软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(5):2043−2062 [doi: 10.13328/j.cnki.jos.007436] [CSTR: 32375.14.jos.007436]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向两阶段分组的测试用例优先级排序方法*

钱忠胜, 秦朗悦, 范赋宇, 付庭峰

(江西财经大学 计算机与人工智能学院, 江西 南昌 330032)

通信作者: 钱忠胜, E-mail: changesme@163.com

摘　要: 测试用例优先级排序方法 TCP 在缓解测试开销方面备受关注. 基于不同优先级策略的贪心算法是 TCP 中常用的方法. 然而, 现有基于贪心算法的 TCP 技术多使用单一的排序策略, 且每轮迭代排序过程一次性考虑全部测试用例, 而未关注用例间的关系, 导致在覆盖信息和排序方面的处理上耗时过长, 极大地降低了排序效率. 同时, 在单一的排序策略中, Additional 策略得到广泛研究, 但其受随机因素影响较大, 当出现平局时, 通常会随机选择待排用例, 影响排序的有效性. 基于此, 提出一种面向两阶段分组的测试用例优先级排序方法 TPG-TCP. 第 1 阶段进行粗粒度测试用例分组, 通过挖掘用例间的隐藏关系, 将它们分为关键用例组和普通用例组, 为下一阶段采用多样性策略排序做准备, 以提高排序效率. 第 2 阶段进行细粒度测试用例分组排序, 根据迭代次数将关键用例分组, 为减少 Additional 策略中随机因素的干扰, 提出基于用例潜力度的 TP-Additional 策略对一部分关键用例排序, 同时采用简单高效的 Total 策略对普通用例与另一部分关键用例排序, 将排序结果追加至 TP-Additional 策略的排序结果中, 在排序有效性提升的同时也提高了效率. 通过与 8 种相关方法在 6 个数据集上的对比结果发现, 所提方法是高效且可行的, 在 *APFD* 与 *TETC* 指标上分别平均提升约 1.29% 和 9.54%.

关键词: 测试用例优先级排序; 两阶段分组; 贪心算法; Additional 策略

中图法分类号: TP311

中文引用格式: 钱忠胜, 秦朗悦, 范赋宇, 付庭峰. 面向两阶段分组的测试用例优先级排序方法. 软件学报, 2026, 37(5): 2043–2062. http://www.jos.org.cn/1000-9825/7436.htm

英文引用格式: Qian ZS, Qin LY, Fan FY, Fu TF. Test Case Prioritization Approach Based on Two-phase Grouping. Ruan Jian Xue Bao/Journal of Software, 2026, 37(5): 2043–2062 (in Chinese). http://www.jos.org.cn/1000-9825/7436.htm

Test Case Prioritization Approach Based on Two-phase Grouping

QIAN Zhong-Sheng, QIN Lang-Yue, FAN Fu-Yu, FU Ting-Feng

(School of Computer and Artificial Intelligence, Jiangxi University of Finance and Economics, Nanchang 330032, China)

Abstract: Test case prioritization (TCP) has gained significant attention due to its potential to reduce testing costs. Greedy algorithms based on various prioritization strategies are commonly used in TCP. However, most existing greedy algorithm-based TCP techniques rely on a single prioritization strategy and process all test cases simultaneously during each iteration, without considering the relationships between test cases. This results in excessive computational overhead when handling coverage information and performing prioritization, thus reducing overall efficiency. Among single-strategy approaches, the Additional strategy has been extensively studied but remains highly sensitive to random factors. When a tie occurs, test cases are typically selected at random, compromising prioritization effectiveness. To address these issues, a test case prioritization approach based on two-phase grouping (TPG-TCP) is proposed. In the first phase, coarse-grained grouping is conducted by mining hidden relationships among test cases, thus dividing them into a key group and an ordinary group. This lays the groundwork for applying diversity-based strategies in the next phase to enhance prioritization efficiency. In the second

