基于网络聚合频谱的软件故障定位方法

何洪豆[1),2)] 任家东[1),2)] 赵谷雨[1),2)] 何海涛[1),2)] 黄国言[1),2)]

[1)](燕山大学信息科学与工程学院 河北秦皇岛 066001)

[2)](河北省软件工程重点实验室(燕山大学) 河北秦皇岛 066001)

摘 要 基于频谱的软件故障定位是一种最广泛研究的启发式方法，具有轻量级且高效的特点。然而，现有研究仅考虑实体在测试用例中的覆盖情况，而忽略了实体间依赖关系的分析。本文以软件实体间依赖关系能够反映故障关联关系和故障可疑性分布为出发点，通过聚合关联实体的影响改进测试频谱计算方法，提出了一种基于网络聚合频谱的软件故障定位方法。(1) 基于复杂网络和测试覆盖构建一种软件频谱依赖网络，描述实体间依赖关系和故障关联强度；(2) 提出网络聚合频谱的概念和计算方法，聚合关联实体的影响综合评估实体的故障可疑性；(3) 基于网络聚合频谱，使用故障定位函数计算故障可疑评分，生成实体故障排序列表。实验使用 33 个现有故障定位函数对提出的网络聚合频谱进行了检验，对除 $ER5^c$ 之外的故障定位函数均具有一定的性能提升；在 33 个故障定位函数上，acc@1、acc@3 和 acc@5 指标的平均性能分别提高了 7.6%、11.9%和 13.3%；对于最优故障定位函数，acc@1、acc@3 和 acc@5 指标性能分别提高了 10%、14.9%和 16.6%。

关键词 故障定位；软件缺陷；软件网络；软件频谱

中图法分类号 TP311 DOI 号

Network Aggregated Spectrum Based Software Fault Localization Approach

HE Hong-Dou[1),2)] REN Jia-Dong[1),2)] ZHAO Gu-Yu[1),2)] HE Hai-Tao[1),2)]

[1)](School of Information Science and Engineering, Yanshan University, Qinhuangdao, Hebei, 066001)

[2)](The Key Software Engineering Laboratory of Hebei Province(Yanshan University), Qinhuangdao, Hebei, 066001)

Abstract Spectrum-Based Fault Localization (SBFL) has been one of the most widely studied heuristic approach considering its light-weight but effective characteristics. While, existing studies mainly inspect the test coverage but ignore the analysis of correlations between software entities. In the paper, based on the hypothesis and intuition that correlations and dependencies reflect the fault association and suspiciousness distribution, an optimized Network Aggregated Spectrum (NAS) based fault localization approach is proposed, which aggregates the influences of associated entities to improve the calculation of test spectrum. (1) Based on the complex network and test coverage, the Software Spectrum Dependence Network (SSDN) is constructed which characterizes the correlation and the Fault Relevance (FR) between entities. (2) The Network Aggregated Spectrum is defined to synthetically estimate the suspiciousness considering the influences of associated entities. (3) With the NAS as input, a SBFL fault function is applied to calculate the final suspicious score and generate the ranking list. Experiments are conducted using 33 existing fault functions to test the effectiveness of proposed NAS. Result shows that almost all the performance of these functions can be improved expect for $ER5^c$. The average performances of acc@1,

收稿日期：年-月-日；最终修改稿收到日期：年-月-日. 本课题得到国家自然科学基金(No.61772449, No.61572420, No.61807028, No.61802332)、河北省自然科学基金(No.F2019203120)资助. **何洪豆**，男，1991年生，博士研究生，主要研究领域为软件故障定位、软件动态行为分析. **任家东**（**通信作者**），男，1967年生，博士研究生，教授，计算机学会(CCF)会员，主要研究领域为数据挖掘、软件安全. E-mail: jdren@ysu.edu.cn. **赵谷雨**，女，1993年生，博士研究生，主要研究领域为数据挖掘、复杂网络. **何海涛**，女，1968年生，博士研究生，教授，主要研究领域为数据挖掘、软件安全. **黄国言**，男，1969年生，博士研究生，教授，主要研究领域为网络协同技术、软件安全。

第1作者手机号码: 18713509835, E-mail: hehondou@yeah.net

acc@3 and acc@5 are improved by 7.6%, 11.9% and 13.3% separately. For the best performance fault locator, the performances of acc@1, acc@3 and acc@5 are improved by 10%, 14.9% and 16.6%.

Key words fault localization; software bug; software network; software spectrum

1 引言

现代软件系统已经成为目前最不可或缺且最复杂的人工系统之一[1]，其复杂性随着自身的演化和发展呈现着不断增长的规律[2]。随之而来的是，修复软件系统中不可避免的故障(bugs)的支出，占据了软件开发和维护费用的 50%~80%[3, 4]。在软件的生命周期中，软件测试、调试和验证工作是主要开销，这是因为精确定位导致故障的软件实体是其中最耗时和费力的工作[5]。因此，在软件工程领域，需要迫切发展自动软件故障定位技术，以协助开发人员识别故障的位置。

在过去的二三十年中，出现了各种协助定位故障根本原因的自动故障定位技术。(1) 基于频谱的故障定位(Spectrum-Based Fault Localization, SBFL)[5, 6-12]，应用测试覆盖信息来度量实体的故障可疑性。(2) 基于变异的故障定位(Mutation-based fault localization)[13, 14-17]，利用程序变异产生的运行差异评估实体故障可能性。(3) 基于切片的故障定位(Slicing-based fault localization)[18-21]，通过分析实体间的动态依赖关系来定位故障原因。(4) 基于信息检索的技术(Information-retrieval-based techniques)[22, 23]，采用自然语言处理方法来分析故障报告和软件实体，基于某种相似性度量对实体可疑性排序。(5) 基于机器学习的方法(Machine-learning-based approaches)[24, 25]，通过引入故障特征模式挖掘来解决故障定位问题。这些技术大多数旨在生成一个软件实体的可疑性排序列表，而开发人员则根据实体的可疑程度大小进行逐一检查，最终发现真实故障的实体。理想情况下，故障定位方法应该总能将真实故障实体排序在列表的最前面。

在现有的自动故障定位技术中，基于频谱的故障定位(SBFL)由于其相对较低的测试执行开销[26]和相对较好的性能表现[13]得到了广泛关注。SBFL 根据失败和成功测试用例执行时对软件实体的覆盖情况计算实体的可疑性评分并排序，其启发式在于：被更多失败测试用例覆盖且被更少成功测试用例覆盖的软件实体更可能存在故障。虽然 SBFL 方法论在软件故障定位中被广泛研究[5, 8, 11, 12, 27, 28]，应用在真实软件故障定位中时，在理论和实践上都存在一定的局限性。在实践上，Parnin 等人[29]和 Person 等人[13]分别对目前先进的 SBFL 方法进行了真实项目定位性能的评估，结果表明之前声称的有效性在实践中并不奏效。在理论上，Yoo 等人[30]发现不存在一种 SBFL 故障定位函数(公式)能够在不同的项目上都优于其他故障定位函数。因此，需要投入更多的努力来提升 SBFL 方法的性能。

针对现有 SBFL 仅分析软件测试覆盖信息而忽略软件实体之间交互依赖关系的问题，本文提出了一种聚合关联实体影响的网络聚合频谱(Network Aggregated Spectrum, NAS)计算方法，用于优化 SBFL 中测试频谱对实体故障可疑程度描述的准确性和显著性。(1) 网络聚合频谱充分考虑了关联(依赖和被依赖)实体的故障关联性，捕捉潜在故障可疑性，综合计算实体可疑评分。(2) 通过建立实体依赖网络表征模型，分析软件动态行为模式，量化了实体间故障关联程度。

本文主要贡献包括：

(1) 构建了一种软件频谱依赖网络(Software Spectrum Dependence Network, SSDN)表征模型，能够准确描述软件函数实体之间的故障依赖关系，量化实体间的故障关联强度(Fault Relevance, FR)。

(2) 提出了一种网络聚合频谱(Network Aggregated Spectrum, NAS)的 SBFL 测试频谱计算方法，聚合关联实体的影响综合评估可疑性，提升软件故障定位的性能。

(3) 实验在真实基准数据集上使用 33 个当前故障定位函数进行了网络聚合频谱的性能检验，证明了所提出方法的有效性，并通过大量实验确定了网络聚合频谱计算的最优方案。

2 研究动机

如图 1 所示，基于频谱的故障定位(SBFL)主要包含三个步骤：(1) 追踪测试用例动态执行，构建测试覆盖矩阵；(2) 根据测试覆盖矩阵计算测试频谱；(3) 应用故障定位函数(公式)计算实体故障可疑评分。其中，测试覆盖矩阵反映软件故障的基本事实，而推导测试频谱和设计故障定位函数是 SBFL 的重点，也是研究的焦点。

软件测试频谱是一个四元组向量(e_f, e_p, n_f, n_p)，

四个指示因子分别表示执行实体的失败测试用例个数、执行软件实体的成功测试用例个数、未执行软件实体的失败测试用例个数和未执行软件实体的成功测试用例个数。这四个指示因子分别从不同角度描述一个软件实体存在或不存在故障的可能性，而每个因子在其各自角度对软件实体进行描述或度量的准确性和显著性就显得十分重要。

