西北工業大學

本科毕业设计论文

题 目 频谱故障定位测试套件优化方法

专业名称 软件工程专业

学生姓名 刘书敏

指导教师 马春燕

毕业时间 2019 年 6 月

毕业 设计 论文 任务书

一、题目

频谱故障定位测试套件优化方法

二、研究主要内容

对测试套件质量评估进行研究，以改进光谱故障定位技术。我们提出了一个理论框架来衡量 SFL 技术的测试套件质量。由于 O 和 Op 是两种使用最广泛的 SFL，因此主要使用这两种技术评估我们的框架。

研究内容：

1）提出一个理论评估框架，该框架提供了一种可重复且客观的方法，可以在单一故障情景中系统地调查和评估测试套件。提出了一种向量表模型，用于概括使用任何程序和相应测试套件的测试覆盖率信息。基于向量表模型，我们可以使用一种普遍的方式通过 O 和 Op 计算程序语句的可疑因子。为了定义不同类型的错误语句，我们根据程序语句的可疑因子划分程序语句的子集。在此基础上，进一步提出了一个平均性能测量公式来评估和比较 O 和 Op 的测试套件的质量。

2）通过提出的理论框架，我们通过优化 O 和 Op 的测试套件，在三个已经被广泛研究的方面探讨了提高频谱故障定位性能的方法，具体方面如下。大多数现有研究都侧重于其中一个方面，所提出的框架可以通过量化测试套件的质量测量来为改善 SFL 性能铺平道路。拟议的框架将衡量测试套件的质量，为许多与测试套件质量相关的研究和实践活动提供平台。

（1）提供一些方法和案例研究，来生成、选择或添加测试用例以生成新测试套件，着重于 O 和 Op。并且明确地确定了不同参数因素对于添加测试用例的性能的影响。

（2）从测试套件中移除部分对结果没有影响的测试用例以改善故障定位性能，并且给出了为已有的测试套件找到最佳子集的方法和案例研究。

（3）通过分析相应测试用例定位潜在故障的能力，确定测试用例的权重，以区分不同测试用例对提高错误定位性能的贡献。

三、主要技术指标

1）本实验程序能够从测试用例的运行结果计算出向量表模型和测试用例的

权重。

2）本实验程序能够达到优化测试套件的效果，能够在两个测试数据集（SIR 和 Defects4j） 上达到平均效应量大于 0 的效果。

四、进度和要求

2018 年 12 月 24 日－2019 年 1 月 17 日：完成开题报告。

2019 年 1 月 17 日－2019 年 3 月 16 日：实现测试框架、覆盖率矩阵和可疑因子计算。

2019 年 3 月 16 日－2019 年 4 月 15 日：实现平均代价和测试用例权重计算。

2019 年 4 月 15 日－2019 年 5 月 15 日：实现加权计算和测试用例划分。

2019 年 5 月 15 日－2019 年 6 月 11 日：论文审阅、修改、答辩。

五、主要参考书及参考资料

[1] Naish, L., Lee, H. J., and Ramamohanarao, K., 2011. A model for spectra-based software diagnosis. ACM Transactions on Software Engineering and Methodology 20, 3, pp. 11:1-11:32.

[2] Naish L.,Lee H. J., 2013. Duals in Spectral Fault Localization, 22nd Australian Conference on Software Engineering, pp.51-59.

[3] Wong, W. E., Qi, Y., Zhao, L., and Cai, K. Y., 2007. Effective fault localization using code coverage. In Proceedings of the 31st Annual International Conference on Computer Software and Applications. Beijing, China, pp. 449-456.

[4] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E., 2002. Pinpoint: Problem determination in large, dynamic internet services. In Dependable Systems and Networks. DSN 2002. Proceedings. International Conference on . IEEE.,pp. 595-604.

学生学号 2015303087　　学生姓名 刘书敏

指导教师 _____　　系 主 任 _____

摘　要

频谱故障定位是一种通过统计被测程序的各条语句在各个测试用例中的运行次数，从而自动定位故障位置的一种自动调试方法。它使用风险评估公式来从动态收集的测试信息中计算语句的风险排名，从而定位故障。为了能够从测试套件中找出对故障定位最有效的测试用例，本文提出了一种测试套件优化方案，从而提升故障定位的准确率，并通过缩减无关的测试用例的方法来降低开发人员运行测试套件所花费的时间。本文提出的理论框架首先使用向量表模型来抽象概括测试套件运行时所产生的语句覆盖信息和测试结果，然后使用任意的故障定位公式（SFL）来计算每条语句的可疑因子，计算出定位故障所需的平均排名代价（average performance），从而使用贪心算法来缩减使排名代价升高的测试用例。

为了验证本文中提出方案的有效性，我们通过实验程序，在两个程序集（SIR 和 Defects4j）和 9 个故障定位公式上运行了我们的优化方案，并使用效应量（Effect Size）来作为最终结果的度量。

综上, 本文主要做的工作有

1. 提出了一个向量表模型，和一个基于平均排名代价的测试套件质量评估方案。

2. 提出了一个基于贪心算法的测试套件优化方案。

3. 使用 Java 实现了一个辅助工具，使其能够运行两个测试数据集 SIR 和 Defects4j 上的测试对象，收集其覆盖信息，并生成向量表模型、计算可疑因子、平均排名代价。并且能够运行之前提出的贪心算法来优化测试套件。

4. 在两个被广泛使用的测试数据集 SIR 和 Defects4j 上应用我们的辅助工具，生成优化前后的统计信息，使用效应量来量化分析实验结果，并且以图表的形式对结果进行展示。

关键词: 故障定位, 频谱故障定位, 测试套件优化

Abstract

Spectral fault localization is an automatic fault-localization technique to expedite debugging, which uses risk evaluation formula to rank the risk of fault existence in each program entity after dynamically collecting testing information. To assess the potential usefulness of test suite for spectral fault localization, a theoretical evaluation framework for test suite quality optimising is proposed in this paper, which improves the accuracy of fault localization, and saves test suites executing time by removing non-relative testcases. A vector table model is firstly used to generalize coverage information and test results produced in testing, a widely choices of SFL techniques are then applied to it, in order to resolve the suspiciousness factors of each statements in the subject program. Average performance of the test object is calculated, before removing unnecessary testcases which make average performance increasing, by applying greedy algorithm.

In order to verify the effectiveness of this method, 2 databases of test objects and testcases (SIR and Defects4j) and 9 SFL techniques are used in the experiment, with effect size used in the measurement of the effectiveness in result.

To sum up, this paper works on those

1. Proposed a vector table model and a soluation to measure test suites quality, based on average performance.
2. Proposed a new method to optimise test suite quality, as greedy algorithm.
3. Developed a experiment software in Java, which can execute test objects, collect the coverage information and generate vector table model, calculate suspiciousness factors and average performances. Besides that, it can also optimise the test suites by the greedy algorithm metioned above.
4. Applied the software on two widely used databases to generate the statistics results before and after optimising and analyze the result using effect size, with diagram presentation.

Key Words: fault localization, spectral fault localization, test suites optimising

目 录

第 一 章　绪论

人工的故障定位和 debug 操作需要使用断点和查看变量来确定程序的执行状态，非常的费时费力。所以，人们很需要一种自动化的故障定位方法来减少调试过程的成本。

近年来，作为自动定位技术的频谱故障定位（SFL）已被广泛研究。该方法使用风险评估公式，在动态收集测试信息后对每个程序实体中的故障存在风险进行排序。测试信息包括两个方面：每个实体的执行覆盖范围（例如语句、函数和代码块），以及每个测试用例的执行结果（即通过或失败）。然后，所有实体将根据其可疑因子排名按升序排序。然后根据从上到下的顺序对实体进行调试，直到找到故障。对于不同的程序实体，语句和代码块特别受到关注。在不失一般性的情况下，本研究将实体视为先前 SFL 研究中的语句。

为了使 SFL 技术有效和有用，应该考虑两件事。一是提出新的和有效的 SFL 技术。到目前为止，多年来已经提出了 50 多种 SFL 技术，如 O 和 Op[1]，DE（J）和 DE（C）[2]，Wong1[3]，Jaccard[4]，Kulczynski1[5]，D*[6]，等等。二是文献 [7–16] 中根据已建立的实验设置和基准（如西门子套件）采用经验方法，通过了解测试套件的质量来改善现有 SFL 技术的性能。目前，大多数现有的测试套件质量调查方法都很大程度上取决于实验设置和样本程序，因为输入的参数包括故障类型，测试用例数，故障代码的执行时间以及执行路径都被分配成了特定的值，不能轻易重新分配。它缺乏一个理论框架，可以概括评估过程并且判定评估结果的合格性。因此，有必要进行系统的研究，以了解当前的实证结果，提高 SFL 的性能，并以定性和准确的方式揭示 SFL 的基本原理。

对测试套件质量评估进行研究，以改进频谱故障定位技术。本文提出了一个理论框架来衡量 SFL 技术的测试套件质量。由于 O 和 Op 是两种使用最广泛的 SFL，因此主要使用这两种技术评估本文的框架。

本文章的主要贡献如下：

1. 提出了一个向量表模型，和一个基于平均排名代价的测试套件质量评估方案。

2. 提出了一个基于贪心算法的测试套件优化方案。

3. 使用 Java 实现了一个辅助工具，使其能够运行两个测试数据集 SIR 和 Defects4j 上的测试对象，收集其覆盖信息，并生成向量表模型、计算可疑因子、平均排名代价。并且能够运行之前提出的贪心算法来优化测试套件。

4. 在两个被广泛使用的测试数据集 SIR 和 Defects4j 上应用我们的辅助工具，生成优化前后的统计信息，使用效应量来量化分析实验结果，并且以图表的形式对结果进行展示。

第 二 章　相关技术

在本文的研究中，使用了如下的 SFL 技术和定位代价评估技术。

2.1 频谱故障定位 (SFL)

在 SFL 中，向量 $\langle a_{nf}, a_{np}, a_{ef}, a_{ep}\rangle$ 表示包含有关单个语句的执行信息的四个元素，其中下标的第一部分指示相应的语句是执行（e）或不执行（n），第二部分指示测试是通过（p）还是失败（f）。例如，给定测试套件和程序 PG，PG 中的语句 S 的 a_{ep} 是执行过该语句并且结果成功的测试用例数。每种 SFL 技术都有自己的风险评估公式，该公式使用语句的向量来计算该语句的可疑因子。

例如，两个 SFL 公式 O 和 Op 分别使用如下两个等式2.1.1和2.1.2来表示。

$$O(a_{nf}, a_{np}, a_{ep}, a_{ef}) = \begin{cases} -1(a_{nf} > 0) \\ a_{np}(otherwise) \end{cases} \tag{2.1.1}$$

$$O^p(a_{nf}, a_{np}, a_{ep}, a_{ef}) = a_{ef} - \frac{a_{ep}}{p+1} \tag{2.1.2}$$

考虑图 2-1 中的示例。一个程序 PG 有 10 条语句（从 s_1 到 s_{10}），测试套件 TS 有 9 个测试用例，从 t_1 到 t_9。具体来说，t_7 到 t_9 的运行结果是测试失败，其余六个测试用例则是测试成功，如二进制结果向量 OC 所示，它记录了 TS 的测试结果。在 OC 中，1 表示通过（测试用例执行成功），0 表示失败。矩阵 M 的第 i 行中的元素表示测试用例 t_1 到 t_9 中，语句 s_i 的测试覆盖率信息，其中 1 表示语句 s_i 在相应的测试用例中执行过，0 表示没有执行。矩阵 MA 定义如下：其第 i 行表示语句 s_i 在向量 $\langle a_{nf}, a_{np}, a_{ef}, a_{ep}\rangle$ 中相应的四个值。SF 是使用 SFL 公式 O 计算出的语句的可疑因子列表。例如，对于 s_3 来说，$a_{np} = 4$ 意味着有四个测试用例在没有执行 s_3 的情况下结果正确; $a_{ef} = 3$ 表示语句 s_3 在三个失败的测试用例中执行过；在 SF 中，s_3 的结果为 4，其可疑因子在所有语句中是最高的，而 s_2 和 s_8 的结果为 -1，其可疑因子是 SF 中最低的。SF 用于给语句 s_1 到 s_{10} 排序。在使用 O 进行计算时，示例的排名结果是 $\langle[s_3], [s_5, s_6, s_{10}], [s_1, s_4, s_7, s_9], [s_2, s_8]\rangle$（语句按照排名从高到低的顺序排列，每对 [] 标记的语句具有相同的排名）。

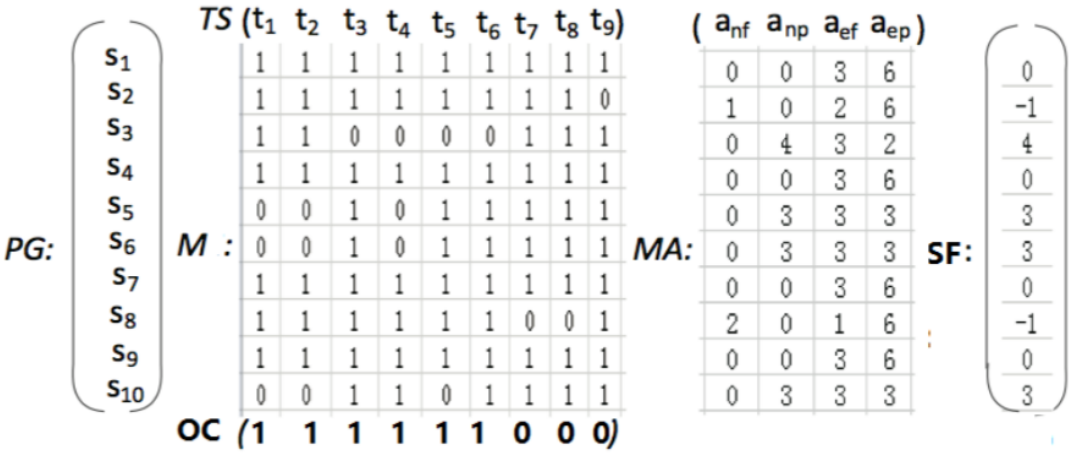

图 2-1　使用 SFL 技术进行计算的例子

2.1.1 SFL 的性能测量

目前，SFL 有两种性能测量的方式：

1. 得分函数（score function）[1]。如果在多组执行路径上应用 SFL 时，故障语句的排名低于其他语句，则 SFL 的分数为 0；如果有 $k-1(1<k)$ 条语句的可疑因子等于故障语句的可疑因子且排名最高，则 SFL 得分为 $\frac{1}{k}$。对于上述 s_3，当应用得分函数时，在公式 O 上的得分为 1。使用得分函数作为测量方式时，分数越高，SFL 的性能越好。

