-evolved formulæ can outperform widely studied human-designed formulæ by up to 5.9 times. Second, GP-evolved formulæ can perform optimally against the `ITE2` program structure,

for which existing formulæ, Op1 and Op2, have been proven to be optimal. Finally, GP-evolved formulæ can outperform Op1 and Op2 for certain studied faults.

Future work will include the use of more sophisticated GP representation (so that GP can evolve conditional formulæ as in Wong3), the inclusion of elements other than program spectra (e.g., code churn, dependency, or data-flow information), and the investigation of the possibility for the evolution of project-specific formalæ.

References

1. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, pp. 89–98. IEEE Computer Society (2007)
2. Agrawal, H., Horgan, J., London, S., Wong, W.: Fault localization using execution slices and dataflow tests. In: Proceedings of IEEE Software Reliability Engineering, pp. 143–151 (1995)
3. Artzi, S., Dolby, J., Tip, F., Pistoia, M.: Directed test generation for effective fault localization. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA 2010, pp. 49–60. ACM, New York (2010)
4. Baah, G.K., Podgurski, A., Harrold, M.J.: Causal inference for statistical fault localization. In: Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA 2010), pp. 73–84. ACM Press (July 2010)
5. Dallmeier, V., Lindig, C., Zeller, A.: Lightweight bug localization with ample. In: Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging, AADEBUG 2005, pp. 99–104. ACM, New York (2005)
6. Do, H., Elbaum, S.G., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering 10(4), 405–435 (2005)
7. Goues, C.L., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Proceedings of the 34th International Conference on Software Engineering, pp. 3–13 (2012)
8. Hao, D., Zhang, L., Pan, Y., Mei, H., Sun, J.: On similarity-awareness in testing-based fault localization. Automated Software Engineering 15, 207–249 (2008)
9. Harman, M.: The relationship between search based software engineering and predictive modeling. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, pp. 1–13. ACM Press, New York (2010)
10. Harman, M., Jones, B.F.: Search based software engineering. Information and Software Technology 43(14), 833–839 (2001)
11. Harrold, M.J., Rothermel, G., Wu, R., Yi, L.: An empirical investigation of program spectra. In: Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 1998), pp. 83–90. ACM, New York (1998)
12. Jaccard, P.: Étude comparative de la distribution florale dans une portion des Alpes et des Jura. Bulletin del la Société Vaudoise des Sciences Naturelles 37, 547–579 (1901)

13. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th International Conference on Automated Software Engineering (ASE 2005), pp. 273–282. ACM Press (2005)
14. Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, pp. 467–477. ACM, New York (2002)
15. Jones, J.A., Harrold, M.J., Stasko, J.T.: Visualization for fault localization. In: Proceedings of ICSE Workshop on Software Visualization, pp. 71–75 (2001)
16. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 15–26. ACM, New York (2005)
17. Naish, L., Lee, H.J., Ramamohanarao, K.: A model for spectra-based software diagnosis. ACM Transactions on Software Engineering Methodology 20(3), 11:1–11:32 (2011)
18. Ochiai, A.: Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions. Bulletin of the Japanese Society of Scientific Fisheries 22(9), 526–530 (1957)
19. Park, S., Vuduc, R.W., Harrold, M.J.: Falcon: fault localization in concurrent programs. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010, vol. 1, pp. 245–254. ACM, New York (2010)
20. Perone, C.S.: PyEvolve, `http://pyevolve.sourceforge.net`
21. Renieres, M., Reiss, S.: Fault localization with nearest neighbor queries. In: Proceedings of the 18th International Conference on Automated Software Engineering, pp. 30–39 (October 2003)
22. Vargha, A., Delaney, H.D.: A critique and improvement of the "CL" common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics 25(2), 101–132 (2000)
23. Wong, E., Debroy, V.: A survey of software fault localization. Tech. Rep. UTDCS-45-09, Department of Computer Science, University of Texas at Dallas (November 2009)
24. Wong, W.E., Qi, Y., Zhao, L., Cai, K.Y.: Effective fault localization using code coverage. In: Proceedings of the 31st Annual International Computer Software and Applications Conference, COMPSAC 2007, vol. 01, pp. 449–456. IEEE Computer Society, Washington, DC (2007)
25. Yoo, S., Harman, M., Clark, D.: FLINT: Fault localisation using information theory. Tech. Rep. RN/11/09, Department of Computer Science, University College London (March 2011)
26. Yu, Y., Jones, J.A., Harrold, M.J.: An empirical study of the effects of test-suite reduction on fault localization. In: Proceedings of the International Conference on Software Engineering (ICSE 2008), pp. 201–210. ACM Press (May 2008)
27. Zeller, A.: Automated debugging: Are we close? IEEE Computer 34(11), 26–31 (2001)
28. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann Publishers Inc., San Francisco (2005)