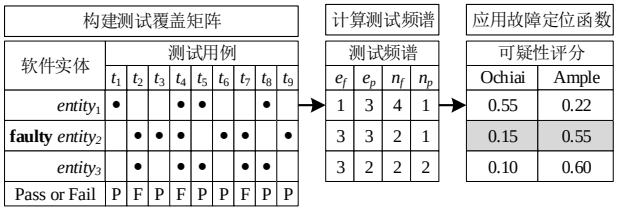


图 1　基于频谱的软件故障定位(SBFL)过程

在 SBFL 方法中，软件测试频谱的计算是整体方法的第二步，也是承上启下的关键性步骤。首先，测试频谱构建于测试覆盖矩阵之上，是对实体在成功和失败测试用例中执行情况的统计表示，反映了存在故障的可能性。其次，测试频谱作为故障定位函数的输入，用于计算软件实体的可疑性评分，其计算方法和结果对故障定位性能具有重要影响。最后，测试频谱作为 SBFL 方法的中间环节，是一种基于原始测试覆盖数据上的特征提取。因此，测试频谱中四个指示因子的度量值，一方面代表了“特征提取”所提取特征的合理性，另一方面影响着故障定位函数的有效性。

传统 SBFL 中，测试频谱计算仅考虑实体在测试用例中执行情况，而忽略了实体的内在调用依赖关系。在软件动态执行时，软件实体往往被多个实体调用或调用多个实体来共同完成某项功能，因此相互依赖的实体之间在故障分析上存在着相互影响。由于测试用例无法完全覆盖所有的执行情况，因此可以通过捕捉关联实体的影响来综合评估实体故障可疑性，由此得出启发如下。

(1) 故障关联：实体间的调用依赖决定了它们在测试执行中的关联性，即同时在失败或成功测试用例中执行。关联实体的故障可疑程度对目标实体的可疑性评估具有补充作用，能够增强目标实体各频谱指示因子的准确性。

(2) 故障分布：由于故障关联的存在，目标实体周围分布的关联实体也具有相似的故障特征。根据关联实体故障可疑性分布，进行影响传播聚合分析，可以增强频谱指示因子评估的显著性。

综合上述分析，本文开展了聚合关联实体影响优化测试频谱计算的研究，提出了一种基于软件动态调用分析的网络聚合频谱计算方法，通过建立针对软件测试过程的动态软件函数调用网络，聚合关联(调用和被调用)函数实体的影响，增强四个频谱指示因子对函数实体故障可疑性度量的准确性和显著性。

3 本文方法

3.1 整体框架

本文所提出基于网络聚合频谱的软件故障定位方法框架如图 2 所示。通过追踪软件测试用例的动态执行过程，收集动态执行和调用日志，建立动态执行轨迹数据库；基于动态执行轨迹数据，进行数据提取得到测试覆盖矩阵，并进一步提取出测试频谱数据；以动态调用关系建立实体调用依赖网络，并结合根据测试覆盖矩阵计算实体间故障关联强度，构建软件频谱依赖网络；基于软件频谱依赖网络和测试频谱，聚合关联实体的影响计算网络聚合频谱向量，最后采用故障定位函数即可计算出实体故障评分。

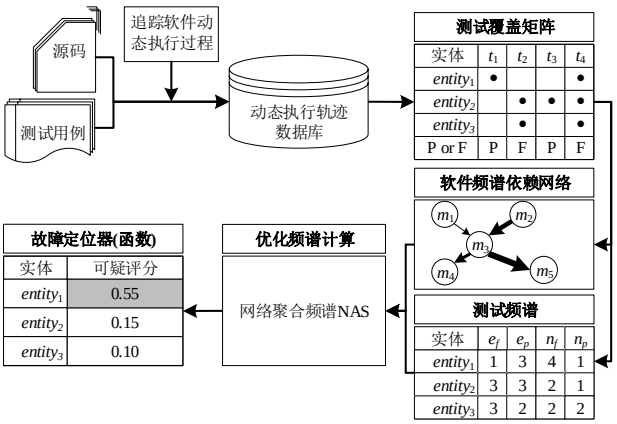


图 2　基于网络聚合频谱故障定位框架图

本本选取函数粒度实体进行故障定位研究，原因包括：(1) 在面向对象的软件开发中，函数是进行单元测试的最小单元[31]；(2) 对象之间的交互也主要通过遵循某种协议的函数调用来实现[32]；(3) 从软件测试角度来说，函数通常包含足够的上下文信息来帮助理解故障的原因[23]；(4) 开发人员更偏向于在函数级别的实体上进行故障调试[33]。同时，为了简化描述，分别将发出调用函数和被调用函数称为 caller 函数和 callee 函数。

3.2 动态执行追踪和数据提取

由于传统 SBFL 方法追踪软件动态执行轨迹时

不包含函数调用信息，而本文方法依赖于函数调用关系，故重新设计和开发了动态执行轨迹追踪工具。

3.2.1 追踪软件测试动态执行

为了收集包含调用关系的软件测试执行轨迹，开发了基于 Java ASM 技术的追踪器。追踪器分别在被调用方法被调用时和被调用方法执行完返回时触发，以准确地识别调用关系。该追踪程序不修改源代码，而是使用 Java 代理对加载到 JVM 中的字节码文件进行动态插桩。测试用例执行完成后，会生成一个记录函数调用关系的日志文件。为了形式化表述该过程，给出如下相关定义。

定义1. 执行轨迹(Trace, T)。执行轨迹T_t表示测试用例t执行产生的调用关系集，其形式化定义如公式(1)。

$$T_t=\{<caller_i, callee_j>, i, j\in\mathbb{N}\} \tag{1}$$

其中，$caller_i$是某个 caller 函数的唯一标识，$callee_j$是对应 callee 函数的唯一标识。函数的唯一标识用全称表示，如 org.jfree.data.Data.getX(int)。特别地，具有相同类名和函数名，但参数不同的函数，被认为是不同的函数。

给定某项目的一个版本，每个测试用例 t 的执行都会生成一个单独的执行轨迹 T_t。这样，可以方便地提取每个测试用例单独的调用关系和测试覆盖数据。此外，为了节省存储空间，每个函数的唯一标识在存储时都被替换成一个唯一的整数数字。

3.2.2 数据提取

在该过程中，首先完成从执行轨迹中提出函数调用关系和函数测试覆盖数据的工作，然后计算测试覆盖矩阵，并进一步计算软件测试频谱，相关定义如下。

定义 2. 测试调用图(Test Call Graph, TCG)。测试调用图TCG_t表示从测试用例 t 产生的执行轨迹T_t抽象出来的有向图，其中节点表示函数，边表示函数调用关系，边的方向为从 caller 函数指向 callee 函数，其形式化表达如公式(2)。

$$TCG_t=(V, E) \tag{2}$$

其中，$V=\{v_1, v_2,\ldots, v_n \mid n\in\mathbb{N}\}$为函数节点集，$E=\{v_i\rightarrow v_j \mid i, j\in|V|\ \}$表示有向边集，$v_i$为 caller 函数，$v_j$为 callee 函数。

定义 3. 测试覆盖矩阵(Test Coverage Matrix, TCM)。测试覆盖矩阵TCM是一个记录函数在一组测试用例下被执行情况的二维矩阵，从一组测试用例产生的罪行轨迹中提取，其形式化表达如公式(3)。

$$TCM=\begin{bmatrix} e_1^1 & \cdots & e_1^t \\ \vdots & \ddots & \vdots \\ e_m^1 & \cdots & e_m^t \end{bmatrix} \tag{3}$$

TCM 矩阵中的每一行对应一个函数，每一列对应一个测试用例；e_m^t表示函数 m 是否在测试用例 t 中被执行，$e_m^t=1$表示函数 m 被执行，$e_m^t=0$表示未被执行。

定义 4. 测试频谱[34](Test Spectrum, TS)。测试频谱TS_m是一个表示给定函数 m 在一组测试用例下被执行情况的统计，其形式化定义如公式(4)。

$$TS_m=(e_f, e_p, n_f, n_p) \tag{4}$$

TS_m包括四个因子e_f、e_p、n_f和n_p，分别表示执行函数 m 的失败测试用例数量、执行函数 m 的成功测试用例数量、未执行函数 m 的失败测试用例数量和未执行函数 m 的成功测试用例数量。

总的来说，所抽取的数据包括 TCG、TCM 和 TS 三种统计结果。不同于 TCG，TCM 是一组测试用例执行情况的统计结果。通过对测试覆盖矩阵 TCM 和每个测试用例的执行成功、失败状态，来进一步计算得到每个函数的测试频谱TS_m。该过程中所定义的测试覆盖矩阵 TCM 和测试频谱 TS 与传统 SBFL 方法中的定义和表达相同。由于收集测试用例动态执行轨迹时存在差异，所以在该部分给出了相关的定义和解释，也为了给出更加合理和完整的方法描述。

3.3 软件频谱依赖网络构建

本文所提出的网络聚合频谱是一种基于软件动态函数调用网络分析的软件测试频谱计算方法。为了进行函数间调用行为关系的描述和行为强度的度量，构建了一个动态的软件频谱依赖网络(Software Spectrum Dependence Network, SSDN)。在构建软件频谱依赖网络时，需要对测试用例执行所产生的函数覆盖数据进行预处理，提取出函数调用关系，以及执行某一函数调用的成功和失败测试用例个数。

由于每个测试用例的执行都会产生一个简单的函数调用关系网络，可以通过对所有产生的调用关系网络进行叠加，即可构建出 SSDN 的网络拓扑结构。在所提出方法中，为了对 caller 和 callee 在聚合频谱中起到的作用进行分析，构建 SSDN 时增加了边的方向进行调用关系的区分。同时，为了度量两个函数节点在测试频谱上的关联程度，提出了故障关联程度 FR 对调用边加权。构建的软件频谱依赖网络 SSDN 如定义 5 所示。