2. EXAM 分数（EXAM score）[8]。EXAM 分数是在达到第一个故障语句之前需要检查的源代码数量的百分比。在 SFL 的实际应用中，需要一个方案来确定具有相同可疑因子的语句的顺序。因此，对于不同的顺序确定方案，SFL 的 EXAM 分数是不同的。现有的研究中已经使用了各种确定顺序的方案，包括 WORST，BEST[8]，期望值 [17] 和排名成本 [2] 等等。BEST 假设认为，故障语句是在可疑因子相同的所有语句中检查到的第一条语句；WORST 假设认为，故障语句是相同可疑因子的所有语句中最后一个被检查的语句；使用期望值来确定顺序的公式为 $\frac{e+1}{k+1}$，其中 k 表示错误语句数量，e 表示错误语句中具有相同可疑因子的语句数量；排名成本则假定有缺陷的语句是在相同可疑因子的所有语句中，在中间检查到的语句。

对于 SFL 的性能测量而言，EXAM 分数更为合适，并且在 SFL 方面的大多数研究都使用 EXAM 或其等价物进行测量。得分函数不是常用的测量

方式。

在实践中，确定具有相同排名的语句的顺序时，基于期望值和排名成本的顺序确定方案是比 WORST 和 BEST 更合理的假设。因此，本文使用 EXAM 分数和排名成本作为性能的评价方法。一条语句的代价计算公式为

$$\frac{g+\frac{e}{2}}{n} \tag{2.1.3}$$

在等式2.1.3中，

1. g 是排名高于所有故障语句的正确语句的数量;
2. e 是和排名最高的故障语句排名相等的所有语句的数量，如果没有和排名最高的故障语句排名相等的正确语句，则 $e=0$。
3. n 是程序中的语句数。

例如，在图2-1中，如果假定 s_1 有故障，那么对于 s_1，$g=4$ 且 $e=3$，并且故障定位使用 O 进行计算，则结果为 $(4+4/2)/10=12/20$。计算出的值越小，代表 SFL 技术定位故障的性能越好。

2.2 国内外研究现状

目前，国内外就从已有的测试套件中减少测试用例以提高 SFL 技术的性能方面，已经有了一些研究。

Hao 等人 [13] 假定相似或冗余的测试用例会降低故障定位的有效性。他们进行了一项实证研究，证明了引入的冗余测试用例可能会削弱故障定位技术的有效性。他们的结果表明减少测试用例的数量可以提高故障定位效率。

在文献 [12] 中，对于使用的被测程序和测试套件，他们发现引入六个以上的失败测试用例或超过二十个通过的测试用例对频谱故障定位的效率产生的影响很小。

在文献 [11] 中，第一个实验使用了两种关于故障定位技术有效性的测试套件简化策略。

1. 基于语句的简化，它生成一个简化的测试套件，涵盖与原始套件相同的语句数量。
2. 基于向量的测试套件简化方法，其中被简化之后的测试套件的语句覆盖与原始测试套件的语句覆盖相同。

基于语句的测试套件简化方法对故障定位效率有显著的影响，而基于向量的却效果有限。该实验表明，对于四种频谱故障定位方式（$Tarantula, Ochiai, SBI, Jaccard$），其效率根据所使用的测试套件简化策略的变化而变化。

研究 [7] 使用了非冗余的测试用例来评估若干个故障定位方案。在他们提出的方法中，通过仅为成功和失败的类保留非冗余的测试用例，可以通过预先的实验设置，在包括 Op 在内的若干测量方式上改进 SFL 的有效性。

Masri 等人 [15] 提出了预测同时正确的测试用例并将其从测试套件中移除以提高 SFL 有效性的技术。

Zhang 等人。汇报了一项综合研究，以调查复制（并重复）失败的测试用例对典型 SFL 技术有效性的影响 [18]。

上述方法表明，一些测试套件可能是冗余的，只将原始测试用例的一些子集应用到测试中可以提高故障定位的有效性。通过删除冗余测试用例，故障语句的排名可能比以前更加靠前，但这种情况并不是一定会发生。

在接下来的文章中，我们将提出一种新的方法，用来削减冗余的测试用例，从而优化故障定位效果。

第 三 章　基于向量表模型的测试套件评估方法

3.1 假设及定义

在提供我们框架的一般概述之前，本文将首先列出从大多数先前的 SFL 研究中 [1, 18–21] 采用的一些假设。

1. 我们假设使用了 SFL 技术来进行测试的程序，它的每个测试用例运行结果要么成功，要么失败，没有第三种情况。先前的研究也采用了这种假设。

2. 我们假设调试器从 SFL 返回的排好序的可疑语句列表的顶部到底部逐个检查语句，并且一旦检查到了错误语句，就一定可以识别其错误。这也被称为“完美的错误检测”，这也是大多数以前的 SFL 研究所采用的假设。

3. 我们假设每次使用相同的输入来运行同一个测试用例时，其返回结果一定是相同的。也就是说，假如我们使用输入 a_1 来运行测试用例 t_1 并且得到结果 $o_1 = 0$，我们则假定每一次使用此输入来运行此测试用例时（在同一个实验对象上），每次都能得到结果 0。这同样也是大多数 SFL 研究所采用的假设。

4. 假设测试套件具有 100%的语句覆盖率，并且还假设测试套件包含至少一个通过的测试用例和一个失败的测试用例。

5. 在单个故障的被测程序中，错误语句必须被且只被所有失败的测试用例执行到。因此，$a_{nf} = 0$ 的单故障程序的每个语句都有 50%的错误概率。

在使用 SFL 技术时，对于一个现实中的故障程序，我们不可能知道具体的哪个测试用例能够用来更好地定位特定故障，因为我们不知道故障在哪里。当故障未知时，无法评估测试套件的效率和有效性。为了能够进行测试套件的质量评估，在本文中我们假设在一些语句中存在单个故障，即矩阵中所有的 a_{nf} 都等于 0（参见上面的假设 5），然后我们测量测试套件找到所有可能的错误语句的平均排名代价（average performance）。对于“*平均排名代价（average performance）*”的定义如下。

定义 1 (测试套件的平均排名代价（average performance）). *给定一个具有 n 条语句的程序 $PG =< s_1, s_2, ..., s_n >$、一个 SFL 和一个测试套件，平均排名代价是对于所有可能的错误语句，根据等式2.1.3计算的代价总和的平均值。*

对于图2-1的示例，语句 $s_1, s_3, s_4, s_5, s_6, s_7, s_9$ 和 s_{10} 是可能的错误语句，因为它们的 a_{nf} 等于零。相应的*平均排名代价*是 $(12/20 + 0 + 12/20 + 5/20 + 5/20 + 12/20 + 12/20 + 5/20)/8 = 63/160$（参考等式2.1.3）。应用于测试套件的 SFL 的*平均排名代价*值越小，根据2.1.1中 SFL 的性能测量方式，该测试套件的质量越高。

给定一个具有 n 条语句的程序 $PG =< s_1, s_2, ..., s_n >$，SFL 和测试套件 TS，为了描述我们拟议的用来评估测试套件质量的理论框架（如图3-1所示），我们首先给出 *I type* 语句和 *II type* 语句的定义。

定义 2 (*I type* 语句和 *II type* 语句). *对于 $\forall s_i(1 \leq i \leq n)$,，如果 $\exists j$, s_i 没有在 t_j 中执行，那么 s_i 即为 II type 语句，否则 s_i 是 I type 语句。*

基于上述假设和定义，我们的研究过程包含以下步骤。

1. 运行 PG 的测试套件 TS，导出 PG 的语句覆盖矩阵 M 和结果向量 OC，如图2-1所示的 M 和 OC。

2. 用 M 和 OC 计算 PG 中每个语句的向量 $< a_{nf}, a_{np}, a_{ef}, a_{ep} >$。然后构建在3.2节中提到的向量表模型。如图2-1中的 MA 所示。向量表模型用于概括表示测试覆盖率信息以构造类似于图2-1中的矩阵 MA。

3. 根据向量表模型，使用对应的 SFL 计算程序 PG 中语句的可疑因子。向量表模型中的 a_{nf}, a_{np}, a_{ef} 和 a_{ep} 变量将被传递给此 SFL 中的风险评估公式，然后使用该数学表达式来计算所有语句的可疑因子。最后可以得到一个包含了所有语句的可疑因子的列表，如图2-1中的 SF 所示。

4. 根据具体的可疑因子计算公式的特征来划分 PG 的 *II type* 语句的子集。在向量表模型中，分别基于 *I type* 和 *II type* 语句的集合来划分两个语句的子集。*I type* 语句的集合不需要进一步细分，因为它们 a_{nf}, a_{np}, a_{ef}, 和 a_{ep} 相同，基于其计算出的可疑因子也是相同的。所以，*II type* 语句需要进一步细分为不同的子集。

5. 根据上述 PG 语句的子集定义错误语句的类型。如果错误的语句是 *I type* 语句，则无论选择何种 SFL 技术，只要它使用 $a_{nf}, a_{np}, a_{ef},$ 和 a_{ep} 作为参数，我们就无法区分不同的错误语句，而只能把它们视为同一种错误类型。如果错误语句为 *II type* 语句，则可以针对上面 PG 的 *II type* 语句的子集进一步定义不同的错误类型。

6. 根据等式2.1.3，使用 TS 来计算每个可能的故障语句的代价，并使用所有不同类型的故障语句的平均代价公式（参见定义1），通过调查可疑因子在不同语句子集的数学表达式中的变化趋势来推导出 TS 的平均代价公式。

7. 将质量评估公式应用于优化和理解测试套件 TS 代表的各种程序，并指导提高 SFL 性能的活动。

3.2 向量表模型（Vector Table Model）

在建立向量表模型之前，我们需要明确以下几点输入。

1. 一个具有 n 条语句的程序 $PG =< s_1, s_2, ..., s_n >$，其中 s_i 表示第 i 条语句。

2. 一个具有 t 个测试用例的测试套件 $TS =< t_1, t_2, ..., t_t >$ 和其中每个测试用例对应的二进制结果组成的向量 $OC =< o_1, o_2, ..., o_t >$。其中 $o_i = 1$ 代表 i 号测试用例的运行成功，为 0 则代表运行失败。

3. 一个具有 $n \times t$ 个元素的测试覆盖矩阵 $M = [m_{ij}]$，其中 $m_{ij} = 1$ 表示在执行测试 t_i 时，语句 s_j 被执行过。为 0 则表示该条语句没有执行。

同时，我们设 t 个测试用例中，通过（执行成功）的测试用例数量为 p。

由于必须将频谱故障定位技术应用于给定的程序和给定的测试套件。所以变量 n, t 和 p 在给定的 PG 和 TS 中是固定的值。

如果 $s_i(1 \leq i \leq n)$ 在每个测试用例中都执行过，则该语句的向量 $< a_{nf}$,a_{np}, a_{ef}, $a_{ep} >$ 在测试套件 TS 中是 $< 0, 0, t-p, p >$。不管使用何种 SFL 技术，每个测试用例都执行的那些语句的可疑因子都不会有区别。受这种语句的启发，我们可以使用变量来表示语句的 a_{nf}, a_{np}, a_{ef}，和 a_{ep}，从而研究 SFL 技术的一些规则。我们构建了一个通用的系统模型，其中我们使用

变量来表示 PG 中每个语句的 a_{nf}, a_{np}, a_{ep}, a_{ef}，并且使用了已知变量 t 和 p。然后，可以将 SFL 技术应用于理论模型，通过将 a_{nf}, a_{np}, a_{ef}, a_{ep} 中的变量替换进相应的风险评估公式，来计算 PG 中语句的可疑因子。

我们假设 $I-set$ 和 $II-set$ 分别是 PG 中 I $type$ 和 II $type$ 语句的集合。令 $I-set=\{s_{i_1}, s_{i_2}, ..., s_{i_x}\}$，其中包含 x 条语句，$II-set=\{s_{j_1}, s_{j_2}, ..., s_{j_{n-x}}\}$，其中包含 $n-x$ 条语句。

向量表模型（Vector table model）的定义

表3-1显示了 PG 中两种类型语句的 $< a_{nf}, a_{np}, a_{ef}, a_{ep} >$ 的值。该表是我们的理论模型，称为向量表模型（简称 VTM）。对于 I $type$ 语句而言，它在每个测试用例中都执行过，因此其 a_{np} 和 a_{nf} 为零。所以 I set 中语句的 $< a_{nf}, a_{np}, a_{ep}, a_{ef} >$ 向量为 $< 0, 0, t-p, p >$，其中 t 表示测试用例总数，p 表示成功的（通过的）测试用例总数。对于 II $type$ 语句而言，因为有一个或多个测试用例不覆盖它（即，它不在某些测试用例中执行），它的 $a_{np}+a_{nf}$ 一定会大于 0。我们定义 II set 中 II $type$ 语句的 $< a_{nf}, a_{np}, a_{ef}, a_{ep} >$ 为 $< m_i, k_i, t-p-m_i, p-k_i >$，且其中的变量满足以下的条件：

1. m_i, k_i, $t-p-m_i$, $p-k_i$ 是整数，并且其值大于等于 0。
2. 如果 $m_i=0$，那么 $k_i>0$.
3. 如果 $k_i=0$，那么 $m_i>0$.