* 基金项目: 国家自然科学基金 (62262025); 赣鄱俊才支持计划主要学科学术和技术带头人领军人才项目 (20243BCE51024); 江西省自然科学基金重点项目 (20224ACB202012)
收稿时间: 2024-12-04; 修改时间: 2025-02-02; 采用时间: 2025-03-23; jos 在线出版时间: 2025-07-30
CNKI 网络首发时间: 2025-07-31

phase, fine-grained prioritization of test cases is performed. Key test cases are further subdivided based on the number of iterations. To mitigate the randomness inherent in the Additional strategy, a TP-Additional strategy based on test case potency is introduced to prioritize a portion of the key test cases. Meanwhile, a simple and efficient Total strategy is applied to prioritize the ordinary test cases and remaining key test cases. The results from the Total strategy are appended to those produced by the TP-Additional strategy. This method improves both the effectiveness and efficiency of test case prioritization. Experimental results on six datasets, compared with eight existing methods, demonstrate that the proposed method achieves average improvements of 1.29% in *APFD* and 9.54% in *TETC*.

Key words: test case prioritization (TCP); two-phase grouping; greedy algorithm; Additional strategy

测试用例优先级排序 (test case prioritization, TCP) 技术在不减少测试用例数量的情况下, 按照某种特定的准则或目标重新安排整个测试套件中测试用例的执行顺序, 使得有较高优先级的测试用例优先执行, 在缓解测试开销方面备受关注[1−4].

在 TCP 中, 贪心算法因简单高效, 适用性强, 已成为解决用例排序问题的常用算法[5], 包括 Additional 策略与 Total 策略, 分别由 Elbaum 等人[6]和 Rothermel 等人[7]提出. 这两种策略自 1999 年提出以来因其实用性强而受到广泛关注[8]. 它们的基本思想是先执行覆盖程序语句多的测试用例. 其中, Total 策略是优先选择总覆盖语句数最多的测试用例, Additional 策略是优先选择覆盖目前未被覆盖的语句数最多的测试用例[9]. Additional 策略因其反馈机制的引入, 相比于 Total 策略可应用于更复杂的场景. 然而, 有研究工作[10]指出, 基于 Additional 策略的贪心算法在优先级划分上耗费的时间过长. 一些方法通过使用额外的数据结构减少冗余数据访问[8,10,11]以改进 Additional 策略, 但它们大多使用单一的排序策略, 且每轮迭代排序过程一次性考虑全部测试用例, 而未关注用例间的关系, 导致在覆盖信息和排序方面的处理上耗时过长. 即使某些方法采用多类型排序策略, 但其策略选择指标较单一, 无法应用于不同程序中, 指标划分不灵活, 应用性较差. 因此, 利用测试用例间的关系以丰富策略选择指标, 并使用多样性排序策略对缓解 TCP 方法的效率至关重要.

除效率之外, TCP 的有效性也是影响其方法广泛应用的另一个重要方面. Additional 策略在故障检测率方面, 已被认为是最有效的 TCP 技术之一[11]. 但 Additional 策略易受随机因素干扰, 当出现平局时, 通常会随机选择待排测试用例, 影响排序的有效性. 已有研究使用历史覆盖值[11], 或者根据用例覆盖程序的分支情况与用例覆盖程序的均衡程度[5]减弱 Additional 策略随机性对故障检测有效性的影响. 然而, 这些方法仅考虑用例覆盖标准的整体性, 忽略了其个体性, 且随机性的改进方法无法提升排序效率, 不能减少每轮排序迭代的时间, 无法同时提升排序的有效性与效率. 因此, 设计能缓解 Additional 策略受随机性影响, 同时加快每轮排序迭代过程的方法, 对提升排序有效性和效率有重大意义.