定义 5. 软件频谱依赖网络(Software Spectrum Dependence Network, SSDN)。软件频谱依赖网络 SSDN 是一个描述软件函数实体测试频谱依赖关系的网络，以函数为网络节点，函数调用关系为有向边，边权重表示函数节点的频谱关联强度，其形式化定义如公式(5)所示。

$$SSCN=(V, E, W) \tag{5}$$

其中，V 表示函数节点集合$\{v_1, v_2, \ldots, v_i, \ldots, v_n\}$，E 表示边集$\{v_i \rightarrow v_j \mid i, j \in |V|\}$，$v_i \rightarrow v_j$ 表示函数节点 v_i 调用 v_j；W 为边权重集合，表示节点间的故障关联强度，被定义为 $FR_{i,j}$。

在软件频谱依赖网络 SSDN 中，函数节点之间存在调用和被调用的依赖关系。在一组测试用例 T 下，通过对函数 m 的被执行情况，可以评估出反应函数 m 是导致功能失效可能性的频谱(e_f, e_p, n_f, n_p)；但是该测试用例 T 不能完全体现函数 m 在整个功能执行下的所有情况，这也就导致了对函数 m 的评估是不充分的。在对这种情况进行弥补时，考虑到与函数 m 存在依赖关系的函数 n，由于它们之间存在调用依赖关系，所以函数 n 的测试频谱在一定程度上能够反应函数 m 在测试组件 T 之外的潜在可疑性。在对这种影响进行评估时，采取将函数 n 的可疑指示因子叠加到函数 m 上的操作。但这种影响又不是绝对的和完全的，因此引入反映函数 m 和 n 之间相关性的度量——故障关联强度 FR。

定义 6. 故障关联强度(Fault Relevance, FR)。故障关联强度 $FR_{i,j}$ 表示 callee 函数 v_j 与 caller 函数 v_i 在测试频谱度量上的关联程度，由分别覆盖两个函数的测试用例集的交叉占比计算得到，定义如公式(6)所示。

$$FR_{i,j} = \frac{|FT_i \cap FT_j|}{|FT_i \cap FT_j| + |PT_i \cap PT_j|} \tag{6}$$

其中，FT_i和FT_j分别为执行函数 v_i 和函数 v_j 的失败测试用例(Failing Test)集，PT_i和PT_j分别为执行函数 v_i 和函数 v_j 的成功测试用例(Passing Test)集，$|\cdot|$表示集合大小。

故障关联强度 FR 反应的是两个函数由于依赖关系所产生的潜在可疑性关联程度，，使用它们各自相关测试用例的交集来计算关联强度。由于故障关联强度是用于评估函数的故障可疑性，而不是全部情况，故强调失败测试用例的交集。

传统软件测试频谱包含四个指示因子e_f、e_p、n_f和n_p，分别在不同角度体现故障可疑性。为了保持度量的全面性，在给出的网络聚合频谱中 NAS 中，同样包含这四种指示因子，但每个指示因子都更新为纳入依赖函数影响的聚合表达。基于软件频谱依赖网络 SSDN 和故障关联强度 FR，分别累加 caller 函数和 callee 函数的加权指示因子，叠加到目标函数原指示因子上，从而得到网络聚合后的频谱向量。

故障关联强度 FR 在 caller 函数和 callee 函数之间进行计算，分析它们之间失败和测试用例集合的交叉情况。计算过程如算法 1 所示，使用 *ftDict* 字典存储执行每个函数的失败测试用例数组，使用 *ptDict* 字典存储执行每个函数的成功测试用例数组。在计算关联强度时，通过 *inter*()函数分别计算两个函数的失败和成功测试用例的交集，再计算共同失败测试用例数量占全部共同测试用例数量的比值作为故障关联强度。

算法 1. 故障关联强度评估算法.

输入：*calleeDict*, *ftDict*, *ptDict*

输出：*frDict*

```
01:  Intialize frDict;
02:  FOR each caller in calleeDict.keys()
03:    IF calleeDict.get(caller).size() > 0
04:      FOR each callee in calleeDict.get(caller)
05:        callKey = caller + '@' + callee;
06:        inFt = inter(ftDict.get(caller), ftDict.get(callee));
07:        inPt = inter(ptDict.get(caller), ptDict.get(callee));
08:        FR = inFt.size() / (inFt.size() + inPt.size());
09:        frDict.set(callKey, FR);
10:      END FOR
11:    END IF
12:  END FOR
13:  RETURN frDict;
```

在算法 1 中，第 3 行判断 caller 是否具有 callee，如果没有则不进行处理。第 4 行至第 9 行遍历 callee 数组，计算故障关联强度，并存储到 *frDict* 字典中。其中，第 6 行和第 7 行分别计算失败和成功测试用例的交集，第 8 行计算故障关联强度 FR。最后在 13 行返回 *frDict*。

3.4 网络聚合频谱计算

定义 7. 网络聚合频谱(Network Aggregated Spectrum, NAS) 网络聚合频谱 NAS_i 为一个四元组 $(na(e_{f,i}), na(e_{p,i}), na(n_{f,i}), na(n_{p,i}))$，表示基于软件频谱依赖网络 SSDN，在故障关联强度 FR 下聚合关联函数实体频谱计算后的测试频谱向量，定义如公式(7)所示。

$$NAS_i=(na(e_{f,i}), na(e_{p,i}), na(n_{f,i}), na(n_{p,i})) \tag{7}$$

在公式(7)中，$na(e_{f,i})$、$na(e_{p,i})$、$na(n_{f,i})$)和 $na(n_{p,i})$分别表示通过网络聚合后执行函数实体 v_i的失败测试用例数量、执行 v_i的成功测试用例数量、未执行 v_i的失败测试用例数量和未执行 v_i的成功测试用例数量；$na(\cdot)$为聚合函数，计算公式如式(8)。

$$na(f_i)=f_i+\sum_{j\in Caller_i} f_j*FR_{j,i}+\sum_{k\in Callee_i} f_k*FR_{i,k} \quad (8)$$

在公式(8)中，f_i 表示传统软件测试频谱中的某一具体指示因子，如 e_f；j表示函数 v_i 的某个 caller 函数 v_j，$Caller_i$表示函数 v_i的 caller 函数集合；k表示函数 v_i 的某个 callee 函数 v_k，$Callee_i$表示函数 v_i 的 callee 函数集合。

在改进的网络聚合频谱之上，下一步需要使用某个具体的故障定位函数计算每个函数实体的最终故障可疑性评分，如公式(9)。由于测试频谱位于 SBFL 的中间步骤，因此改进的网络聚合频谱适合于所有现有的故障定位函数。

$$Susp_F(v_i)=F(NAS_i) \quad (9)$$

在公式(9)中，F 表示某个具体的故障定位函数(如 Ochiai)，F 以网络聚合频谱 NAS_i 为输入进行计算；$SuspF(v_i)$表示函数 v_i 的最终故障可疑性评分。

最后，根据计算的故障可疑性评分 SuspF，对函数进行降序排序，获得排序列表。开发人员则根据排序列表逐一检查函数是否真正存在故障。

在使用网络聚合频谱进行软件故障定位时，首先需要根据公式(7)和公式(8)计算出各个函数的网络聚合频谱 NAS，然后选定要使用的故障定位函数 F，使用公式(9)计算出每个函数的故障可疑性评分。在计算网络聚合频谱时，需要分别遍历目标函数的 caller 集合和 callee 集合，分别聚合不同类型关联函数实体的影响。具体过程如算法 2 所示，其中 *methodSet* 为函数集，*callerDict* 字典存储每个函数的 caller 函数数组，*calleeDict* 字典存储每个函数的 callee 函数数组，*rawSpDict* 字典存储每个函数的原始频谱向量，*frDict* 字典存储函数调用对的故障关联强度，F 为特定的故障定位函数，*suspDict* 字典存储每个函数的最终可疑评分。

算法 2. 网络聚合频谱故障定位算法.