表 3-1　向量表模型

type	stat	a_{nf}	a_{np}	a_{ef}	a_{ep}
I type	s_{i_1}	0	0	$t-p$	p
	s_{i_2}				
	s_{i_3}				
	...				
	s_{i_x}				
II type	s_{j_1}	m_1	k_1	$t-p-m_1$	$p-k_1$
	s_{j_2}	m_2	k_2	$t-p-m_2$	$p-k_2$
	s_{j_3}	m_3	k_3	$t-p-m_3$	$p-k_3$
	...				
	$s_{j_{n-x}}$	m_{n-x}	k_{n-x}	$t-p-m_{n-x}$	$p-k_{n-x}$

VTM 是任意实际程序的理论模型，它使用变量而不是实际程序的具体值来表示每条语句的向量 $< a_{nf}, a_{np}, a_{ef}, a_{ep} >$。我们可以根据给定测试套件

表 3-2 图2-1中 PG 的向量表模型实例

type	stat	a_{nf}	a_{np}	a_{ef}	a_{ep}
I type	s_1	0	0	3	6
	s_4				
	s_7				
	s_9				
II type	s_5	0	3	3	3
	s_6	0	3	3	3
	s_{10}	0	3	3	3
	s_3	0	4	3	2
	s_2	1	0	2	6
	s_8	2	0	1	6

的两种类型的语句对实际程序的语句进行分类，并且可以构造其向量表模型的实例。分类是根据测试套件执行的语句的频率特性（频谱）实现的。例如，图2-1中的 PG 的 VTM 如表3-2所示。

3.3 基于平均排名代价的测试套件质量评估

根据上文的平均排名代价及向量表的定义，我们可以使用如下的步骤来计算任意程序的平均排名代价。

1. 运行 PG 的测试套件 TS，导出 PG 的语句覆盖矩阵 M 和结果向量 OC。
2. 用 M 和 OC 计算 PG 中每个语句的向量 $< a_{nf}, a_{np}, a_{ef}, a_{ep} >$。然后构建在3.2节中提到的向量表模型。
3. 根据向量表模型，使用对应的 SFL 计算程序 PG 中语句的可疑因子。
4. 根据向量表模型的特征，找出其中 $a_{nf} \neq 0$ 的语句，这些语句在单故障程序中不可能是故障语句。
5. 对 SF 按照其可疑因子从大到小进行排序，得到排名。
6. 使用公式 2.1.3 计算其 $ExamScore$，随后从中移除 $a_{nf} \neq 0$ 的语句计算出的值。
7. 将上一步中剩下的数值取平均值。即可得到平均排名代价。

使用此平均排名代价计算方式，我们可以获取任意程序在任意公式上的平均排名代价。图3-1展示了平均排名代价的计算流程及后续步骤，其中的最后一步将在后文第4小节中说明。

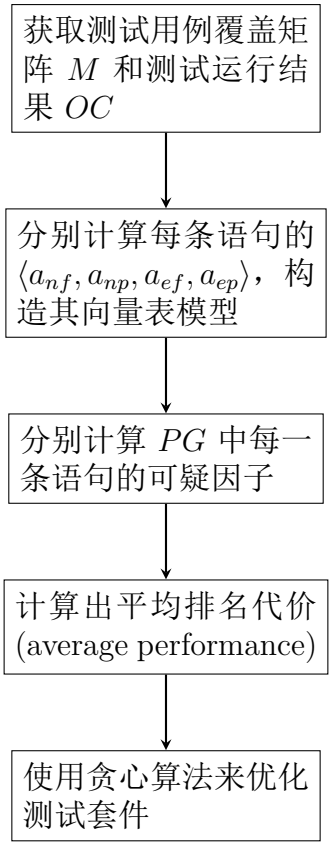

图 3-1　计算框架的大致流程

3.4 案例分析

我们使用图2-1中的 SFL 作为例子。对于应用了 O 公式得到的 SF，我们对其从大到小排序，结果是 $\langle[s_3],[s_5,s_6,s_{10}],[s_1,s_4,s_7,s_9],[s_2,s_8]\rangle$，语句按照可疑因子从高到低的顺序排列，每对 [] 标记的语句具有相同的排名。

按照公式2.1.3计算 Exam Score，n 取 10，对于排名第一的语句 s_3，其 g 等于 1，由于没有和它排名相同的语句，因此它的 e 等于 0，因此它的平均代价为 $\frac{1}{10}$ 。对于排名第二的语句 s_5, s_6, s_{10}，其 g 等于 2，排名为 2 的语句共有 3 个，因此 e 为 3；平均代价 ap 为 $\frac{2+\frac{3}{2}}{10} = \frac{7}{20}$。依次类推，我们可以得到每条语句的平均代价，即 $s_1 = \frac{7}{10}$, $s_2 = 1$, $s_3 = \frac{1}{10}$, $s_4 = \frac{7}{10}$, $s_5 = \frac{7}{20}$, $s_6 = \frac{7}{20}$, $s_7 = \frac{7}{10}$, $s_8 = 1$, $s_9 = \frac{7}{10}$, $s_{10} = \frac{7}{20}$。

去除 $a_{nf} \neq 0$ 的语句 s_2, s_8，将剩下的结果取平均值，得到 $\frac{7}{10} + \frac{1}{10} + \frac{7}{10} + \frac{7}{20} + \frac{7}{20} + \frac{7}{10} + \frac{7}{10} + \frac{7}{20} = 3.95$。即为其平均代价。

第 四 章　基于贪心算法的测试套件优化方法

在本章中，我们将介绍一个基于贪心算法的测试套件优化方法，其可以在 $O(n)$ 的时间复杂度之内通过削减测试用例计算出一个测试套件子集，使其平均排名代价小于或等于原先的测试套件的平均排名代价。

4.1 启发式原则

本算法基于以下的启发式原则而设立。

1. 我们假定平均排名代价同测试套件质量存在负相关的关系，即，平均排名代价越低，测试套件质量越高。我们将在后文中通过实验来验证这一点。

2. 同时移除两个使得平均排名代价升高的测试用例，平均排名代价在统计意义上有更大的可能性会降低。也就是说，虽然理论上可能存在以下情况：单独移除测试用例 t_1 和 t_2 时都能使平均排名代价降低，但是同时移除它们会使平均排名代价升高。但是我们假设发生这种情况的概率较低。

因此，本文中提到的计算框架通过贪心算法来获得一个“较优”的测试套件子集。其计算过程将在后文章节描述。

4.2 基于贪心算法的测试套件优化算法

下文中将首先描述本优化方式的核心算法，随后描述此核心算法当中使用到的子过程。

本文核心算法为算法1，其1到2行中首先保留了全部的测试用例并计算其平均代价。接下来对于每一个测试用例（第3行），首先移除这个测试用例（第4行），计算总体的平均代价（第5行），假如平均代价更低或不变，则保持移除这个测试用例（第7行）并转到下一个测试用例；否则，将这个测试用例重新加入到测试用例子集中（第9行）。

算法2用来计算一个测试用例子集的平均代价。将首先计算出每条语句的 $ExamScore$（4到16行），然后从中移除 $a_{nf} \neq 0$ 的值（第20行），将剩下的取平均值。

算法3用于计算某一个排名上所有语句的 exam score，并且填充 $ExamScore$ 向量。由于可能有多条语句的可疑因子取相同的值，因此它

算法 1: 求解较优测试套件子集

```
Input: M[n_t][n_s] ▷ 测试用例的覆盖矩阵
Input: OC[n_t] ▷ 测试用例的运行结果向量
Input: n_t ▷ 总的测试用例数量
Input: n_s ▷ 语句数量
Output: N[n_t] ▷ 保留的测试用例的向量
1  N ← [1, 1, 1, ..., 1]
2  best ← resolveAveragePerformance(M, OC, N)
3  for i ← 0 to n_t − 1 do
4  |   N[i] ← 0
5  |   ap ← resolveAveragePerformance(M,N,OC)
6  |   if ap ≤ best then
7  |   |   best ← ap
8  |   else
9  |   |   N[i] ← 1
10 |   end
11 end
12 return N
```

们的排名应该一样。在算法的第4到8行，用来判断是否还有其他的和本语句排名相同的语句，如果没有，e 取 0，否则 e 取这类语句的总体数量（包含本语句）。

算法4用于根据 VTM 计算可疑因子向量。算法中的 f 根据实际情况可以替换成任意可疑因子计算公式，比如 O_p 公式为 $\frac{line_i.a_{ef}-line_i.a_{ep}}{line_i.a_{ep}+line_i.a_{np}+1}$。

算法5用于根据覆盖矩阵和运行结果计算向量表模型。传入的 $N[n_t]$ 表示是否保留某个测试用例（$N[i]$ 为 1 时表示保留 i 号测试用例，为 0 时不保留）。第1行首先将向量表初始化成每一行都是 0。然后在接下来的循环中，首先判断是否保留该测试用例（第3行），然后根据向量表定义来更新向量表（6到18行）。

算法 2: resolveAveragePerformance

Input: n_s ▷ 语句数量
Input: n_t ▷ 测试用例数量
Input: $M[n_t][n_s]$ ▷ 测试用例的覆盖矩阵
Input: $OC[n_t]$ ▷ 测试用例的运行结果向量
Input: $N[n_t]$ ▷ 保留的测试用例的向量
Output: ap ▷ 计算出的可疑因子

$VTM \leftarrow$ resolveVectorTableModel(M, OC, N)
$SF \leftarrow$ resolveSuspiciousnessFactor(VTM)
sort SF by $suspiciousnessFactor$ in descending order
$ExamScore[n_s] \leftarrow [0, 0, ..., 0]$
$rankBeginIndex \leftarrow 0$ ▷ 当前元素的开始位置（可疑因子一样的元素）
$rankEndNextIdx \leftarrow 0$ ▷ 当前元素的截止位置的下一个元素的位置
while $rankEndNextIdx < n_s$ **do**
    if $SF[rankBeginIndex].suspiciousnessFactor = SF[rankEndNextIdx].suspiciousnessFactor$ **then**
        $rankEndNextIdx \leftarrow rankEndNextIdx + 1$
    else
        fillExamScoreArray$(SF, rankBeginIndex, rankEndIndex, ExamScore)$
        $rankBeginIndex \leftarrow rankEndNextIdx$
        $rankEndNextIdx \leftarrow rankEndNextIdx + 1$
    end
end
fillExamScoreArray$(SF, rankBeginIndex, rankEndIndex, ExamScore)$ ▷ 填充最后一个排名的位置
$sum \leftarrow 0$
$count \leftarrow 0$
for ***each*** $line_i$ ***in*** VTM **do**
    if $line_i.a_{nf} = 0$ **then**
        $sum \leftarrow sum + ExamScore[i]$
        $count \leftarrow count + 1$
    else
    end
end
return $\frac{sum}{count}$

算法 3: fillExamScoreArray

Input: $SF[n_s]$ ▷ 每条语句的可疑因子（排序过的）
Input: n_s ▷ 语句数量
Input: $rankBeginIndex$ ▷ 相同排名的语句开始位置的下标
Input: $rankEndIndex$ ▷ 相同排名的语句结束位置的下标
Input: $ExamScore[n_s]$ ▷ 输出用的向量（可能已经部分填充过）
Output: $ExamScore[n_s]$

$n \leftarrow n_s$
$g \leftarrow rankBeginIndex$
$thisRankCount \leftarrow rankEndNextIdx - rankBeginIndex$
if $thisRankCount > 1$ **then**
  $e \leftarrow thisRankCount$
else
  $e \leftarrow 0$
end
$examScore \leftarrow \frac{g+\frac{e}{2}}{n}$
for $i \leftarrow rankBeginIndex$ **to** $rankEndNextIdx - 1$ **do**
  $ExamScore[SF[i].statementIndex] \leftarrow examScore$
end
return $ExamScore$

算法 4: resolveSuspiciousnessFactor

Input: $VTM[n_s]$ ▷ VectorTableModel
Output: $SF[n_s]$ ▷ 可疑因子

for ***each*** $line_i$ ***in*** VTM **do**
  $SF[i] \leftarrow \{statementIndex \leftarrow i, f(line_i)\}$
end
return SF

算法 5: resolveVectorTableModel

Input: n_s ▷ 语句数量
Input: n_t ▷ 测试用例数量
Input: $M[n_t][n_s]$ ▷ 测试用例的覆盖矩阵
Input: $OC[n_t]$ ▷ 测试用例的运行结果向量
Input: $N[n_t]$ ▷ 保留的测试用例的向量
Output: $VTM[n_s]$ ▷ 得到的 VectorTableModel

```
1  VTM ← [line_0, line_1, line_2, ..., line_{n_s−1}] where
     line_i ← {a_ef ← 0, a_ep ← 0, a_nf ← 0, a_np ← 0}
2  for i ← 0 to n_t − 1 do
3      if N[i] = 1 then
4          for j ← 0 to n_s − 1 do
5              hit ← M[i][j]
6              if OC[i] then
7                  if hit then
8                      VTM[j].a_ep ← VTM[j].a_ep + 1
9                  else
10                     VTM[j].a_np ← VTM[j].a_np + 1
11                 end
12             else
13                 if hit then
14                     VTM[j].a_ef ← VTM[j].a_ep + 1
15                 else
16                     VTM[j].a_nf ← VTM[j].a_np + 1
17                 end
18             end
19         end
20     else
21     end
22 end
23 return VTM
```

第 五 章　辅助工具设计与开发

5.1 辅助工具需求分析

想要完成实验，本文中的计算程序需要实现以下功能。

1. 能够运行被测程序，分别收集其在每个测试用例中的运行结果（正确、错误）和语句覆盖信息。
2. 能够从运行结果和覆盖信息生成 VTM 矩阵、可疑因子向量。
3. 能够根据可疑因子向量生成语句的排名，并分析得出的结果。
4. 能够根据向量和矩阵来优化测试套件（移除测试冗余的测试用例，选择测试用例子集，从而使得总体定位性能更好）。
5. 可以将任意的可疑因子计算公式应用到上述的过程中。
6. 以上功能可以任意组合，以达到随意定制计算过程的效果。
7. 可以对以上任意功能的计算结果进行分析，包括但不限于生成 Excel 表格，生成统计图表。
8. 由于部分内容需要较多时间进行计算，本工具必须能够保存运行结果，并且能够使用之前保存的运行结果直接进行之后的计算。