基于上述两个方面的问题, 本文提出一种面向两阶段分组的测试用例优先级排序 (test case prioritization based on two-phase grouping, TPG-TCP) 方法. 首先, 通过挖掘测试用例间的隐藏关系, 将待排测试用例进行粗粒度分组, 分为关键用例组和普通用例组. 然后, 根据迭代次数将关键用例组进行细粒度分组, 对于一部分关键用例, 提出基于用例潜力度的 TP-Additional 策略排序得到主序列; 对于普通用例组与另一部分待排关键用例, 采用简单高效的 Total 策略排序得到次序列, 将其追加至主序列后得到一个优先级排序序列, 这样通过多轮迭代后得到一个序列集. 最后, 对每个优先级排序序列实施故障检测操作, 并根据故障检测报告得到最佳优先级排序序列.

本文主要工作与贡献如下.

1) 利用测试用例间的隐藏关系, 将待测用例集分为关键用例组与普通用例组, 为下一阶段采用多样性策略排序做准备, 以减少在覆盖信息和排序方面的时间消耗, 以提高排序效率.

2) 根据迭代次数, 将关键用例分组, 对部分关键用例采用提出的基于用例潜力度的 TP-Additional 策略排序, 降低 Additional 策略随机性的影响, 提升排序有效性.

3) 提出一种两阶段分组 TCP 框架, 将基于测试用例间隐藏关系的粗粒度分组与基于 TP-Additional 策略的细粒度分组排序相结合, 同时提升排序的有效性与效率.

4) 与 8 种当前经典的、主流的 TCP 方法在 6 个程序上展开实验对比, 所提 TPG-TCP 在有效性和效率上均占优势.

1 相关工作

在测试用例优先级排序中, 基于相似性的 TCP 方法和基于贪心算法的 TCP 方法是两种常见的技术. 基于相似性的 TCP 方法优先考虑已选择用例中最不相似的测试用例, 增加测试的广度和深度, 旨在快速发现软件中的缺陷. 这种方法通常利用聚类分析技术来识别测试用例的相似性, 可有效减少冗余测试行为并提高测试效率. 然而, 聚类分析法常涉及较复杂的数据处理操作, 若将其与基于贪心算法的 TCP 方法结合会极大地降低算法效率. 但是, 受基于相似性的 TCP 方法的启发, 利用测试用例表现出的相似行为, 即用例覆盖信息相似性, 可简化排序操作. 为避免聚类分析带来的复杂度, 更好地将测试用例的相似性用于基于贪心算法的 TCP 方法中, 我们采用简单的算法挖掘测试用例间的关系, 将具有包含行为关系的用例分为一组, 可优化用例的排序策略, 提高排序效率.

基于相似性的 TCP 方法主要关注测试用例之间的相似性, 将相似的测试用例分组, 从而在测试阶段进行更有效的排序和执行. Miranda 等人[10]提出 Fast 技术, 借用大数据领域常用的算法来查找相似性, 并在白盒与黑盒测试中提供可扩展的基于相似性的测试用例优先级排序. 此外, 近年来大多数基于相似性的 TCP 研究通过聚类分析法捕获相似的测试用例. Wang 等人[12]提出一种基于聚类的自适应测试用例优先级排序方法, 可在预优先级排序中添加新的自适应调整内容. Chen 等人[13]提出一种基于聚类的静态黑盒测试用例优先级排序方法, 采用 K-medoids 方法捕获测试用例的相似性进行更精细地划分. Li 等人[14]提出一种基于语义感知的两阶段 TCP 框架, 利用基于信息检索对测试用例进行初步排序和过滤, 并使用预训练的 Siamese 语言模型基于语义相似性对用例进行细粒度排序.

基于贪心算法的 TCP 方法通过贪心策略优先选择能够覆盖最多未覆盖代码的测试用例, 以提高测试覆盖率和缺陷检测效率. 这种方法侧重于最大化每一步的收益, 通过逐步选择最优解来达到整体最优的效果. Elbaum 等人[6]最早提出基于贪心算法的 TCP 技术, 包括 Total 策略和 Additional 策略. Li 等人[8]使用多个邻接表以减少基于 Additional 策略的贪心算法中的冗余数据访问, 并建议将迭代次数减少到相对较小的值, 以在保持方法有效性的同时提高效率. Zhang 等人[11]提出 OCP 算法, 引入一种通用的部分注意机制, 该机制根据先前选择的用例优先级值进行排序, 并根据上一轮用例优先级值打破 Additional 策略陷入平局的情况. 范书平等人[5]提出一种基于关键用例获取的测试用例排序方法, 根据候选用例的分支偏离度和候选用例覆盖新分支情况, 对用例进行排序.