输入：*methodSet*, *callerDict*, *calleeDict*, *rawSpDict*, *frDict*, F

输出：*suspDict*

```
01:  Intialize naSpDict, suspDict;
02:  FOR each method m in methodSet
03:     rawSp = rawSpDict.get(m);
04:     IF calleeDict.get(m).size() > 0
05:        FOR each callee in calleeDict.get(m)
06:           callKey = m + '@' + callee;
07:           FR = frDict.get(callKey);
08:           rawSp += FR * rawSpDict.get(callee);
09:        END FOR
10:     END IF
11:     IF callerDict.get(m).size() > 0
12:        FOR each caller in callerDict.get(m)
13:           callKey = caller + '@' + m;
14:           FR = frDict.get(callKey);
15:           rawSp += FR * rawSpDict.get(caller);
16:        END FOR
17:     END IF
18:     naSpDict.set(m, rawSp);
19:  END FOR
20:  FOR each method m in naSpDict.keys()
21:     suspDict.set(m, F(naSpDict.get(m)));
22:  END FOR
23:  RETURN suspDict;
```

在算法 2 中，第 2 行至第 20 行用于计算每个函数的网络聚合频谱；其中第 4 行至第 11 行进行 callee 函数的关联影响聚合，第 12 行至第 18 行进行 caller 函数的关联影响聚合，第 19 行将计算的网络聚合频谱保存到 *naSpDict* 中。在 21 行至第 23 行，通过遍历每个函数的网络聚合频谱，使用指定故障定位函数 F 计算最终的故障可疑性评分，并存入 *suspDict* 中。最后在 24 行返回函数评分集 *suspDict*。

4 实验结果及分析

4.1 实验对象

实验所选取的数据集为 Defects4J[35]，是一个成熟的用于软件测试实验的真实 bug 数据集，在软件测试研究中被广泛应用[23, 27, 28, 32, 36]。

表 1 实验对象统计信息

项目	故障数量	源码行数(千行)	测试代码行数(千行)	测试用例个数
Chart	25	96	50	2, 205
Closure	132	90	83	7, 927
Lang	62	22	6	2, 245
Math	104	85	19	3, 602
Time	26	28	54	4, 130

实验中选取了 349 个真实故障，来源于以下 5 个开源 Java 项目：JFree**Chart**、Google **Closure** Compiler、Apache Common **Lang**、Apache Common

Math 和 Joda-**Time**。表 1 是所使用的项目的统计描述信息。对于每个故障，Defects4J 提供了存在 bug 的版本源码、具有最少修改的 bug 修复版本源码、失败测试用例列表和 bug 相关测试用例列表。

4.2 评价指标

软件故障定位方法输出一个可疑实体列表，尽可能地将真实故障实体排在列表的第一位。在实验对比中，使用了 wasted effort、acc@n 和 mean effort 三种最常用的度量[25, 27, 28, 32]来评估所提出方法的性能。

Mean Wasted Effort (MWE)–越小越好。 Mean Wasted Effort 度量是所有故障的排序列表 Wasted Effort 度量平均值。Wasted Effort 度量是一种绝对度量，考察定位到第一个真实故障实体之前必须要排查的实体数量，计算公式如下。

$$\text{Wasted Effort} = m + \frac{n}{2} \tag{10}$$

其中，m是排序严格高于故障实体的非故障实体个数，n表示与故障实体排序相同的非故障实体个数，n与排序列表中的 tie-break 问题有关。

acc@n–越大越好。acc@n 定义为真实故障实体排在列表前 n 位置内的故障数量。受文献[27]的启发，n 的值选取为 1、3 和 5。由于两个实体排在相同位置的情况比较常见，在进行 tie-break 时使用文献[28]中的随机 tie-breaker。

Mean Average Precision (MAP)–越大越好。 MAP 是一种在信息检索领域用于评价排序质量的度量，同样适用于故障定位研究。MAP 考察所有的故障实体，并强调准确率之上的召回率。因此，MAP 更适合开发人员深度搜索排序列表以找出更多故障相关实体的情形。MAP 是 Average Precision 的平均值，Average Precision 的计算方法如公式(11)和(12)所示。

$$\text{Average Precision} = \sum_{i=1}^{M} \frac{P(i) \times pos(i)}{\text{number of faulty methods}} \tag{11}$$

$$P(i) = \frac{\text{number of faulty methods in top } i}{i} \tag{12}$$

式中，i为一个函数实体在排序列表中的位置，M为排序列表的长度，$pos(i)$是一个指示在排序列表第i个位置上的函数实体是否为故障实体的布尔值(0 或 1)，$P(i)$是排序列表第i个位置上的准确度。

4.3 实验设计

为了评价所提出网络聚合频谱的有效性，以及分析影响网络聚合频谱的相关因素，在改进的网络聚合频谱上分别应用了 33 个现有故障定位函数[23, 25]进行实验。在实验结果的评价时，分别从以下几个方面进行合理性和有效性分析：(1) 最佳性能结果分析，证明所提出网络聚合频谱对于故障定位性能提升上的效果；(2) 加权聚合性能分析，验证在计算网络聚合频谱时所使用的故障关联强度 FR 的有效性；(3) 聚合实体类型性能分析，分析对 caller 函数影响的聚合和对 callee 函数影响的聚合对于故障定位性能的提升分别有什么效果；(4) 不同知识因子性能分析，研究软件测试频谱中的四个指示因子的聚合是否都能够有助于提升方法性能。

4.4 实验结果及分析

4.4.1 最佳性能结果分析

在对所提出方法进行检验时，需要讨论三个因素对于故障定位性能提升的影响，包括聚合关联实体影响时是否加权、聚合实体类型(caller 和 callee)影响以及聚合不同组合指示因子(e_f、e_p、n_f和 n_p)的效果。对于这些因素的讨论分别在第 4.4.2 节、第 4.4.3 节和第 4.4.4 节进行，本小节对所提出的网络聚合频谱的最佳性能结果进行分析。在网络聚合频谱取得最佳性能时，以上三个因素分别确定为：加权聚合、仅聚合 callee 类型实体影响和聚合指示因子组为 e_f+n_p。

表 2 是在使用原始频谱时，33 个 SBFL 故障定位函数的指标性能统计；表 3 是使用本文所提出的网络聚合频谱时，同样 33 个 SBFL 故障定位函数的指标性能统计。在两个表格中，@1、@3 和@5 分别表示 acc@1、acc@3 和 acc@5 评价指标，R 表示不同故障定位函数的性能总排序。在对故障定位函数的性能进行排序时，采取依次对比 acc@1、acc@3、acc@5、MAP 和 MWE 度量指标的方式进行，如果前面指标性能的对比能够区分开来，则不对后面的指标再进行对比。采用这种对比方式的原因在于，在软件故障定位中，越能够将真实故障实体排在前面就越快排除故障。

通过对比表3-3和表3-2可知，在使用网络聚合频谱之后，所有33个故障定位函数的性能都得到了一定幅度的提升。平均来说，网络聚合频谱将33个故障定位函数的平均acc@1性能从47提升到73.5，相对于349个软件故障，定位性能提升了7.6%，增长率为56.4%。同样对于acc@3和acc@5指标来说，acc@3性能从87提升到128.4，提升了11.9%；acc@5性能从104提升到了150.5，性能提升了13.3%。对于指标MAP和MWE，它们的性能都有不同程度的提

表 2 原始频谱下的故障定位性能

故障定位器(函数)	@1	@3	@5	MAP	MWE	R
Tarantula	66	118	136	0.318	73.25	1
ER1[b]	66	113	138	0.313	73.99	2
GP13	66	113	138	0.313	74.03	3
Ochiai2	65	116	136	0.312	88.17	4
Ochiai	65	115	135	0.317	70.65	5
Zoltar	65	114	137	0.320	72.88	6
Kulczynski2	65	113	134	0.318	73.11	7
M2	65	111	137	0.309	73.88	8
ER1[a]	65	111	133	0.304	87.85	9
Jaccard	64	115	135	0.314	73.97	10
Dice	64	115	135	0.314	73.97	10
Goodman	64	115	135	0.314	73.97	10
Sϕrensen Dice	64	115	135	0.314	73.97	10
Anderberg	64	115	135	0.314	73.97	10
AMPLE	59	99	116	0.276	137.0	11
Kulczynski1	53	102	122	0.270	85.31	12
GP2	52	98	119	0.275	153.8	13
Wong3	46	77	94	0.227	103.2	14
Hamming	42	71	80	0.209	263.7	15
Euclid	42	71	80	0.209	263.7	15
Wong2	42	71	80	0.209	263.7	15
Simple Matching	42	71	80	0.209	263.7	15
Rogers Tanimoto	42	71	80	0.209	263.7	15
Sokal	42	71	80	0.209	263.7	15
GP19	32	70	92	0.189	207.5	16
M1	30	49	57	0.147	280.0	17
Hamann	28	50	55	0.146	289.7	18
ER5[a]	20	60	75	0.156	159.8	19
Wong1	20	60	75	0.156	159.8	19
ER5[b]	20	60	75	0.156	159.8	19
RusselRao	20	60	75	0.156	159.8	19
ER5[c]	19	58	72	0.150	173.4	20
GP3	8	24	31	0.072	299.3	21
Average	**47**	**87**	**104**	**0.243**	**151.8**	

表 3 网络聚合频谱下的故障定位性能

故障定位器(函数)	@1	@3	@5	MAP	MWE	R
GP2	101	170	194	0.421	80.41	1
ER1[b]	99	169	199	0.423	23.29	2
M2	94	150	180	0.396	29.18	3
GP13	91	165	193	0.404	19.56	4
ER5[a]	88	157	185	0.377	22.81	5
Wong1	88	157	185	0.377	22.81	5
ER1[a]	87	155	180	0.394	61.61	6
GP19	86	151	189	0.378	75.18	7
Wong3	80	136	164	0.354	36.22	8
Euclid	79	136	154	0.349	142.5	9
Hamming	79	136	154	0.349	142.9	10
Kulczynski1	77	138	162	0.347	45.86	11
Zoltar	77	138	161	0.360	35.84	12
Kulczynski2	76	139	162	0.361	30.69	13
Goodman	76	137	161	0.359	34.50	14
Anderberg	76	137	161	0.359	34.51	15
Sϕrensen Dice	76	137	161	0.359	34.51	16
Dice	76	137	161	0.359	34.51	17
Jaccard	76	137	161	0.359	34.51	18
Ochiai	76	137	160	0.360	32.22	18
Ochiai2	74	131	155	0.351	48.63	19
Tarantula	72	129	149	0.344	51.43	20
ER5[b]	72	126	154	0.308	40.08	21
RusselRao	72	126	154	0.308	40.08	22
AMPLE	69	116	137	0.326	88.19	22
Wong2	66	111	125	0.309	220.1	23
Simple Matching	60	92	105	0.269	212.2	24
Rogers Tanimoto	60	92	105	0.269	212.3	25
Sokal	60	92	105	0.269	212.3	26
M1	55	83	93	0.230	228.5	26
GP3	52	98	112	0.217	217.2	27
Hamann	40	67	77	0.189	236.5	28
ER5[c]	19	58	72	0.150	173.4	29
Average	**73**	**128**	**150**	**0.333**	**89.53**	