另外，为了代码的可读性和可接受性，本文使用的辅助工具全部使用 Java 编写。

5.2 辅助工具功能结构图

根据上面的需求分析，本辅助工具的功能结构图如图5-1所示。

5.3 辅助工具设计方案

为了最大程度的保持和论文的一致性，本计算程序使用了面向数据的编程方式，因此整个程序在大致上可以分为两部分。

1. 数据部分。本部分为纯数据，在 Java 代码中大多都是 POJO，存放在各个包的 pojo 子包中。

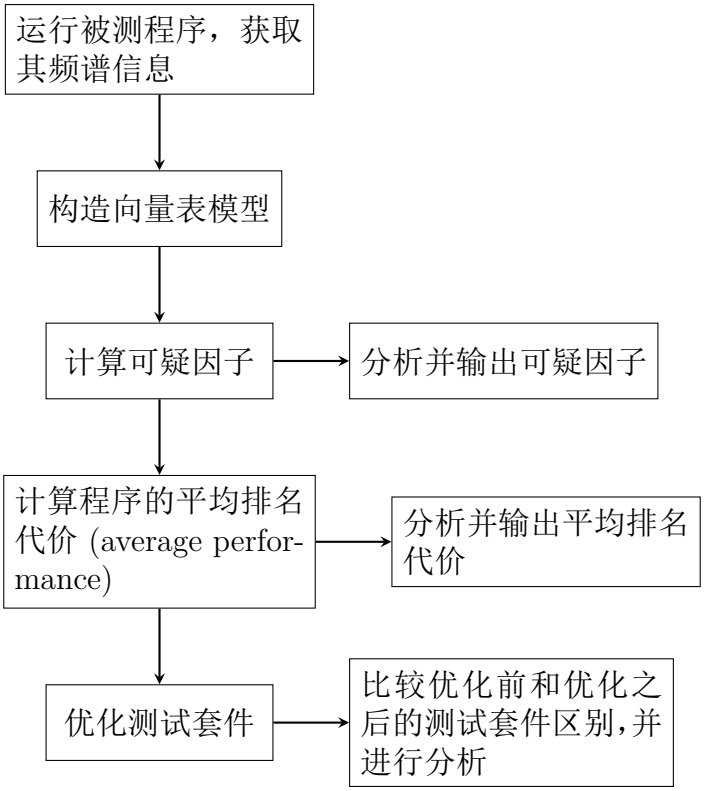

图 5-1　功能结构图

2. 计算部分。本部分的大部分代码都为静态方法。可以直接通过 *类名. 方法名*的方式来调用。每个方法都负责将一种数据转换成另一种数据。对于部分需要参数的方法（比如计算可疑因子向量需要使用的可疑因子计算公式），方法则为普通的对象方法，参数通过类的构造函数注入，部分拥有可选参数的计算，将通过工厂类来产生用于调用计算方法的对象。这一部分代码存储在各个包的根目录下。

此外，计算程序的大致分包如下。

1. runner。用于运行被测程序的代码。
2. analyze。计算程序的核心，用于计算 VTM、可疑因子、子集等数据。
3. chart。用于绘制图表的类。
4. utils。辅助类，用于提供部分公共函数。比如对象存储和读取、缓存处理、日志处理等等。
5. application。其中每个类都具有 main 函数，是某个功能的入口点，可以

单独运行。一般情况下，某个 main 函数就是多个对于 runner、analyze、chart 包中的方法的调用，再加上文件的输入和输出。

5.4 实验环境配置及搭建

本辅助工具使用 Java 编写，由于使用了 Java 8 中的部分功能，因此本辅助工具需要在 Jdk 1.8 及以上的环境中运行。同时，本工具使用了 maven 作为其包管理工具，因此实验设备上还需要安装 maven；剩余的 Java 依赖库可以使用 maven 自动安装。如果需要运行 Defects4j 数据集，还需要实验设备中安装有 Docker，因为此数据集只支持 JDK 1.7 及以下的运行环境，与本项目运行环境不兼容，需要使用 docker 来隔离环境。

在配置 docker 时，如果使用的是 windows 平台上的 docker-desktop，需要按照图 5-2所示，开启端口访问功能。

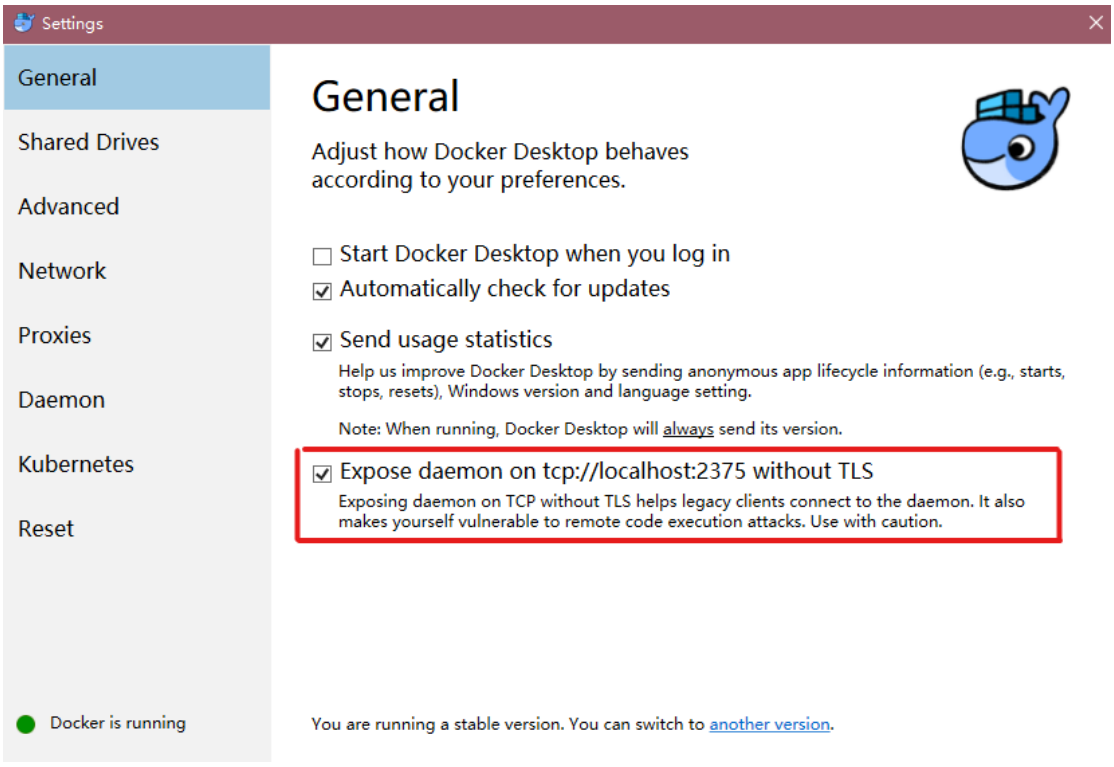

图 5-2　Docker 中需要开启的设置

5.5 辅助工具实现

5.5.1 使用技术

为了克服 Java 语法冗长、样板代码多的问题以及减少编写辅助工具时的错误率，我们使用了以下的工具和技术来编写计算程序的代码。

1. 使用 Lombok 来生成样板代码。本程序使用了 Lombok 来自动生成 POJO 的 get、set 方法、构造函数、equals 等方法。在某些情况下，也使用它来生成工厂类，从而简化代码，提高程序可读性。

2. 使用 Java8 提供的 Stream 来处理序列。通过使用 Java8 Stream 而不是普通循环，本程序可以用声明式的方法来编写程序，让代码着重于描写“应该做什么”而不是“应该怎么做”，可以显著提高代码的可读性，减少样板代码并减少出错的可能性。同时，可以利用其提供的并发编程功能显著提高计算速度。

3. 使用 Stream 的辅助类库 StreamEx 来处理 Stream。这样可以进一步减少由 Java8 Stream 而引入的样板代码。

4. 尽量使用不变量。除了 runner 中的一部分代码和工厂类以外，本程序的其他代码均以常量、不可变对象和不可变列表作为输入和输出，这样可以有效减少在并发计算中由共享状态而导致的 bug。

除此之外，我们还使用了 JFreeChart 作为绘图工具，使用 Gson 来存储和读取中间结果。

5.5.2 分包

本文使用的计算程序由 Java 实现，其分包如下所示。

1. runner。用于运行被测程序的代码。

   (a) 根目录存储被测程序的运行框架。使用面向对象的方式来编写，只要给不同类型的程序（C 语言、Java）编写相应的实现，就可以测试任意的程序。

   (b) data。被测程序的输入输出及覆盖信息的对象表示。

   (c) impl。运行具体类型的程序的代码实现。

(d) pojo。将被测程序的输入和输出转换成计算程序需要的数据之后，数据的对象表示。

2. analyze。计算程序的核心，用于计算 VTM、可疑因子、子集等数据。

   (a) 根目录存储用于运算的类。

   (b) pojo。存储运算的输入和输出数据的定义。某些方法的输入可能是其他方法的输出，或者 *runner.pojo* 中的对象。

3. chart。用于绘制图表的类。

   (a) 根目录存储用于绘图的类，均为静态类。其中每个方法的输入都为 *analyze.pojo* 或者 *chart.pojo* 中的对象。

   (b) pojo。存储部分输入数据的定义。

   (c) renderer。对于部分自行定制的图表类型，需要自己编写 renderer 来绘制图表，本包存储了这些实现。

4. utils。辅助类，用于提供部分公共函数。比如对象存储和读取、缓存处理、日志处理等等。

5. application。其中每个类都具有 main 函数，是某个功能的入口点，可以单独运行。一般情况下，某个 main 函数就是多个对于 runner、analyze、chart 包中的方法的调用，再加上文件的输入和输出。

   (a) temporary。和根目录中存储的代码类型相同，不过大部分都是一次性的代码，用于临时检查输出结果等等。

   (b) utils。存储一些 application 包中使用的公共方法。

5.5.3 计算过程

本程序的计算过程如图5-3所示。由于 runner 部分用于运行程序，必须使用变量和可变对象的方式来存储数据，因此我们引入了类 *RunningResultResolver* 来作为 runner 包和 analyze 包之间沟通的桥梁。它输入一个可变的 *RunResultFromRunner*，输出一个不可变的 *RunResult*，便于后续的计算。

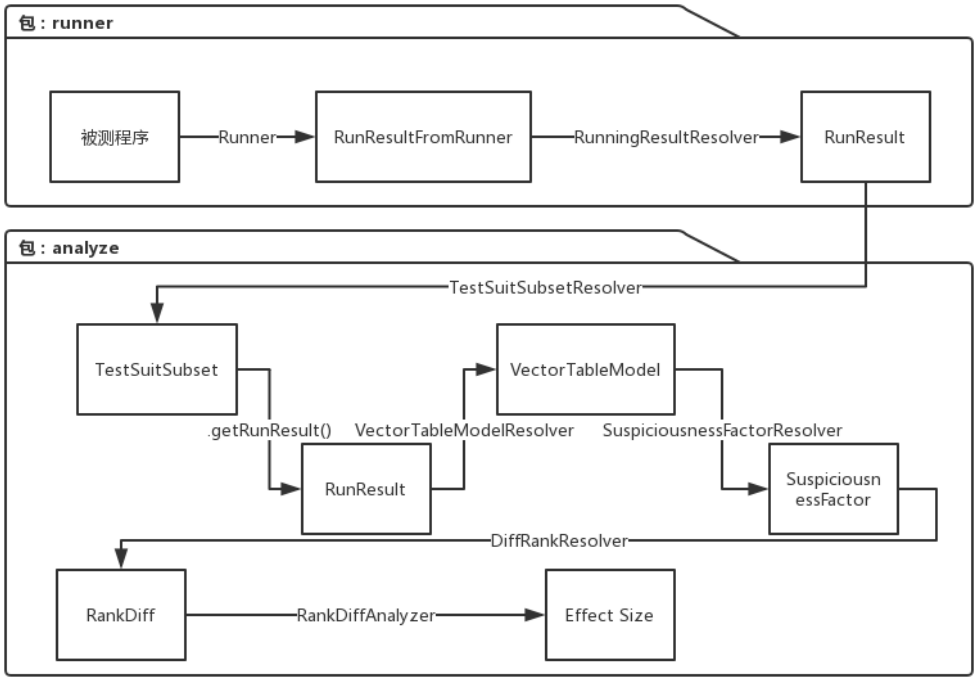

图 5-3　计算过程及分包

5.5.4 类的具体功能

本程序中的所有类都使用了基于约定的命名方式，其约定如下。

每个实际进行计算的类，其类名都以 Resolver 结尾，比如 $VectorTableModelResolver$，表示用于计算 VTM 的类。标有 Helper 或者 Utils 的类保存了辅助 Resolver 进行计算的代码。比如 $SuspiciousnessFactorUtils$ 中的静态方法，用于将同一个输入应用到多个 $SuspiciousnessFactorResolver$ 中，并将结果收集到一起。

每个 POJO 类的后缀都具有规律，比如 $VectorTableModelForStatement$ 保存了一条语句的向量 $\langle a_{nf}, a_{np}, a_{ef}, a_{ep} \rangle$，而 $VectorTableModelForProgram$ 中则保存了一个程序中的所有语句的向量、$VectorTableModelJam$ 则保存了所有的程序（通常是一个项目中的所有版本）的集合。

类的具体功能如下所示。按照包来进行分类。

1. analyze 包。计算程序的核心，用于计算 VTM、可疑因子、子集等数据。其中的类由表 5-1所示。

表 5-1 analyze 包结构图

AveragePerformanceResolver	从 VectorTableModelForProgram 中生成测试用例的平均代价
DiffRankFilters	DiffRank 的筛选器，用来筛选 DiffRankResolver 获得的结果
DiffRankResolver	比较两个 Suspiciousness Factor，输出结果
RankDiffAnalyzer	分析两个排名的相关程度，用来分析结果好坏
SuspiciousnessFactorFormulas	保存可疑因子的计算公式
SuspiciousnessFactorResolver	统计一次运行中所有语句的可疑因子，忽略从来没有运行过的语句 (aep+aef==0)
SuspiciousnessFactorUtils	
TestcaseWeightHelper	
TestcaseWeightMultiplyingResolver	将测试用例的权重扩大，使结果更加明显。
TestcaseWeightResolver	生成测试用例的权重
TestcaseWeightResolver	Builder
TestSuitSubsetResolver	从一系列测试用例中获得一个较优的测试用例子集
VectorTableModelResolver	生成 VectorTableModelForProgram

2. analyze.pojo 包。存储运算的输入和输出数据的定义。其中的类由表 5-2 所示。

3. runner 包。用于运行被测程序的代码。其中的类由表 5-3 所示。

4. runner.data 包。被测程序的输入输出及覆盖信息的对象表示。其中的类由表 5-4 所示。

5. runner.impl 包。运行具体类型的程序的代码实现。其中的类由表 5-5 所示。

6. runner.pojo 包。将被测程序的输入和输出转换成计算程序需要的数据之后，数据的对象表示。其中的类由表 5-6 所示。

7. chart 包。用于存储绘制图表的类。其中的类由表 5-7 所示

8. application 包。其中每个类都具有 main 函数，是某个功能的入口点，可以单独运行。其中的类由表 5-8 所示

5.5.5 辅助工具类图

图 5-4 列出了项目总体的包依赖关系，其中可以看出，application 包是可以调用任何其他的包，utils 包可以被任何的其他包调用。除此之外，剩下的三个包 runner、analyze 和 chart 都不能调用其他的包，这样可以保证各个类之间的解耦，便于维护代码。