然而, 在基于贪心算法的 TCP 研究中, 大多数方法将邻接矩阵作为输入, 少部分使用邻接表. 将邻接表作为输入相比于邻接矩阵排序效率较高, 可在很大程度上缩减集合规模, 以降低用例相关性计算的复杂度. 并且, 以往的研究只根据用例覆盖标准的整体状态选择测试用例, 忽略了其个体性, 即覆盖标准的被覆盖信息丢失. 此外, 基于 Additional 策略方法排序指标单一, Additional 策略随机性的选择会降低故障检测的有效性.

综上分析可知, 当前基于贪心算法的 TCP 方法未利用低复杂度的可挖掘测试用例覆盖信息相似性的方法, 从而造成在每轮选择测试用例时将所有的候选用例考虑在内, 影响排序效率; 此外, 多数 TCP 方法采用单一的排序策略和排序指标, 无法同时提高排序有效性和效率. 针对这两个方面的问题, 本文提出一种面向两阶段分组的测试用例优先级排序方法 TPG-TCP. 该方法利用集合判断操作挖掘测试用例间的隐藏关系, 将测试用例进行粗粒度分组, 避免在后续的排序中考虑所有的测试用例, 以提高排序效率; 另外, 根据迭代次数将关键用例进行细粒度分组, 并增加用例潜力度指标改进 Additional 策略, 对更能暴露问题的关键用例进行排序, 剩余用例采用简单高效的 Total 策略排序, 以同时提升排序的有效性与效率.

2 面向两阶段分组的测试用例优先级排序方法

大多数基于贪心算法的 TCP 技术在处理覆盖信息和排序时存在耗时过长、排序效率低的问题. 其中, 贪心算法的 Additional 策略容易受到随机因素的影响, 导致排序的有效性降低. 基于此, 提出一种面向两阶段分组的测试用例优先级排序方法 TPG-TCP, 主要包含粗粒度用例分组与细粒度用例分组排序这两个模块, 其整体框架如图 1 所示.

1) 粗粒度用例分组, 减少在覆盖信息和排序处理上的时间消耗. 将用于存储覆盖信息的覆盖矩阵转换为用例-语句列表, 基于该列表对各用例的语句覆盖关系进行检测, 将待测用例分为关键用例集和普通用例集. 为下一阶段

采用多样性策略排序做准备.

2) 细粒度用例分组排序, 提高排序效率的同时提升排序有效性. 根据迭代次数对关键用例实施细粒度分组的方法, 以优化排序策略. 针对能快速发现潜在问题的部分关键用例, 采用基于用例潜力度的 TP-Additional 策略排序, 生成主用例序列. 对于普通用例组和余下的关键用例, 采用 Total 策略排序, 生成次用例序列. 将次用例序列合并到主用例序列后, 形成一个优先级排序序列. 通过多轮迭代, 得到一个包含多个排序序列的序列集合. 最终, 对每个优先级排序序列执行故障检测操作, 根据检测结果产生最优序列.