升，其中MAP从0.244增长到0.333；而对于MWE，从151.78降到89.53，表示根据排序需要检查的函数平均减少了62.25个。根据以上分析可知，网络聚合频谱对软件故障定位问题的改善具有相当的作用，能够提升几乎所有33个故障定位函数的性能。

通过表 3 可知，在使用 NAS 聚合频谱后，性能最佳的故障定位函数为 GP2，其 acc@1、acc@3 和 acc@5 性能分别为 101、170 和 194。相对于原始频谱最佳定位函数 Tarantula，在这三个指标上性能分别提升了 10%(66 至 101)、14.9%(118 至 170)

和 16.6%(136 至 194)。而相对于原始频谱下的 GP2 本身，三个指标的性能提升分别为 14%、20.6%和 21.5%。在 NAS 下，性能其次的分别为 ER1[b] 和 M2，在 acc@n 的三个度量指标上也均有较大的提升。

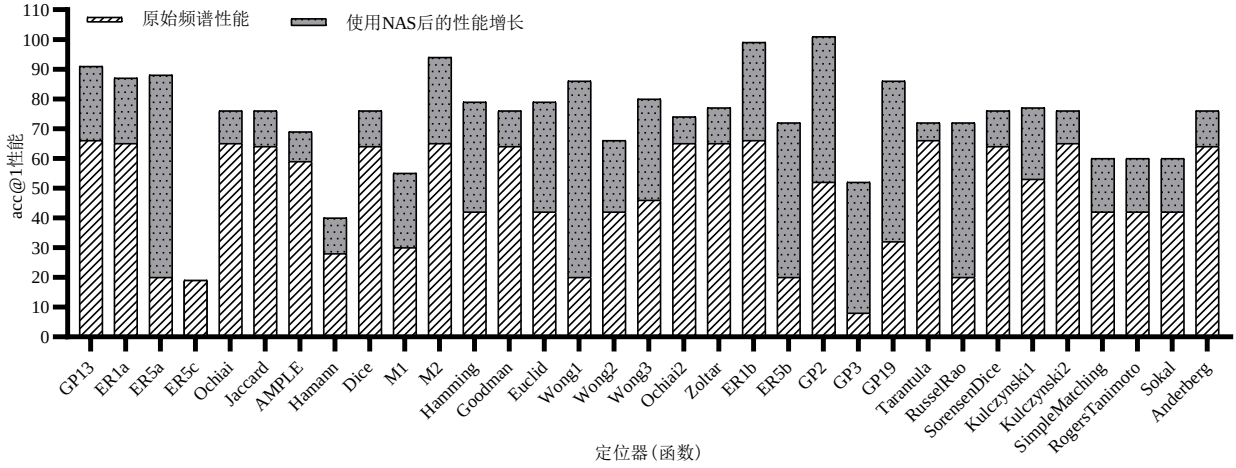


图 3　acc@1 指标性能增长

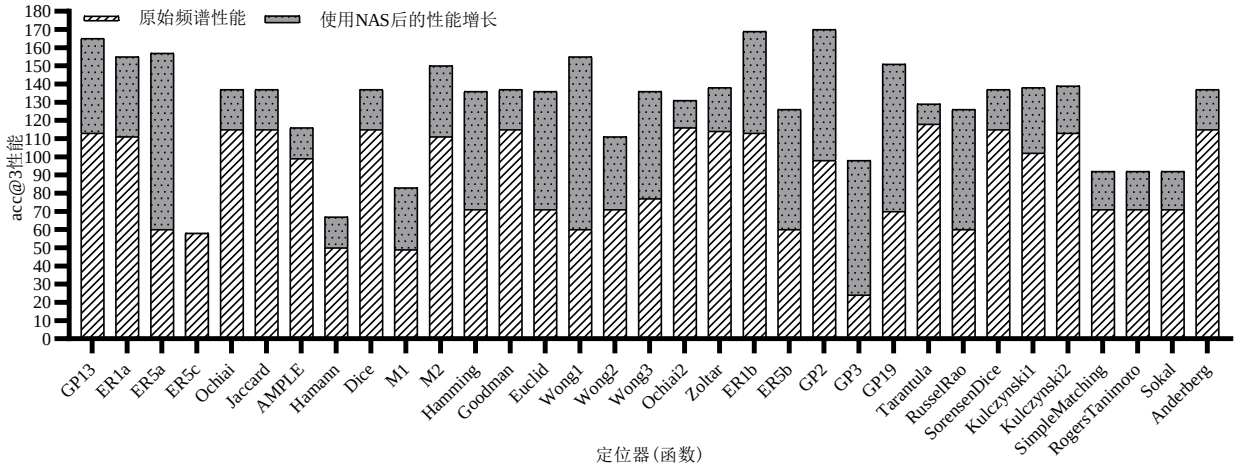


图 4　acc@3 指标性能增长

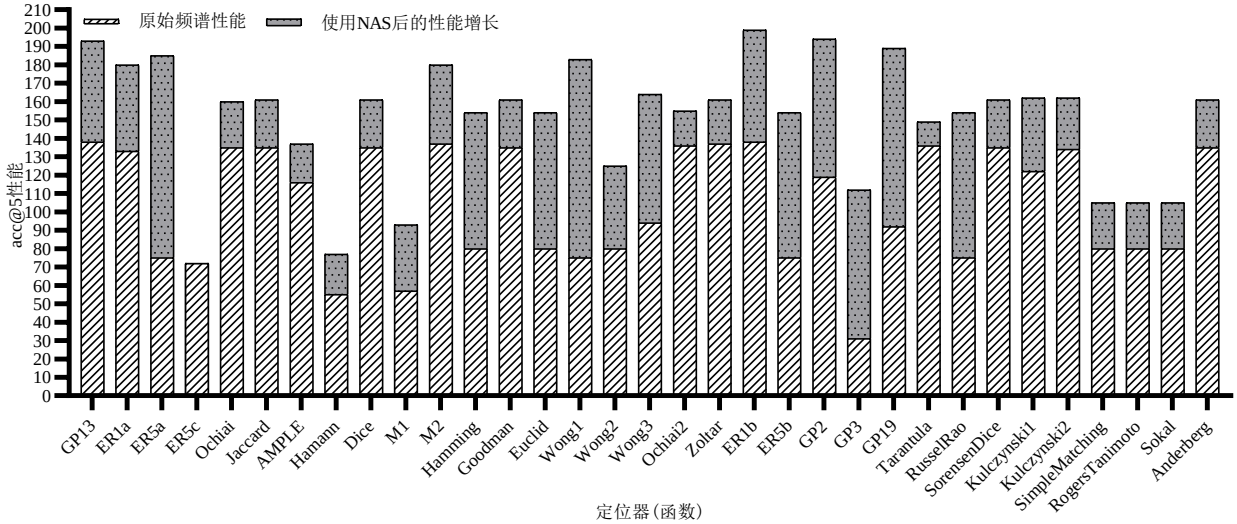


图 5　acc@5 指标性能增长

图 3 至图 5 分别为评价指标 acc@1、acc@3 和 acc@5 在使用网络聚合频谱 NAS 后的对比增长情况。除故障定位函数 ER5[c] 外，其余 32 个故障定位函数的三个指标性能都具有不同程度的增长，表明

所提出的网络聚合频谱具有较好的适应性，对于故障定位的性能提升效果是整体性的。

结合图表分析，对指标 acc@1，性能提升最大的是 ER5[a]，从 20 提升至 88，增加了 68，性能提升了 19.48%。对于指标 acc@3 和 acc@5，ER5[a] 的性能提升也是最大的，分别提升了 27.79%和 31.52。

从增长率上来说，取得最大增长率的故障定位函数为 GP3，其 acc@1 指标从 8 增长到了 52，性能提升了 12.6%，性能增长率为 550%；在 acc@3 和 acc@5 上也具有较大的增长率，分别为 308%和 261%。总体来说，33 个函数在 acc@n 上的平均增长率分别为 89.77%、59.07%和 55.22%。

以上对比分析，证明所提出网络聚合频谱在软件故障定位性能提升上是具有较好效果，软件实体之间的依赖关系分析对于软件故障定位研究具有重要意义，也证明了软件实体之间的依赖关系对于正确评估实体的故障可疑性具有一定的补充作用。

4.4.2 加权聚合性能分析

在计算网络聚合频谱时，对关联实体影响的聚合采取了加权叠加的方法。实际上，加权的这种策略是通过实际实验的对比分析得到的。下面对关联实体影响聚合采取加权和不加权的两种情况进行讨论，并通过实验结果分析来证明加权的有效性。

实验中，分别在三种不同聚合实体类型情况下对加权聚合和不加权聚合的效果进行了对比。在表 4 和表 5 中，Both 表示同时聚合 Caller 和 Callee 两种实体的影响，Caller 表示仅聚合 Caller 实体影响，Callee 表示仅聚合 Callee 实体影响；@1、@3 和@5 分别表示评价指标 acc@1、acc@3 和 acc@5。在图 6 和图 7 中，横坐标为聚合实体类型与度量指标的组合情况，如“Both@1”表示同时聚合两种实体影响下的 acc@1 性能，而图例“Without”和“With”分别表示不加权聚合和加权聚合。