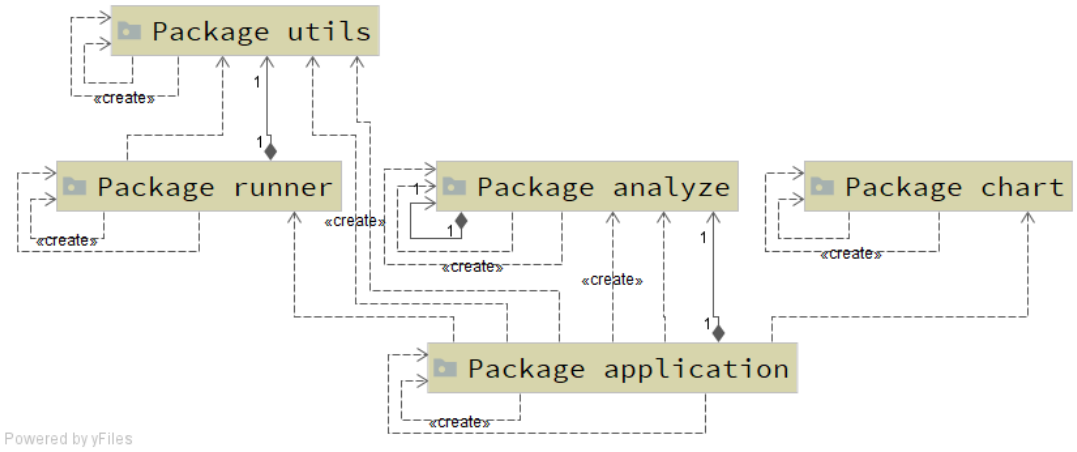

图 5-4　项目总体的包依赖关系

由于辅助工具使用的是流程式的处理方法，各个类之间不存在继承关系，这里只贴出单个 Resolver 类对于其他类的调用关系的类图。其中，左侧的类会调用（使用）中间的类，中间的类会调用右侧的类。

RunningResultResolver 的类图如图 5-5 所示。

VectorTableModelResolver 的类图如图 5-6 所示。

SuspiciousnessFactorResolver 的类图如图 5-7 所示。

AveragePerformanceResolver 的类图如图 5-8 所示。

DiffRankResolver 的类图如图 5-9 所示。

图5-10中列出了 chart 包中的类图。

图5-11中列出了 runner 包中的类图。runner 包中执行具体类型的被测程序的类使用到了接口，这里列出接口和相应的实现类之间的依赖关系。

5.6 辅助工具使用说明

我们可以通过 maven 来启动试验程序，也可以通过 IDE 的运行功能直接启动实验程序，因此在下文中，我将其分成两部分分别解释。

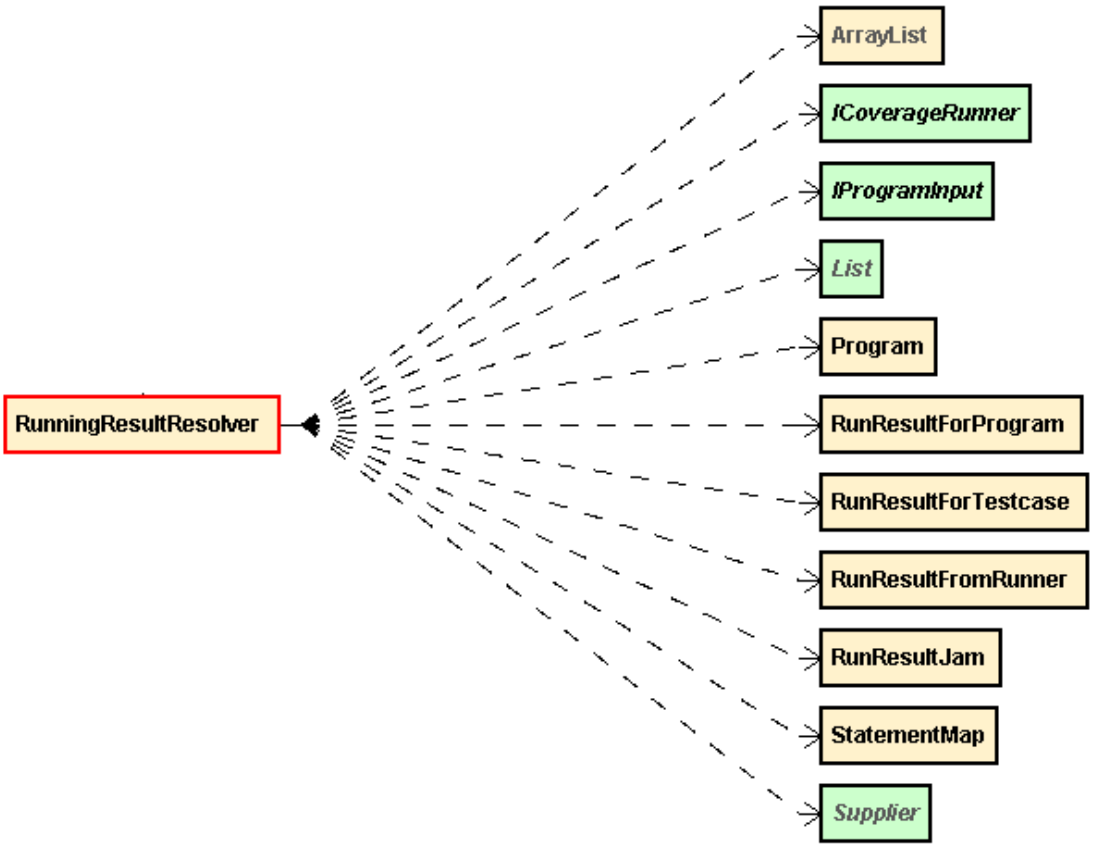

图 5-5　RunningResultResolver 的类图

1. 通过 IDE 启动试验程序。这里以 Intellij IDEA 为例。实际上，其他的 IDE 的操作方式大致相同。我们首先需要在 IDE 中打开项目，然后直接在类列表中右键单击 application 包下的任何一个类，选择“Run 类名”即可直接运行。如图 5-12 所示。

2. 通过命令行启动试验程序。使用者可以直接在项目根目录下使用命令‘run-local.cmd 类名’的方式来运行实验程序。比如‘run-local.cmd Main’将会运行 *application.Main* 类。

在运行实验程序时，某个类可能会依赖上一个类的输出，比如 *ResolveTestSuitSubset* 类会依赖 *Run* 类。在运行过程中，如果被依赖的类没有事先运行过，实验程序将会报错，并提示运行对应的类。

实验程序的运行结果将被储存进实验程序根目录下的 *target/outputs* 目录中，依赖其他实验结果的类也会从这个目录下读取之前的输出结果。除此之外，在 *target/outputs/.cache* 目录中储存了运算时的中间结果，使得实验程序可以在任何时候暂停或重启。

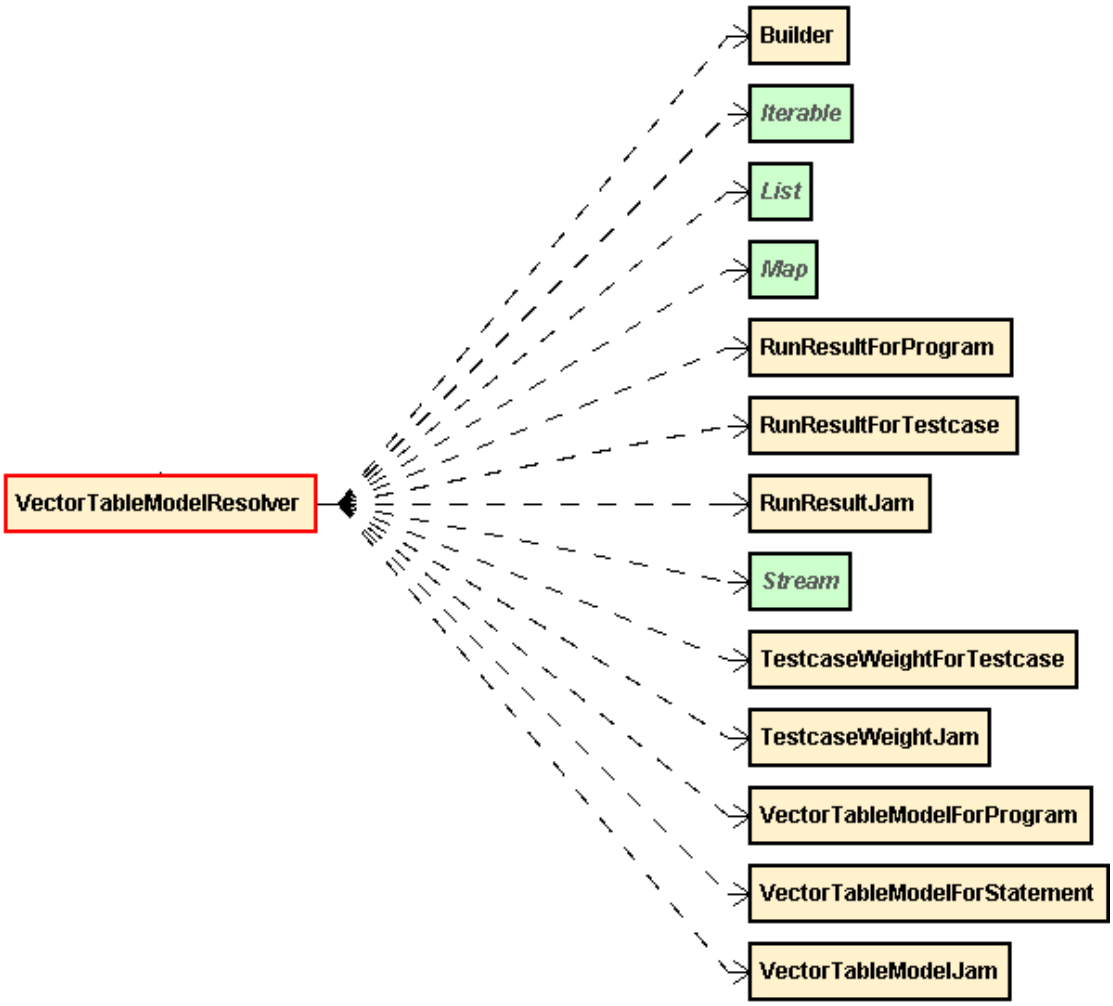

图 5-6　VectorTableModelResolver 的类图

application.Main 类包含了执行实验所需要的其他程序的调用，通过执行这一个类，可以按照依赖顺序执行实验中的所有计算，不需要一个个手动启动计算。

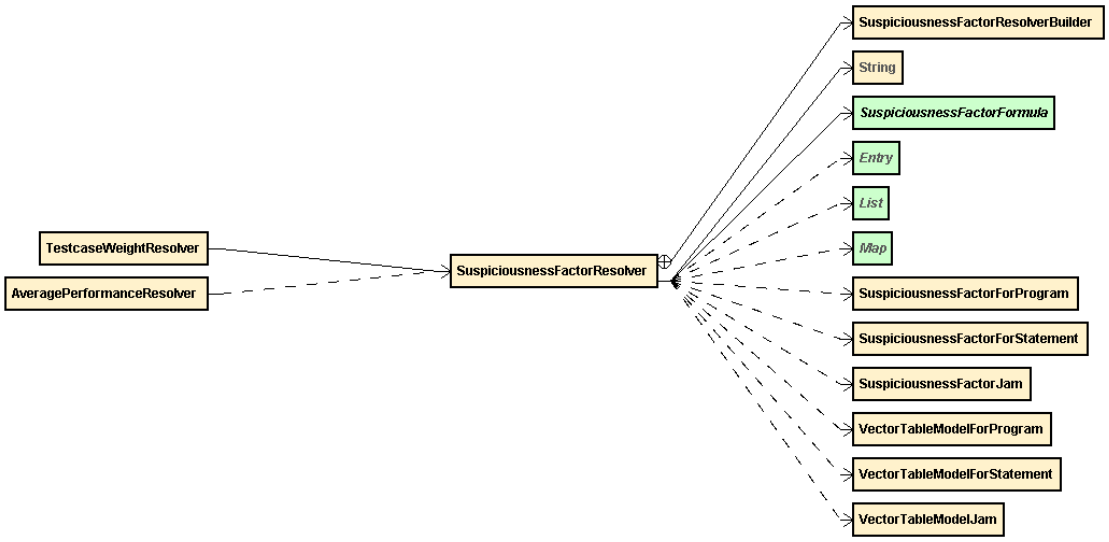

图 5-7　SuspiciousnessFactorResolver 的类图

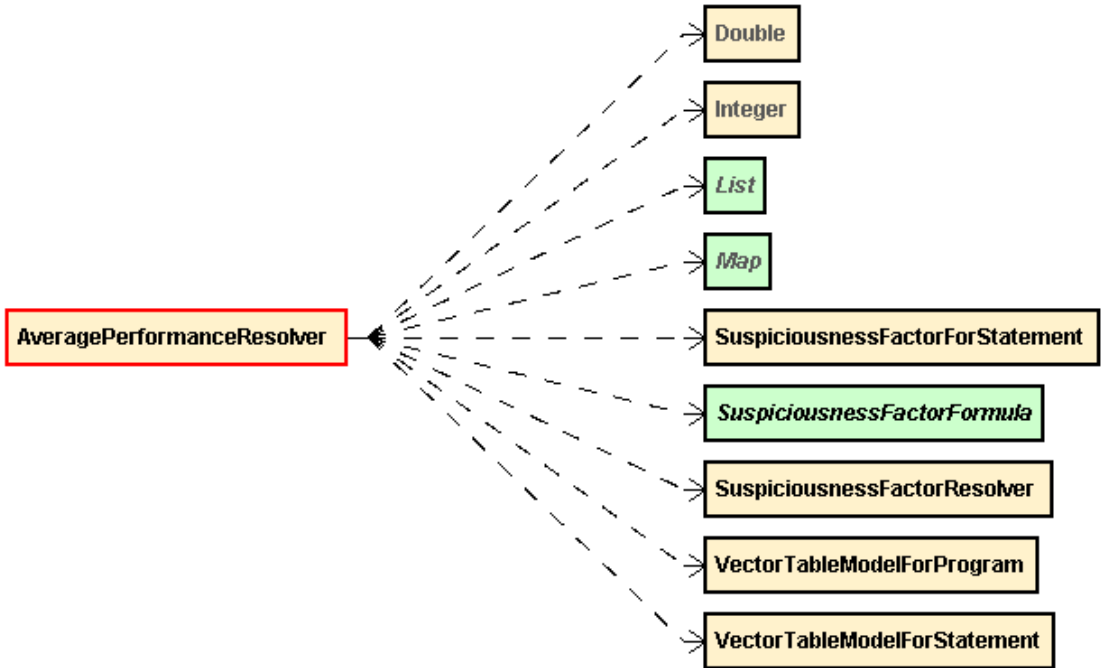

图 5-8　AveragePerformanceResolver 的类图

表 5-2 analyze.pojo 包结构图

DiffRankForProgram	一个程序中所有的加权比较
DiffRankForSide	一侧的排名信息
DiffRankForStatement	表示一条语句在两种计算方式下排名的区别
DiffRankJam	
FaultLocationForProgram	一个程序中所有可能的错误位置，便于分析计算结果用
FaultLocationJam	
MultipleFormulaSuspiciousnessFactorForProgram	
MultipleFormulaSuspiciousnessFactorForStatement	一条语句中有多个公式的结果
MultipleFormulaSuspiciousnessFactorJam	
SuspiciousnessFactorForProgram	程序中的所有可疑因子
SuspiciousnessFactorForStatement	
SuspiciousnessFactorJam	
TestcaseWeightForProgram	表示一个程序的 title 和测试用例
TestcaseWeightForTestcase	一个测试用例的权重
TestcaseWeightJam	一个程序的测试用例权重所需的所有数据
TestSuitSubsetForProgram	由 TestSuitSubsetResolver 处理过的结果。
TestSuitSubsetJam	
VectorTableModelForProgram	一个 vtm
VectorTableModelForStatement	表示 vector table model 中的一行
VectorTableModelForStatement.Builder	用于生成 VectorTableModelForStatement
VectorTableModelJam	

表 5-3 runner 包结构图

ICoverageRunner	通过特定的分析器运行程序，得出覆盖信息的接口
IProgramInput	一次运行的输入。
RunningResultResolver	运行特定组织结构的程序的工具
RunningScheduler	用于在一个程序上运行所有的测试用例，输出结果和覆盖信息。
TestcaseResolver	获取一个程序的所有测试用例
CoverageRunnerException	

表 5-4 runner.data 包结构图

Coverage	表示一次运行的语句覆盖信息
Program	一个源程序代码，用于储存用来分析的程序的元信息
RunResultFromRunner	程序的单次运行结果，采用了适合 runner 的抽象
StatementInfo	一条语句的信息
StatementMap	表示一个程序源代码的语句信息，包含了文件中每条语句的位置和语句编号。

表 5-5 runner.impl 包结构图

Defects4jContainerExecutor	持有一个 docker container 实例，用来运行 defects4j 有关的命令。
Defects4jContainerExecutor	CoverageRunResult：一个测试用例的运行结果
Defects4jCoverageParser	读取 Defects4j 的覆盖文件，获取覆盖信息。
Defects4jCoverageParser	Defects4jStatementMap：处理 Defects4j 的语句编号
Defects4jRunner	用于运行 Defects4j 的测试程序
Defects4jTestcase	定义了 Defects4j 测试程序的一个输入
GccReadFromStdIoInput	定义了 C 语言测试程序的一个输入。
GccReadFromStdIoRunner	使用 gcov 执行 c++ 源代码的覆盖率检测。
GcovParser	从 gcov 文件生成覆盖信息，基于行。

表 5-6 runner.pojo 包结构图

RunResultForProgram	一个程序的运行结果
RunResultForTestcase	表示一次运行的结果，不包含多余的数据
RunResultJam	

表 5-7 chart 包结构图

DiffRankChart	生成用于比较排名的图表。
EffectSizeChart	生成 Effect Size 的图表

表 5-8 application 包结构图

DiffSfSubsetByCertainFormula	最简化版本的比较代码，只比较使用特定公式划分测试用例子集、并且使用该公式计算结果的代码。
DiffSfWeightedByCertainFormula	最简化版本的比较代码，只比较使用特定公式加权、并且使用该公式计算结果的代码。
FindNotOneTestcaseWeight	在测试用例权重列表中筛选出来权重不是 1.0 的权重列表
Main	按顺序运行多个程序
ResolveAllEffectSize	计算所有程序，所有公式的 effect size
ResolveDefects4jFaultLocations	从 https:// github.com/ program-repair/ defects4j-dissection 里提取出我需要的数据
ResolveDefects4jTestcase	获取 defects4j 中的所有测试用例
ResolveSortedTestcaseWeight	获取根据权重排名过的测试用例，便于观察结果
ResolveSuspiciousnessFactor	获取可疑因子
ResolveTestSuitSubset	获取划分之后的测试用例子集
ResolveTestSuitSubsetByOp	获取划分之后的测试用例子集，使用 Op 来划分
Run	批量执行测试程序，为每个程序单独返回结果