表 4 和表 5 为各种情况下最优故障定位器性能指标和 33 个故障定位器的平均性能指标。实验数据表明，在所有情况中，采取加权聚合、仅聚合 Callee 实体影响的策略时，最优性能指标和平均性能指标同时达到了最佳的性能提升。图 6 和图 7 分别为采用加权和不加权策略时，最优故障定位器 acc@n 和故障定位器平均 acc@n 性能的对比图。

结合表 4 和表 5 分析可知，在各种聚合实体类型情况下，acc@1、acc@3 和 acc@5 三项指标均在采用加权聚合策略时取得最佳效果。在图 6 最优故障定位器的 acc@n 指标性能对比中，采用加权聚合

表 4 加权和不加权策略下最优故障定位器指标性能

聚合类型	是否加权	@1	@3	@5	MAP	MWE
原始频谱		66	118	138	0.320	70.65
Both	否	68	113	134	0.282	45.73
	是	90	159	184	0.389	22.94
Caller	否	56	119	137	0.309	82.87
	是	65	125	142	0.329	74.79
Callee	否	72	125	147	0.305	37.52
	是	**100**	**170**	**200**	**0.423**	**19.43**

表 5 加权和不加权策略下 33 个故障定位器平均指标性能

聚合类型	是否加权	@1	@3	@5	MAP	MWE
原始频谱		47.5	87.6	104.2	0.244	151.78
Both	否	36.6	71.3	90.1	0.199	159.77
	是	49.8	98.5	116.5	0.255	133.48
Caller	否	35.1	74.2	88.3	0.204	177.64
	是	43.6	84.9	101.1	0.237	158.10
Callee	否	41.9	78.4	96.8	0.217	156.3
	是	**54.2**	**101**	**120.7**	**0.265**	**131.13**

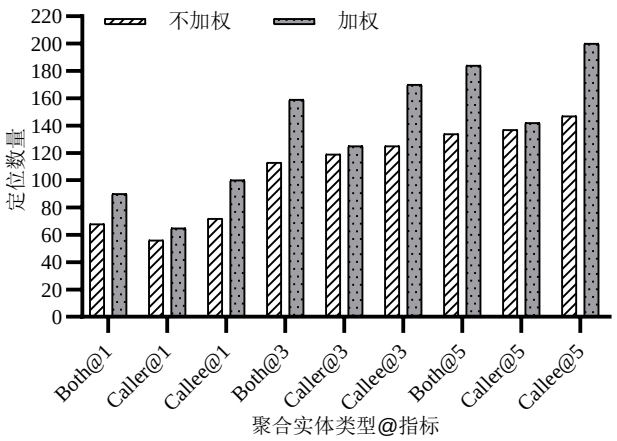


图 6 最优故障定位器 acc@n 指标性能对比

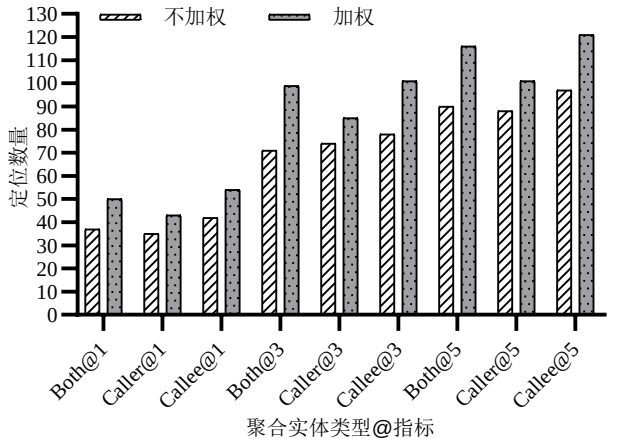


图 7 故障定位器平均 acc@n 指标性能对比

时 acc@n 三项指标都比不加权聚合好很多，且在仅聚合 Callee 实体影响时性能最优。图 7 对 33 个故障定位器的平均 acc@n 指标进行了对比。可以看出，同样是在采用加权聚合且仅聚合 Callee 实体影响时故障定位的性能最好。综上可知，在采用网络聚合频谱 NAS 时，采用加权聚合的方式更有利于改善软件故障定位的性能，但这种影响时建立在一定关联强度之上的，过度影响会产生相反效果。

4.4.3　聚合实体类型性能分析

根据网络聚合频谱定义和公式，进行聚合计算时，分别对目标函数的 Caller 函数和 Callee 函数两种实体类型进行了频谱聚合。由于两种实体在软件频谱依赖网络中描述的依赖关系不同，故实验进行了聚合不同实体类型的性能提升分析，分别是只聚合 Caller 函数实体影响、只聚合 Callee 函数实体影响和同时聚合 Caller 和 Callee 两种函数实体影响。

表 6 和表 7 为各种情况下最优故障定位器性能和 33 个故障定位的平均性能。分析可知，无论采取加权聚合还是不加权聚合，平均性能均在采用仅聚合 Callee 实体影响策略时达到最优。但不加权聚合时，平均指标性能比原始频谱更差。对于最优故障定位器指标性能，同样在仅聚合 Callee 实体影响时取得最优。唯一特殊的是，在不加权聚合且仅聚合 Callee 实体影响情况下，MAP 的指标值为 0.305，没有取得最优，但相差较小。因此，可以判断采用仅聚合 Callee 实体影响策略时，故障定位效果最好。

图 8 和图 9 分别为最优故障定位器指标性能和 33 个故障定位器平均指标性能的对比。可以看出，对于最优故障定位器，仅聚合 Callee 实体影响时带来的性能提升最大；而仅聚合 Caller 实体影响时的效果最差，还有可能带来负面影响。同样，在 33 个故障定位器上的平均指标性能对比中，也可以证实相同的结论。通过图 8 和 9 也可以证明，使用加权聚合的效果要不加权聚合具有更好的效果。

综上可知，在聚合实体类型的选择上，采取仅聚合 Callee 实体影响时的网络聚合频谱 NAS 具有最佳的性能提升。在软件函数实体的依赖关系中，Callee 函数是 Caller 函数的一份子，但 Callee 函数可以是多个 Caller 函数的一份子。因此，Callee 函数被测试用例执行的情况会比 Caller 函数更多，即 Callee 函数的故障可疑性会伴随着其贡献给 Caller 函数的复杂性产生关联影响。对于 Caller 函数，它的测试频谱向量只代表了它自身的故障倾向性，而不会对 Callee 函数产生影响。

表 6　不同聚合实体类型下最优故障定位器指标性能

是否加权	聚合类型	@1	@3	@5	MAP	MWE
原始频谱		66	118	138	0.320	70.65
否	Both	68	113	134	0.280	45.73
	Caller	56	119	137	**0.309**	82.87
	Callee	**72**	**125**	**147**	0.305	**37.52**
是	Both	90	159	184	0.389	22.94
	Caller	65	125	142	0.329	74.79
	Callee	**100**	**170**	**200**	**0.423**	**19.43**

表 7　不同聚合实体类型下 33 个故障定位器平均指标性能

是否加权	聚合类型	@1	@3	@5	MAP	MWE
原始频谱		47.5	87.6	104.2	0.244	151.78
否	Both	36.6	71.3	90.1	0.199	159.77
	Caller	35.1	74.2	88.3	0.204	177.64
	Callee	**41.9**	**78.4**	**96.8**	**0.217**	**156.29**
是	Both	49.9	98.5	116.5	0.255	133.48
	Caller	43.6	84.9	101.1	0.237	158.10
	Callee	**54.1**	**101**	**120.7**	**0.265**	**131.13**

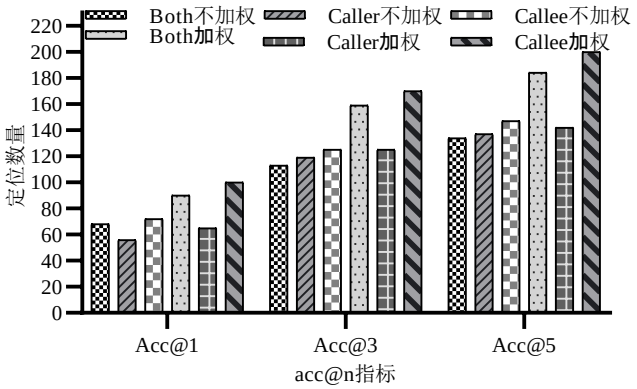


图 8　最优故障定位器 acc@n 指标性能对比

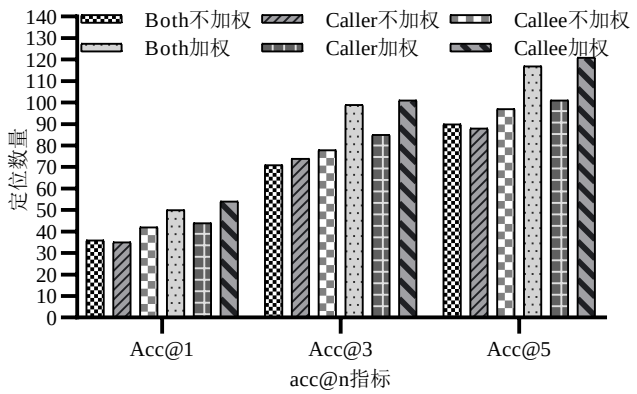


图 9　故障定位器平均 acc@n 指标性能对比

4.4.4　不同指示因子性能分析

在测试频谱向量中，四个指示因子e_f、e_p、n_f和

n_p分别从不同角度反映故障可疑性。因此，网络聚合频谱中，默认对这四个指示因子都进行聚合计算。但四个因子的聚合计算并非都对故障定位性能提升有所帮助，故在实验中对四个因子进行了部分聚合的对比实验，以找出最优组合。