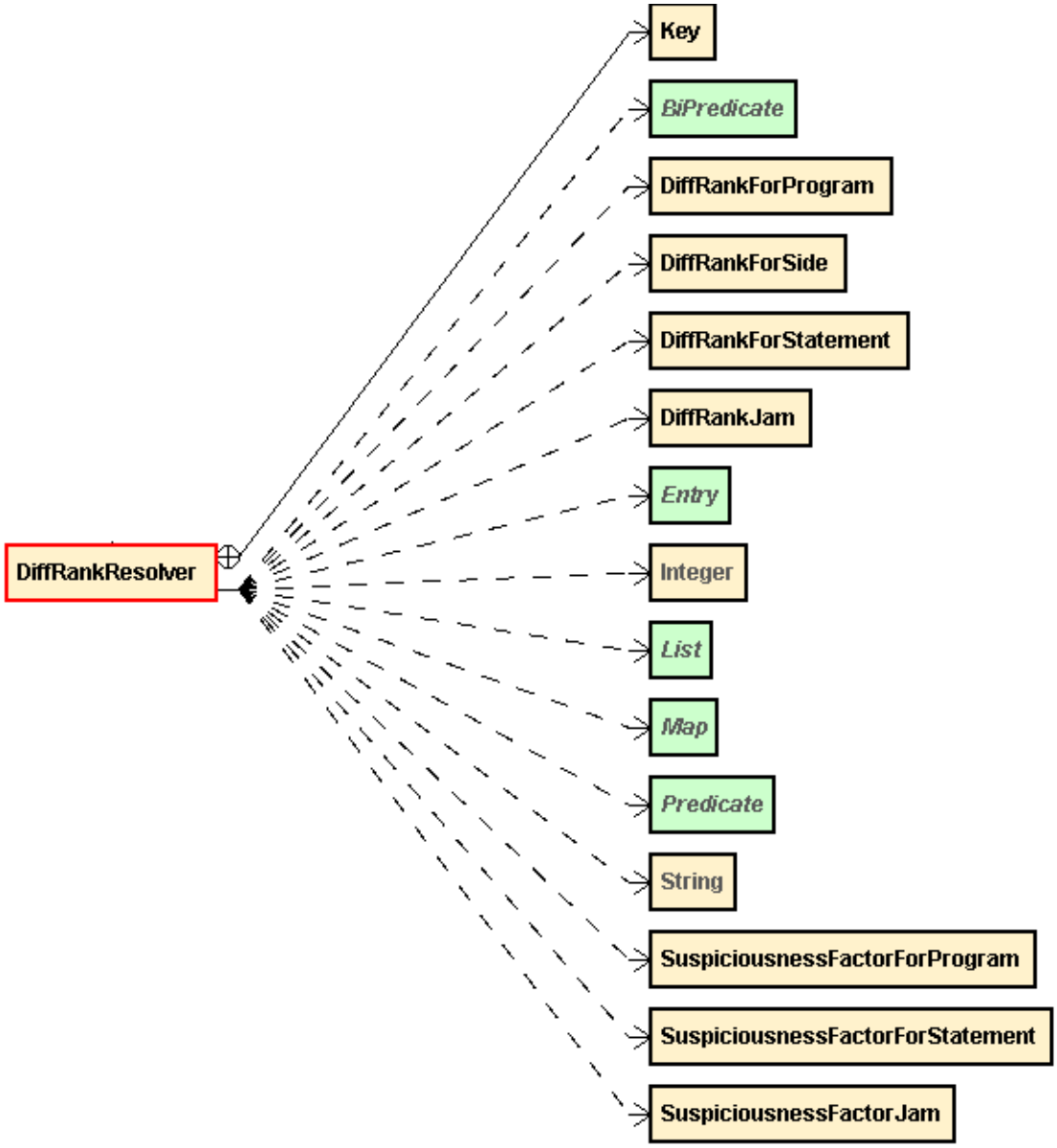

图 5-9　DiffRankResolver 的类图

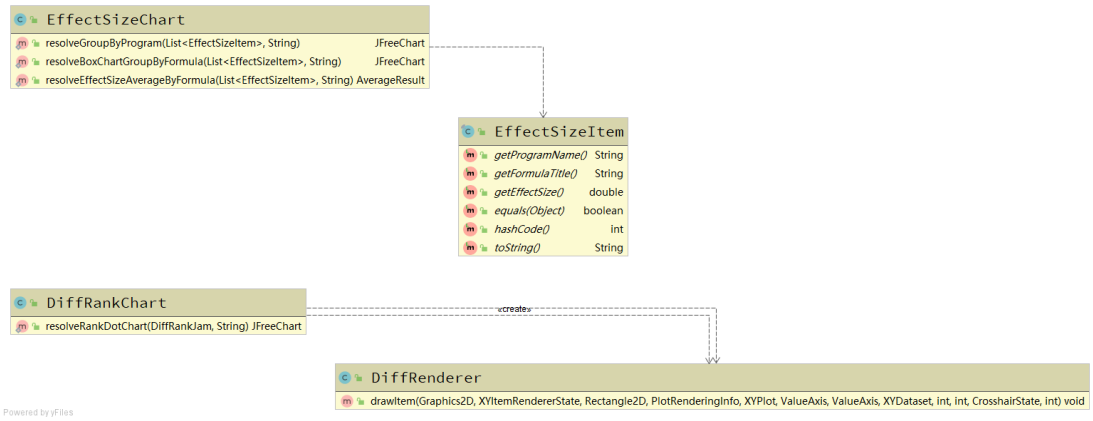

图 5-10　chart 包中的类图

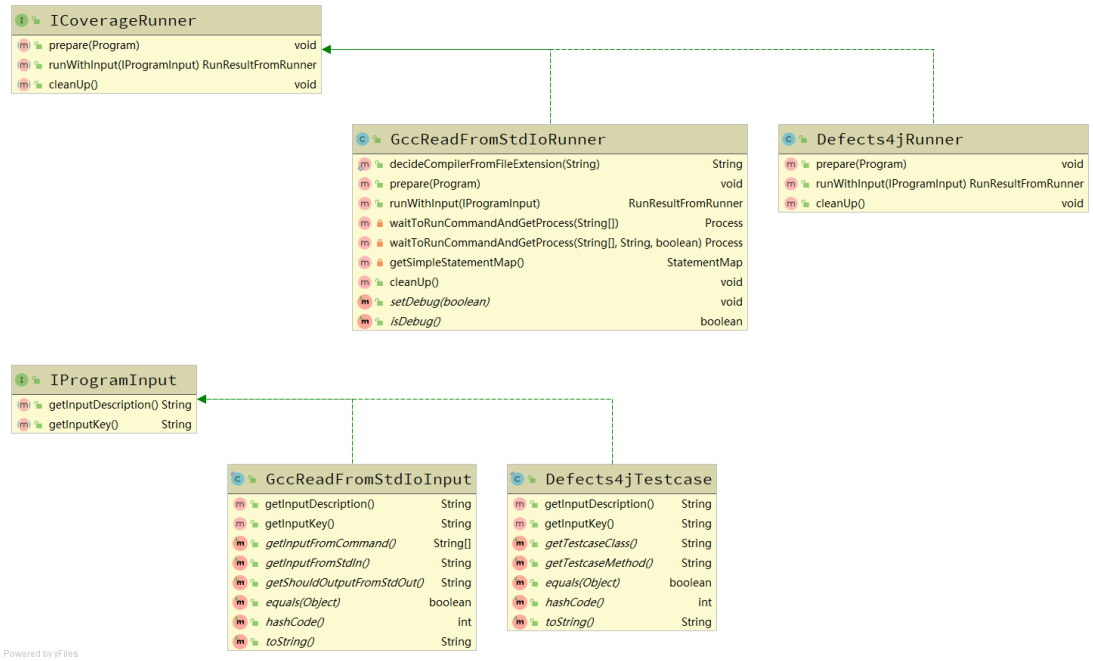

图 5-11　runner 包中的类图

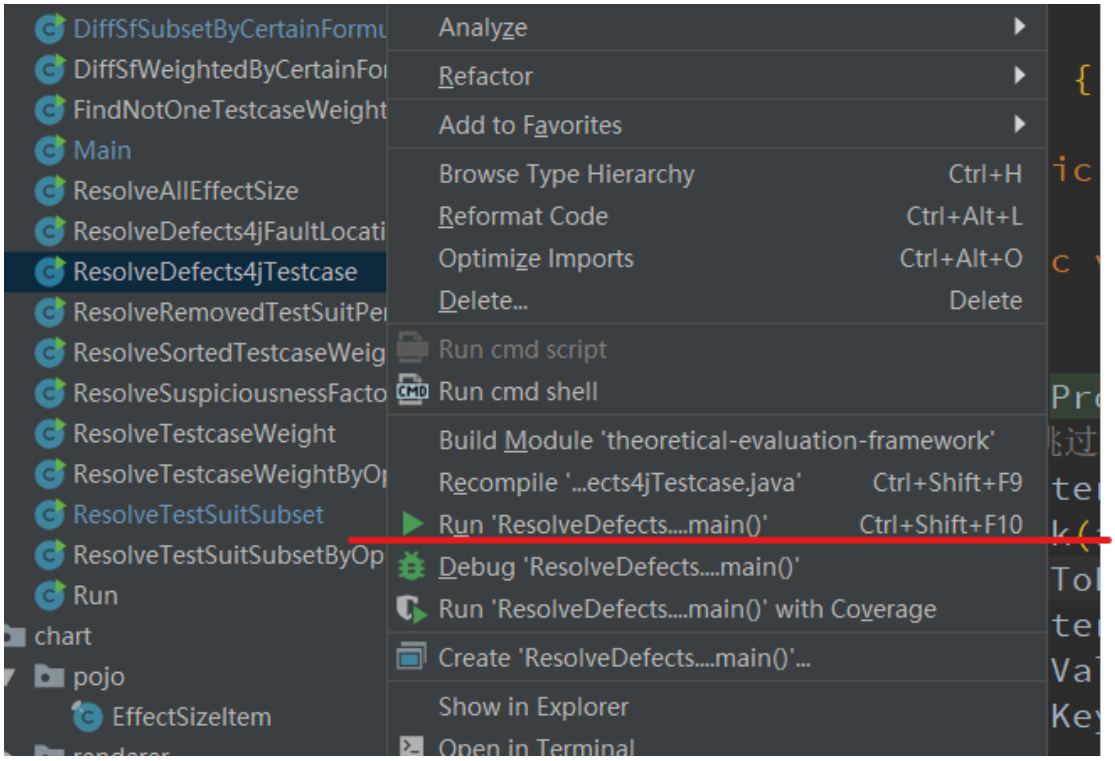

图 5-12　在 IDE 中运行试验程序

第 六 章　实验验证

6.1 实验对象

本次实验使用了 7 个 C 语言程序和 4 个 Java 程序。实验对象分别在下面两节中列出。

6.1.1 C 语言实验对象

本次的 C 语言实验对象分别由 Software-artifact Infrastructure Repository[22] 中的 5 个程序和本文自行添加测试用例的 2 个程序组成。所有的 C 语言程序均使用变异方式人工添加错误。表6-1列出了所有的程序，其中 R 表示测试用例和变异版本来自 Software-artifact Infrastructure Repository，M 表示测试用例和变异版本为自行添加。

表 6-1　使用 C 语言的实验对象

来源	程序名	功能描述	故障版本数	测试用例数	代码行数
R	tcas	空中防撞系统	41	1500	174
R	print_tokens	词法分析器	7	4130	726
R	schedule2	优先级调度器	10	2710	374
R	replace	模式替换	32	5542	564
R	tot_info	信息统计	23	1052	565
M	expression_parser	表达式解析和运算	13	1361	1039
M	sort	各类排序算法比较	10	1500	2512

6.1.2 Java 实验对象

本次的 Java 实验对象均来自 Defects4j[23]，其中的错误是程序在开发过程中真实出现的错误。程序在表6-2中。由于 Defects4j 中不同的版本测试用例不一样，因此在表格中只统计了测试用例数量的最大值和最小值。

6.2 实验涉及的频谱故障定位技术

本实验涉及的可疑因子计算公式如表6-3所示。

表 6-2 使用 Java 的实验对象

程序名	项目名称	故障版本数	测试用例数
Chart	JFreeChart	26	1591∼2193
Lang	Apache commons-lang	65	1540∼2291
Math	Apache commons-math	106	817∼4378
Time	Joda-Time	27	3749∼4041

表 6-3 频谱故障定位技术（可疑因子计算公式）

公式名称	公式
Tarantula	$\frac{\frac{a_{ef}}{f}}{\frac{a_{ef}}{f}+\frac{a_{ep}}{p}}$
Ochiai	$\frac{a_{ef}}{\sqrt{f\times(a_{ef}+a_{ep})}}$
Ochiai2	$\frac{a_{ef}\times a_{np}}{\sqrt{(a_{ef}+aep)\times(a_{nf}+a_{np})\times(a_{ef}+a_{np})\times(a_{nf}+a_{ep})}}$
Op	$a_{ef}-\frac{a_{ep}}{p+1}$
SBI	$1-\frac{a_{ep}}{a_{ep}+a_{ef}}$
Jaccard	$\frac{a_{ef}}{f+a_{ep}}$
Kulcznski1	$\frac{a_{ef}}{a_{nf}-a_{ep}}$
Dstar2	$\frac{a_{ef}^2}{a_{ep}+a_{nf}}$
o	if $a_{nf}>0$ then -1 else a_{np}

6.3 实验方法

我们在 11 个实验对象上，使用 9 个可疑因子计算公式分别对测试套件进行了优化，并将在优化后的测试套件上计算出来的错误语句排名同优化之前的错误语句排名进行比较。由于我们已经事先知道了实验对象中具体的错误语句的行号，因此可以直接查看相应的错误语句的排名变化情况。如果在优化之后大多数的错误语句排名均出现上升的情况，代表实验结果是成功的。为了将计算结果使用数值的方式直观地显示出来，我们使用了 Effect Size（效应量）来表示优化效果。整体实验流程如图6-1所示。

6.3.1 统计的错误版本

由于本实验是探究的对于频谱分析的优化效果，如果实验对象的错误之处在于“缺少了语句”（即在修复时需要添加语句，而不是删除或修改语句），

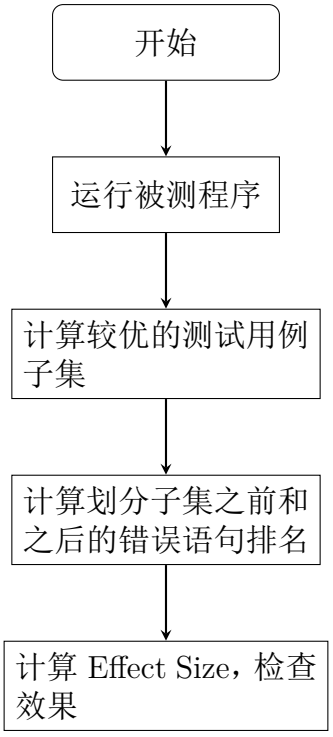

图 6-1　总体实验流程

其错误之处将无法使用频谱进行分析，也缺乏对优化效果的度量方法。因此，在本实验中，我们将排除所有使用了“添加语句”的方法来修复程序的错误版本。表6-4指出了全部错误版本数和本实验使用的错误版本数之间的对比。

6.4 实验结果分析

在后文的分析章节，我们采用了效应量（Effect Size）来对实验结果的好坏进行评价。效应量是一个用于表现两个集合之间相关程度的值，目前常用的效应量度量方法有如下两种。

1. Pearson 相关系数。主要用来计算两个集合之间的相关程度。

2. Cohen’s d 度量。主要用来计算两个集合之间的差异。

由于本实验的主要目的是探究测试套件优化前后，在错误语句排名上的区别。因此使用 Cohen’s d 度量作为效应量的计算公式。其计算公式6.4.1如下。

$$d = \frac{u_1 - u_2}{s} \tag{6.4.1}$$

表 6-4 在计算 effect size 和绘制图表时使用的错误版本数

程序名称	全部错误版本数	在计算结果时使用的错误版本数
my_sort	7	7
schedule2	10	5
tcas	41	41
tot_info	23	22
replace	32	31
print_tokens	7	4
expression_parser	13	12
Chart	26	13
Math	106	54
Time	27	6
Lang	65	21

其中，u_1 表示第一个样本的平均值，u_2 表示第二个样本的平均值。s 表示两个样本的并合标准差，其公式由6.4.2给出。

$$s = \sqrt{\frac{(n_1 - 1) \times s_1^2 + (n_2 - 1) \times s_2^2}{n_1 + n_2 - 2}} \tag{6.4.2}$$

上文公式中，n_1 和 n_2 分别是两个样本的大小，s_1^2 和 s_2^2 表示两个样本的方差。

本文中的实验，将优化之前的排名数据作为上文公式中下标为 1 的数据，将优化之后的排名数据作为下标为 2 的数据，由于排名越高（数值越小）的数据越好，因此 Effect Size 为正数代表在优化之后的测试套件上的错误语句平均排名要比优化之前高，反之则更低。其绝对值越大，代表两组数据（优化之前和之后）相差越大。简而言之，Effect Size 在数值上越大，则实验效果越好。

由于效应量是一个标准化的数值，因此我们有一种度量方法，来表示其大小。度量方法如下所示。

1. 小效应量：d = 0.20

2. 中效应量：d = 0.50

3. 大效应量：d = 0.80

6.4.1 实验结果的可视化展示

Effect Size 的计算结果如图6-2所示。

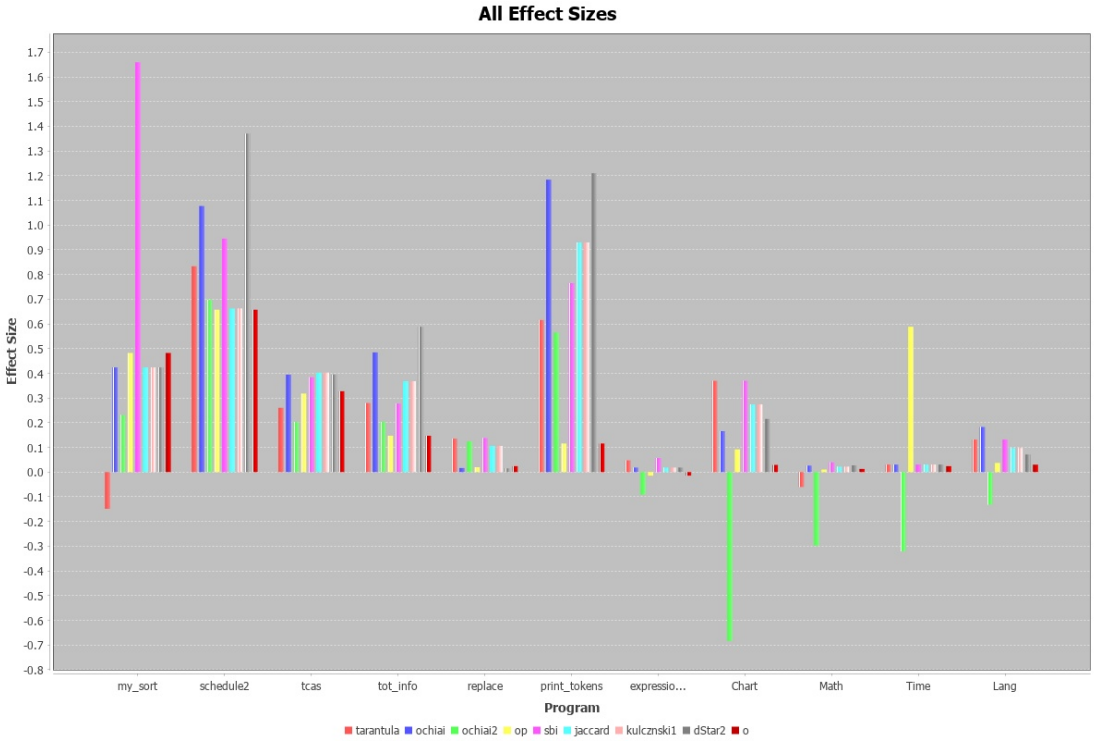

图 6-2　所有实验对象和公式的 Effect Size

Effect Size 在各个公式上的平均值如图6-3所示。

通过本实验，移除的测试用例占总测试用例的百分比如图6-4所示。

6.4.2 基于效应量的实验结果分析

根据试验结果可知，在取得的子集上运行测试用例，使得实验对象中的错误语句排名有了明显的上升。对于每个公式而言，其效应量的平均值均为正数（代表平均下来每条错误语句的排名都有所升高）。对于每个程序单独的效应量而言，可以看到在 SIR[22] 上的效应量明显高于 Defects4j[23] 上的效应量。因为 SIR 的测试套件中有大量的重复（冗余）测试用例，所以通过削减测试用例数量来优化测试套件的方法对其效果较好；而 Defects4j 中不存在冗余的测试用例，因此结果相较并不明显。值得注意的是，公式 Ochiai2 在每个 Defects4j 测试对象上都得到了负的效应量，说明其并不等价于 O 或 Op

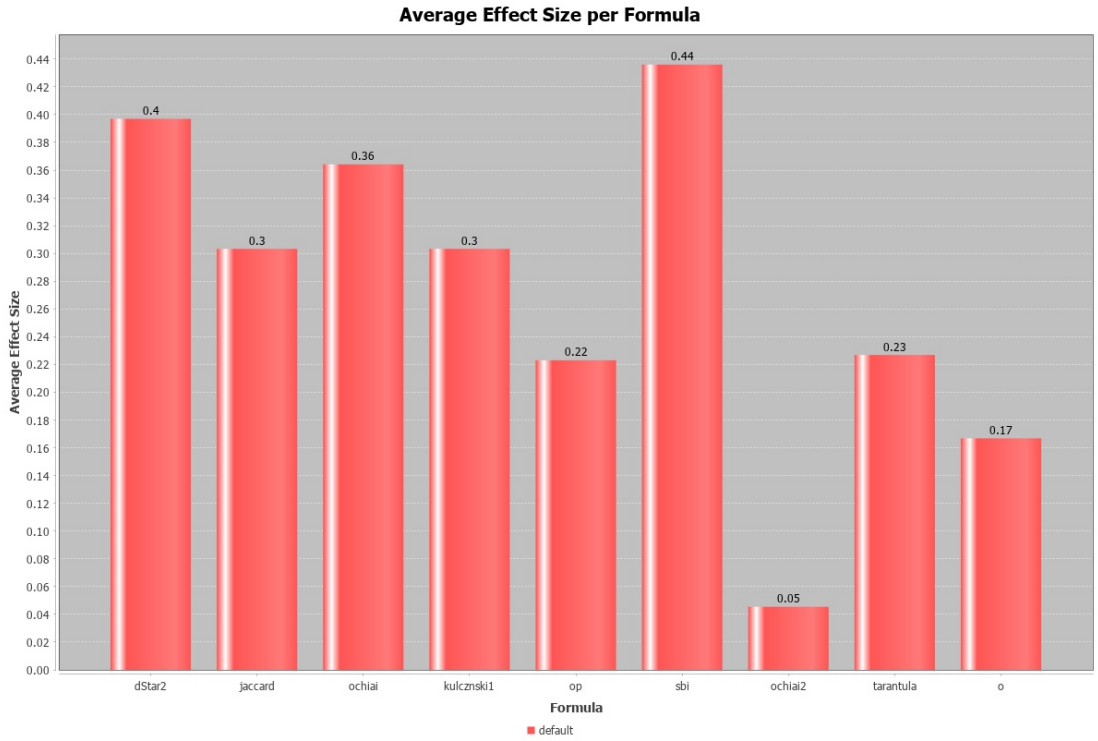

图 6-3　各个公式的 Effect Size 平均值

公式，因此本优化方案对其的优化效果并不好。

就移除的测试用例数量而言，由于 Defects4j 数据集使用的是非冗余测试用例，并且其单元测试中只有少数几个测试用例可以覆盖到错误语句，因此绝大多数的测试用例都被此方法削减了。

从效应量还可以看出，本测试套件优化方法在 Op 上的运行结果并不是最好的。因为在大部分的测试用例运行结果中，Op 得到的错误语句排名已经非常靠前，因此本优化方法能对其产生的提升效果非常的有限。图6-5中可以看出这一点。这张图是将 Op 公式作为 SFL 公式，使用本测试套件优化方法在程序 replace 上得出的结果。其横轴的每一格代表一条错误语句，每一条错误语句对应两个点，红点表示在优化之前该语句的可疑因子排名，蓝点表示优化之后的排名，如果排名相等，只显示红点。其中，点在图上的位置越高代表排名越高，说明定位效果越好。如果在优化之后排名上升（蓝点比红点高），则这两点之间使用蓝线连接；否则使用红线连接。根据该图的结果可见，绝大部分的错误语句在优化前已经获得了很高的排名，因此对其优化较为有限。

除此之外，部分公式在某个程序上的效应量出现了负数，expression_parser 的运行结果在大多数公式上也并不好，这说明本优化方式对

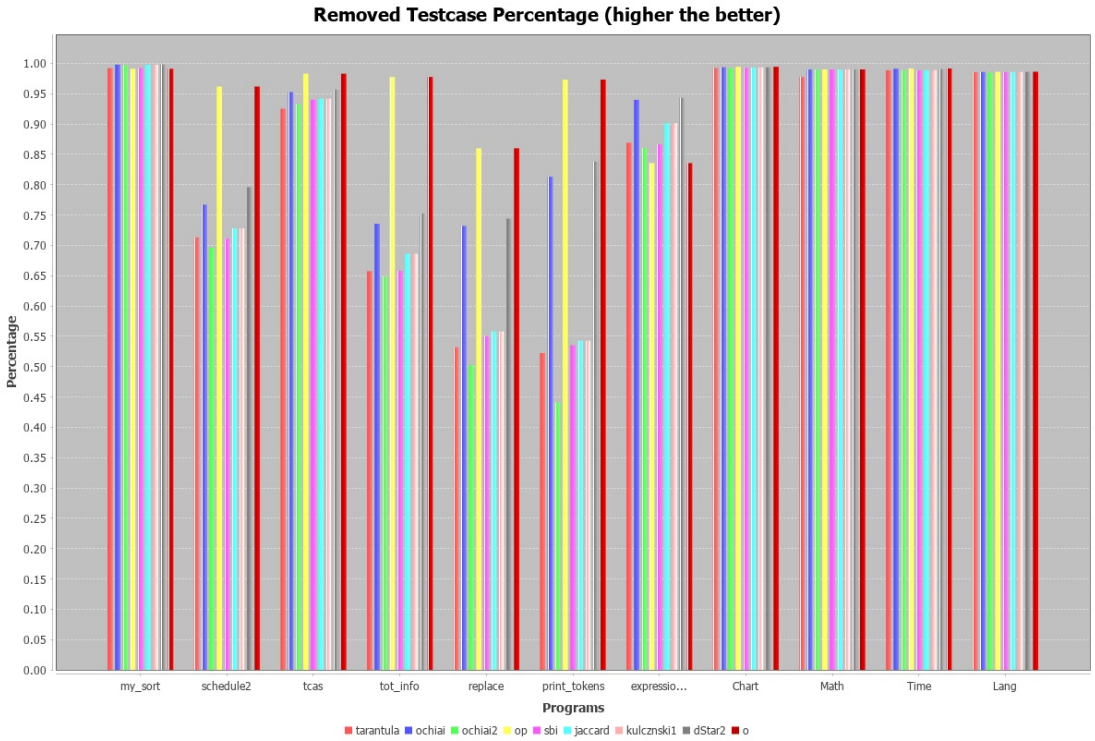

图 6-4　移除的测试用例的百分比

于错误定位并不一定是绝对会优化的，在某些情况下，错误定位的性能可能会下降。

6.5 小结

本次实验通过一个 Java 编写的实验程序，在两个测试数据集共 11 个实验对象上运行了 9 个 SFL 公式，并比较应用了本文提出的测试套件优化方法之前和之后的结果。使用效应量来量化优化效果，并将结果可视化表示。

经过实验，我们可以得出以下结论：本测试套件优化方案在 9 个 SFL 公式中的 8 个上，对每个被测程序上都有定位效率提升；另一个公式只在一半的被测程序（SIR 测试集）上有效率提升。而从平均效应量来看，本优化方案对于参与实验的每个 SFL 公式，都有优化效果。

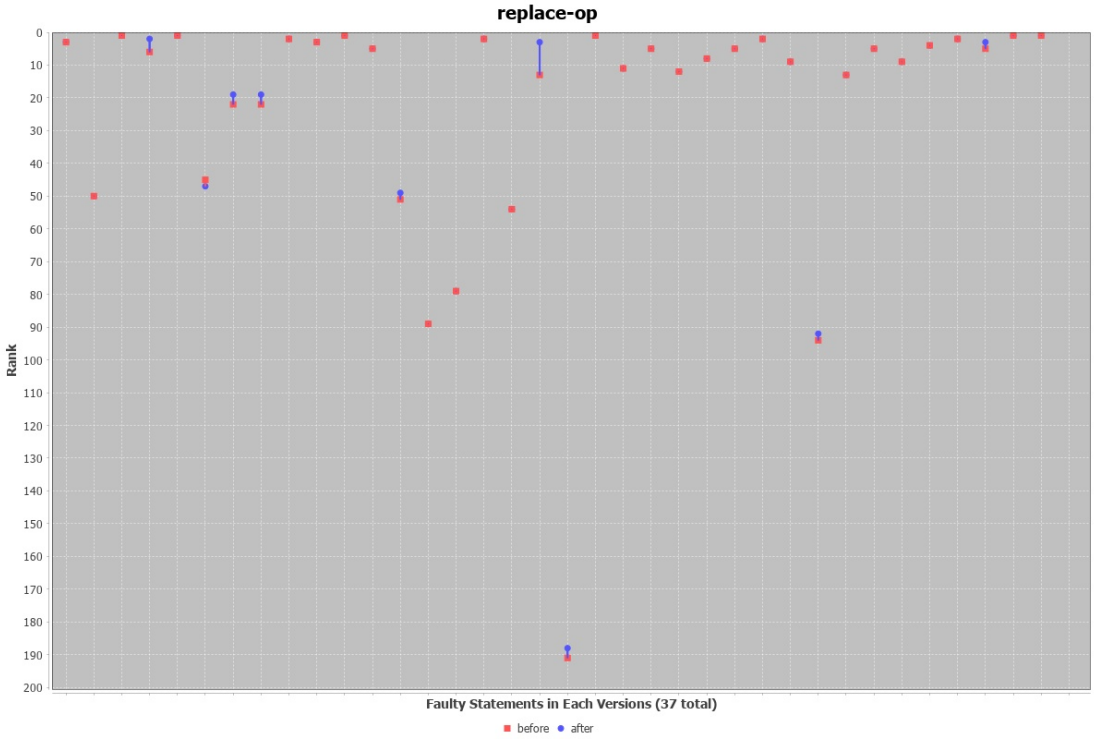

图 6-5　将 Op 作为 SFL 公式应用在程序 replace 上的结果

第 七 章　总结与展望

本文主要包含了以下内容：

1. 我们提出了一个向量表模型，和一个基于平均排名代价的测试套件质量评估方案。
2. 提出了一个测试套件优化方案（使用贪心算法来缩减测试用例数量）。
3. 构造了一个辅助工具，用来实现在前文中描述的优化方案。
4. 在两个测试数据集上应用了我们的辅助工具，并使用多种方法分析其试验结果（以效应量为主）

在运行测试程序进行分析时，我们还运行了 Defects4j 中的 Mockito，但由于其优化结果是“去掉所有的测试用例”，因此我们没有将其包含在本文的结果统计分析中。经过分析，我们发现 Mockito 中绝大多数的错误版本都是多故障程序，虽然这些故障都有相同的模式 [24]，但这些错误代码并不是在每一个测试用例中都会运行到的，因此每条故障语句的 anf 都不为 0，在优化过程中就被去除了。

Defects4j 中另一个没有包含在此文章中的项目是 Clojure，这个项目有超过 200 个故障版本，每个版本有超过 7000 个测试用例，测试用例的执行速度太慢，就没有统计到本文中。因此，在以后的实验中，我们将采取一些方案来优化 Defects4j 的测试用例的执行过程，比如，只执行针对故障类的测试用例。由于本文中执行了所有的测试用例，却只统计了故障类的覆盖率，因此在统计结果里显示移除了绝大部分的 Defects4j 测试用例。这样的统计结果其实并不精确；可见优化执行方案的必要性。

除此之外，本项目还有很多需要继续研究的地方，比如，公式 Ochiai2 在 Defects4j 数据集上得到了负数的效应量，expression_parser 的优化结果并不好，其原因仍需要分析。

参考文献

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

[2] Naish L, Lee H J. Duals in spectral fault localization[C]//2013 22nd Australian Software Engineering Conference. IEEE, 2013: 51-59.

[3] Wong W E, Qi Y, Zhao L, et al. Effective fault localization using code coverage[C]//31st Annual International Computer Software and Applications Conference (COMPSAC 2007): volume 1. IEEE, 2007: 449-456.

[4] Chen M Y, Kiciman E, Fratkin E, et al. Pinpoint: Problem determination in large, dynamic internet services[C]//Proceedings International Conference on Dependable Systems and Networks. IEEE, 2002: 595-604.

[5] Choi S S, Cha S H, Tappert C C. A survey of binary similarity and distance measures [J]. Journal of Systemics, Cybernetics and Informatics, 2010, 8(1):43-48.

[6] Wong W E, Debroy V, Gao R, et al. The dstar method for effective software fault localization[J]. IEEE Transactions on Reliability, 2013, 63(1):290-308.

[7] Lee H J, Naish L, Ramamohanarao K. The effectiveness of using non redundant test cases with program spectra for bug localization[C]//2009 2nd IEEE International Conference on Computer Science and Information Technology. IEEE, 2009: 127-134.

[8] Wong W E, Debroy V, Choi B. A family of code coverage-based heuristics for effective fault localization[J]. Journal of Systems and Software, 2010, 83(2):188-208.

[9] Baudry B, Fleurey F, Le Traon Y. Improving test suites for efficient fault localization [C]//Proceedings of the 28th international conference on Software engineering. ACM, 2006: 82-91.

[10] Yu Y, Jones J, Harrold M J. An empirical study of the effects of test-suite reduction on fault localization[C]//2008 ACM/IEEE 30th International Conference on Software Engineering. IEEE, 2008: 201-210.

[11] Abreu R, Zoeteweij P, Van Gemund A J. On the accuracy of spectrum-based fault localization[C]//Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007). IEEE, 2007: 89-98.

[12] Hao D, Pan Y, Zhang L, et al. A similarity-aware approach to testing based fault localization[C]//Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. ACM, 2005: 291-294.

[13] Jiang B, Zhang Z, Chan W K, et al. How well does test case prioritization integrate with statistical fault localization?[J]. Information and Software Technology, 2012, 54 (7):739-758.

[14] Rao P, Zheng Z, Chen T Y, et al. Impacts of test suite's class imbalance on spectrum-based fault localization techniques[C]//2013 13th International Conference on Quality Software. IEEE, 2013: 260-267.

[15] Masri W, Assi R A. Cleansing test suites from coincidental correctness to enhance fault-localization[C]//2010 Third International Conference on Software Testing, Verification and Validation. IEEE, 2010: 165-174.

[16] Bandyopadhyay A. Improving spectrum-based fault localization using proximity-based weighting of test cases[C]//2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). IEEE, 2011: 660-664.

[17] Steimann F, Frenkel M, Abreu R. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators[C]//Proceedings of the 2013 International Symposium on Software Testing and Analysis. ACM, 2013: 314-324.

[18] Landsberg D, Sun Y, Kroening D. Optimising spectrum based fault localisation for single fault programs using specifications.[C]//FASE. 2018: 246-263.

[19] Xie X, Wong W E, Chen T Y, et al. Spectrum-based fault localization: Testing oracles are no longer mandatory[C]//2011 11th International Conference on Quality Software. IEEE, 2011: 1-10.

[20] Ma C, Nie C, Chao W, et al. A vector table model-based systematic analysis of spectral fault localization techniques[J]. Software Quality Journal, 2018:1-36.

[21] Chen T Y, Xie X, Kuo F C, et al. A revisit of a theoretical analysis on spectrum-based fault localization[C]//2015 IEEE 39th Annual Computer Software and Applications Conference: volume 1. IEEE, 2015: 17-22.

[22] Software-artifact infrastructure repository[EB/OL]. 2005. http://sir.csc.ncsu.edu/php/index.php.

[23] Defects4j[EB/OL]. https://github.com/rjust/defects4j.

[24] Defects4j repair mockito no. 6[EB/ OL]. https://github.com/program-repair/defects4j-dissection/commit/615be9f.

[25] Shen S, Wang Z, Zhang W. LaTeX-template-for-npu-thesis[Z]. 2016.

致　谢

感谢马老师提供给我这个机会，能够接触到业界前沿的技术，并亲自参与其中。感谢我们学校的三位同学开发出的毕业论文 LaTeX 模板 [25]，在我编写论文时提供了很多方便。

毕业设计小结

毕业论文是大学四年的最后一份大作业，通过此类研究，我们学习到了在学术研究活动中所需要的技能，明确了论文的写法，对于我们后续的科研活动有很大的帮助。