综合上述分析可知，目前最佳性能的聚合方法是采用加权、仅聚合Callee实体影响的策略，故进行聚合不同指示因子的性能对比实验时默认采取该策略。表8为在不同因子组合情况下，最优故障定位器性能和33个故障定位器平均性能的统计。

表 8 不同因子组合下最优故障定位器指标性能和 33 个故障定位器平均指标性能

指标因子选取	最优故障定位器指标性能					33 个故障定位器平均指标性能				
	acc@1	acc@3	acc@5	MAP	MWE	acc@1	acc@3	acc@5	MAP	MWE
原始频谱	66	118	138	0.320	70.65	47.5	87.6	104.2	0.244	151.78
e_f	97	168	198	0.417	19.56	71.8	126.3	147.5	0.329	100.97
e_p	93	152	181	0.394	39.91	40.5	74.6	87.42	0.202	219.98
n_f	66	117	138	0.323	73.99	47.3	86.6	102.8	0.243	160.27
n_p	93	162	189	0.406	39.69	57.3	102.2	120.1	0.277	136.82
e_f+e_p	100	169	198	0.415	**19.43**	55.4	102.7	122.2	0.269	129.85
e_f+n_f	97	168	196	0.417	19.56	72.1	123.9	145.2	0.328	106.62
e_f+n_p	**101**	**170**	199	**0.423**	19.56	73.5	**128.4**	**150.5**	**0.333**	**89.54**
e_p+n_f	93	152	181	0.394	39.91	40.3	73.9	86.5	0.202	225.25
e_p+n_p	97	166	199	0.417	34.25	43.6	81.6	95.8	0.218	199.36
n_f+n_p	93	162	189	0.407	39.69	57.2	100.9	118.9	0.276	143.60
e_f+e_p+n_f	100	169	198	0.415	**19.43**	55.7	102.4	122	0.269	137.59
e_f+e_p+n_p	100	**170**	**200**	**0.423**	**19.43**	53.8	101.3	121.2	0.265	123.92
e_f+n_f+n_p	**101**	**170**	199	0.423	19.56	**73.7**	125.9	149	0.331	95.08
e_p+n_f+n_p	97	166	199	0.417	34.27	43.7	80.9	94.8	0.217	204.57
全部	100	170	**200**	**0.423**	**19.43**	54.2	101	120.7	0.265	131.1

如表8所示，对于最优故障定位器指标性能，除仅对n_f因子聚合的情况外，其他各种组合都能够显著提升各项指标的性能。其中，acc@1、acc@3和acc@5的最佳性能分别达到了101、170和200，最佳MAP为0.423，最小的MWE为19.43。在各种不同因子组合中，e_f+n_p组合、e_f+e_p+n_p组合、e_f+n_f+n_p组合和“全部”组合，均在不同指标上达到性能最优。但没有一组因子组合能够在五项指标上均取得最优。

在33个故障定位器平均指标性能上，10组因子组合能够提升性能，而另外5组因子组合对性能提升产生了不利影响。其中，e_f+n_p组合在四项指标上获得最佳性能，e_f+n_f+n_p组合仅在acc@1指标上取得最优。但对于acc@1指标，e_f+n_p组合的性能值为73.5，与最优的73.7相差不多。

综上，将e_f+n_p组合选定为最佳指示因子组合。它在最优故障定位器指标性能评估和33个故障定位器平均指标性能评估中取得最优的情况最多；而对于没有取得最优的指标上，其性能与最优性能也相差较小。四个因子中，表示执行实体的失败测试用例个数的e_f作用最大，n_p次之，而e_p具有负作用。即，在软件频谱依赖网络中，函数实体之间的影响主要表现在e_f和n_p上，其他指示因子上的影响较小。

5 相关工作

本文研究内容涉及基于频谱的故障定位、软件网络建模及分析，相关工作将从以下两方面展开。

5.1 基于频谱的软件故障定位

基于频谱的故障定位技术是一种最常用和轻量级的技术，也叫基于覆盖的故障定位[37]和统计调试[38]，主要依据测试用例执行成功或失败的结果，来对实体进行可疑性排序。基于频谱的故障定位包括三步[36]：(1) 构建软件测试覆盖矩阵；(2) 推导测试频谱；(3) 使用故障定位器进行计算可疑性评分。目前大部分工作集中要第(3)步故障定位器的改进，近些年也有一些工作致力于改进测试频谱的推导。

在改进测试频谱方面，2018 年，Laghari 等人[39]中提出了一种基于序列分析的测试频谱计算方法，该方法使用滑动窗口将某函数其内部所调用函数的执行序列切分成多个子序列，然后以子序列为单元计算测试频谱和可疑性评分，并将子序列评分的最大值作为该函数的可疑性评分。相似地，Laghari 在文献[36]中采用类似的方法，使用频繁项集挖掘方法改进了测试频谱的计算。2017 年，Zhang 等人[40]通过考察不同测试用例对测试频谱的不同贡献，使用 PageRank 构建了一种加权的测试频谱计算方法。而在本文中也使用了 PageRank 算法，不同的是本文中的 PageRank 算法用于分析软件实体间的影响力，优化故障定位函数的计算。

故障定位器，也叫故障定位函数或公式，以测试频谱为输入，输出软件实体的故障可疑性评分。Jones 等人[9]提出了第一个基础排序公式 Tarantula，其直觉在于被失败测试用例执行次数多且被成功测试用例执行次数少的实体可疑性更大。2007 年，Abreu 等人[5]将分子生物领域中的 Ochiai 引入到基于频谱故障定位中，取得了较好性能。2012 年，Yoo 等人[12]使用遗传编程方法演化出了 GP01…GP30 等评分函数。相似地，Sohn 等人[27]也使用遗传编程算法，结合了“Code and Change”度量改进了基于频谱故障定位方法。到目前为止，研究所提出的各类故障定位函数已多达几十个。

5.2 软件网络建模及分析

近年来，越来越多的研究聚焦于基于复杂网络的软件分析模型构建，旨在系统地考察系统的整体行为模式，挖掘软件运行时特性。其中，软件网络作为一种动力学统计模型，受到了众多关注。2015 年，Qu 等人[41]构建了 Calling Network，其聚合了一系列动态调用子图形成动态调用网，以函数为节点，调用关系为边，并将调用频度作为边的权重。同年，Qu 等人又在文献[42]中对软件网络的社区结构进行了研究，以衡量面向对象软件的内聚性。相似地，在 2013 年，Pan 等人[43]在软件网络上进行了社区检测研究，目的在于改善软件重构问题。在 2010 年，Pan 等人[44]通过建立加权的软件调用网络，分析了 bug 在其中的传播特性。

在软件故障定位方面，Zakari 等人[45]通过分析语句间的交互关系建立网络，并基于度中心性和介数中心性提出了故障定位方法。就目前来说，使用复杂网络的方法进行故障定位的研究还很少。文献[45]虽然提出了基于复杂网络理论的故障定位方法，但并未与基于频谱的技术进行结合。

综上，本文将软件网络建模分析与基于频谱的软件故障定位相结合，从优化软件测试频谱的角度出发，提出聚合关联实体影响综合评估实体故障可疑性的方法，增强测试频谱度量故障可疑程度的准确性和显著性。

6 结束语

针对现有基于频谱的故障定位方法仅考虑测试覆盖情况而忽略软件实体间依赖关系分析的缺点，本文从软件动态行为分析出发，提出了一种网络聚合频谱的测试频谱计算方法，来增强测试频谱指示因子体现软件实体故障可疑性的准确性和显著性。为了计算目标函数在关联函数实体影响下的综合频谱向量，根据动态函数调用关系构建了软件频谱依赖网络 SSDN，并提出了故障关联强度 FR 来度量关联函数实体间的影响关系，最后给出了网络聚合频谱 NAS 的计算方法。通过建立软件频谱依赖网络，分析和量化软件实体之间的调用依赖关系，为软件测试频谱的改进提供了一种网络表征模型。由于在一定程度上弥补了测试用例组件无法覆盖实体所有执行情况的不足，网络聚合频谱在软件故障定位的改进上取得较好的效果。实验在 33 个故障定位函数上进行实验，证明了所提出频谱计算方法的有效性。

参 考 文 献

[1] Ghanavati M. Automated fault localization in large Java applications [Doctoral dissertation]. Heidelberg University, Germany, 2019.

[2] Lehman M M. Programs, life cycles, and laws of software evolution. Proceedings of the IEEE, 1980, 68(9): 1060-1076.

[3] Collofello J S, Woodfield S N. Evaluating the effectiveness of reliability assurance techniques. Journal of Systems and Software, 1989, 9(3): 191-195.

[4] Tassey G. The economic impacts of inadequate infrastructure for software testing. North Carolina: National Institute of Standards and Technology, RTI Project: 7007.011, 2002.

[5] Rui A, Zoeteweij P, Van Gemund A J. On the accuracy of spectrum-based fault localization//Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, IEEE. Windsor, UK, 2007: 89-98.

[6] Xie X, Chen T Y, Kuo F C, Xu B. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Transactions on Software Engineering and Methodology, 2013, 22(4):

31.

[7] Harrold M J, Rothermel G, Sayre K, Wu R, Yi L. An empirical investigation of the relationship between spectra differences and regression faults. Software Testing Verification and Reliability, 2000, 10(3): 171-194.

[8] Valentin D, Lindig C, Zeller A. Lightweight bug localization with AMPLE//Proceedings of the sixth international symposium on Automated analysis-driven debugging. California, USA, 2005: 99-104.

[9] James J A, Harrolld M J, Stasko J. Visualization of test information to assist fault localization//Proceedings of the 24th International Conference on Software Engineering. Orlando, USA, 2002: 467-477.

[10] Naish L, Lee H J, Ramamohanarao K. A model for spectra-based software diagnosis. ACM Transactions on software engineering and methodology, 2011, 20(3): 11.

[11] Xie X, Kuo F C, Chen T Y, Yoo S, Harman M. Provably optimal and human-competitive results in sbse for spectrum based fault localization//International Symposium on Search Based Software Engineering. Berlin, Heidelberg, 2013: 224-238.

[12] Yoo S. Evolving human competitive spectra-based fault localisation techniques//International Symposium on Search Based Software Engineering. Berlin, Heidelberg, 2012: 244-258.

[13] Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst M D, Pang D, Keller B. Evaluating and improving fault localization//Proceedings of the 39th International Conference on Software Engineering. Buenos Aires, Argentina, 2017: 609-620.

[14] Papadakis M, Le Traon Y. Metallaxis-fl: mutation-based fault localization. Software Testing, Verification and Reliability, 2015, 25(5-7): 605–628.

[15] Moon S, Kim Y, Kim M, Yoo S. Ask the mutants: Mutating faulty programs for fault localization//2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. OH, USA, 2014: 153–162.

[16] Papadakis M, Le Traon Y. Using mutants to locate ‘unknown’ faults//2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. Montreal, Canada, 2012: 691-700.

[17] Zhang L, Zhang L, Khurshid S. Injecting mechanical faults to localize developer faults for evolving software. ACM SIGPLAN Notices, 2013, 48(10): 765-784.

[18] Agrawal H, Horgan J R, London S, Wong W E. Fault localization using execution slices and dataflow tests//Proceedings of Sixth International Symposium on Software Reliability Engineering. Toulouse, France, 1995: 143–151.

[19] Renieres M, Reiss S P. Fault localization with nearest neighbor queries//18th IEEE International Conference on Automated Software Engineering. Piscataway, USA, 2003: 30–39.

[20] Mao X, Lei Y, Dai Z, Qi Y, Wang C. Slice-based statistical fault localization. Journal of Systems and Software, 2014, 89: 51-62.

[21] Xuan J, Monperrus M. Test case purification for improving fault localization//Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. Hong Kong, China, 2014: 52-63.

[22] Zhou J, Zhang H, Lo D. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports//2012 34th International Conference on Software Engineering (ICSE). Zurich, Switzerland, 2012: 14–24

[23] B Le T D, Lo D, Le Goues C, Grunske L. A learning-to-rank based fault localization approach using likely invariants//Proceedings of the 25th International Symposium on Software Testing and Analysis. Saarbrücken, Germany, 2016: 177-188.

[24] Feng M, Gupta R. Learning universal probabilistic models for fault localization//Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. Toronto, Canada, 2010: 81-88.

[25] Xuan J, Monperrus M. Learning to combine multiple ranking metrics for fault localization//2014 IEEE International Conference on Software Maintenance and Evolution. Victoria, Canada, 2014: 191-200.

[26] Ribeiro H L, de Souza H A, Araujo R P D A, Chaim M L, Kon F. Evaluating data-flow coverage in spectrum-based fault localization//2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. Porto de Galinhas, Brazil, 2019: 1-11.

[27] Sohn J, Yoo S. Fluccs: Using code and change metrics to improve fault localization//Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. Santa Barbara, USA, 2017: 273-283.

[28] Laghari G, Demeyer S. On the use of sequence mining within spectrum based fault localization//Proceedings of the 33rd Annual ACM Symposium on Applied Computing. Pau, France, 2018: 1916-1924.

[29] Parnin C, Orso A. Are automated debugging techniques actually helping programmers?//Proceedings of the 2011 international symposium on software testing and analysis. Toronto, Canada, 2011: 199-209.

[30] Yoo S, Xie X, Kuo F C, Chen T Y, Harman M. No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. RN, 2014, 14(14): 14.

[31] Binder R. Testing object-oriented systems: models, patterns, and tools. UAS, Addison-Wesley Professional, 2000.

[32] Wasylkowski A, Zeller A, Lindig C. Detecting object usage anomalies//Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. Dubrovnik, Croatia, 2007: 35-44.

[33] Kochhar P S, Xia X, Lo D, Li S. Practitioners' expectations on automated fault localization//Proceedings of the 25th International Symposium on Software Testing and Analysis. Saarbrücken, Germany, 2016: 165-176.

[34] Reps T, Ball T, Das M. The use of program profiling for software maintenance with applications to the year 2000 problem//Proceedings of the Sixth European Software Engineering Conference. Heidelberg, Germany, 1997: 432-449.

[35] Just R, Jalali D, Ernst M D. Defects4J: a database of existing faults to enable controlled testing studies for Java programs//Proceedings of the 2014 International Symposium on Software Testing and Analysis. San Jose, USA, 2014: 437-440.

[36] Laghari G, Murgia A, Demeyer S. Fine-tuning spectrum based fault localisation with frequent method item sets//Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. Singapore, Singapore, 2016: 274-285.

[37] DiGiuseppe N, Jones J A. On the influence of multiple faults on coverage-based fault localization//Proceedings of the 2011 international symposium on software testing and analysis. Toronto, Canada, 2011: 210-220.

[38] Zheng A X, Jordan M I, Liblit B, Naik M, Aiken A. Statistical debugging: simultaneous identification of multiple bugs//Proceedings of the 23rd international conference on Machine learning. Pittsburgh, USA, 2006: 1105-1112.

[39] Laghari G, Demeyer S. On the use of sequence mining within spectrum based fault localisation//Proceedings of the 33rd Annual ACM Symposium on Applied Computing. Pau, France, 2018: 1916-1924.

[40] Zhang M, Li X, Zhang L, et al. Boosting spectrum-based fault localization using PageRank//Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. Santa Barbara, USA, 2017: 261-272.

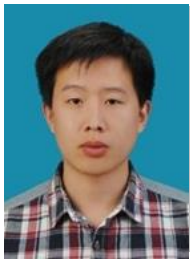
[41] Qu Y, Guan X, Zheng Q, Liu T, Zhou J, Li J. Calling network: A new method for modeling software runtime behaviors. ACM SIGSOFT Software Engineering Notes, 2015, 40(1): 1-8.

[42] Qu Y, Guan X, Zheng Q, Liu T, Wang L, Hou Y. Exploring community structure of software call graph and its applications in class cohesion measurement. Journal of Systems and Software, 2015, 108: 193-210.

[43] Pan W F, Jiang B, Li B. Refactoring software packages via community detection in complex software networks. International Journal of Automation and Computing, 2013, 10(2): 157-166.

[44] Pan W F, LiB M Y T. Measuring structural quality of object-oriented softwares via bug propagation analysis on weighted software networks. Journal of Computer Science and Technology, 2010, 25(6): 1202-1213.

[45] Zakari A, Lee S P, Chong C Y. Simultaneous localization of software faults based on complex network theory. IEEE Access, 2018, 6: 23990-24002.



HE Hong-Dou, born in 1991, Ph.D. candidate. His research interests include software fault localization and software dynamic behavior analysis.

REN Jia-Dong, born in 1967, Ph.D., professor. His research interests include data mining and software security.

ZHAO Gu-Yu, born in 1993, Ph.D. candidate. Her research interests include data mining and the complex network.

HE Hai-Tao, born in 1968, Ph.D., professor. Her research interests include data mining and software security.

HUANG Guo-Yan, born in 1969, Ph.D., professor. His research interest is software security.

Background

With the rapidly increasing size and complexity of software system, software debugging and fault localization are facing unprecedented challenges. Statistics show that attempts to fix inevitable bugs are estimated to consume 50% to 80% of the development and maintenance effort. Therefore, automated software fault localization techniques are urgently needed in the field of software engineering.Among existing automated fault localization techniques, Spectrum-Based Fault Localization (SBFL) is a promising one due to its relatively low overhead in test execution and comparable performance compared with the other techniques. While, recently SBFL techniques have been proved that they perform disappointingly performance in locating real-world faults. It seems that SBFL have reached to a bottleneck, and is it very promising to mining additional information from program spectra.

Considering that existing studies mainly inspect the test coverage and ignore the analysis of correlations between software entities. In the paper, an optimized Network Aggregated Spectrum (NAS) based fault localization approach is proposed, which aggregates the influences of associated entities to synthetically estimate the test spectrum. The intuition of our approach is to mine additional faulty suspiciousness from the interactive behaviors between software entities. To verify the effectiveness of the proposed approach, 33 SBFL fault formulas are tested using the benchmark dataset of Defects4J. The experimental results show that our proposed approach can improve almost all the performance of the 33 formulas except for ER5^c. For the metrics of acc@1, acc@3 and acc@5, the best localization formula achieves the improvements of 10%, 14.9% and 16.6%.

This work is supported by the National Natural Science Foundation of China (No.61772449, No.61572420, No.61807028, No.61802332) and the Natural Science Foundation of Hebei Province of China (No.F2019203120).