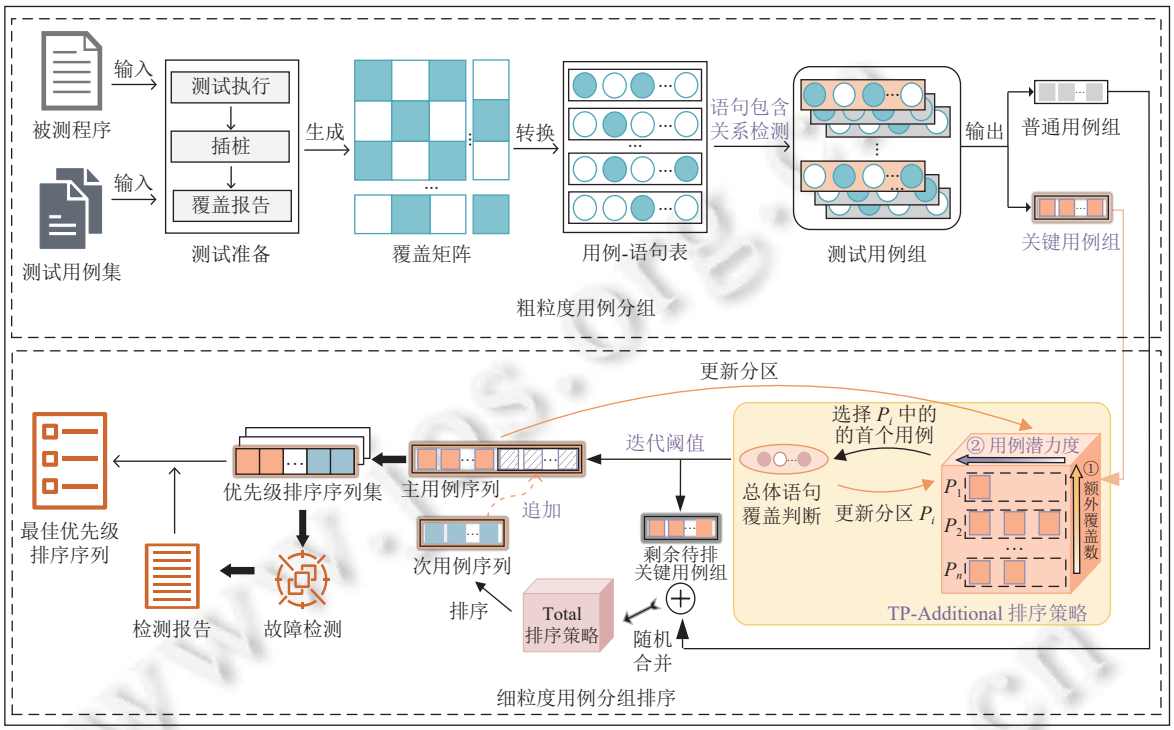

图 1 面向两阶段分组的测试用例优先级排序方法框架

为便于理解, 对文中一些主要符号进行说明, 如表 1 所示.

表 1 主要符号及含义

符号	含义	符号	含义
T	无序用例集	G	分区集
K, N	关键/普通用例集	WG	待排分区集
S	用例覆盖语句集	g_{count}	分区覆盖语句数
S_k, S_n	关键/普通用例覆盖语句集	tp	用例潜力度
t_{count}	用例覆盖语句数	R_k	关键用例中剩余待排集
SC	语句-用例数表	R	总剩余待排用例集
TC	用例-语句数表	M	迭代阈值
S_{cov}	语句标记表	P', P''	主/次用例序列
T_{cov}	用例标记表	P	最佳优先级排序序列

此外, 为更好地理解本方法, 先给出一个示例表来展示测试套件的覆盖信息. 我们使用邻接表表示覆盖信息, 如表 2 所示. 该测试套件包括 8 个测试用例 (即 t_1, t_2,…, t_8) 以及 8 个程序元素 (即 e_1, e_2,…, e_8), 这里, 程序元素指语句、分支、方法等.

表 2 覆盖信息示例

测试用例	覆盖元素			
t_1	e_1	e_2	e_3	e_4
t_2	e_1	e_5	e_7	e_8
t_3	e_1	e_3	e_4	e_6
t_4	e_1	e_2	e_5	—
t_5	e_1	e_2	e_6	—
t_6	e_1	e_3	e_4	—
t_7	e_1	e_2	e_3	e_4
t_8	e_1	e_3	e_4	e_7

2.1 粗粒度测试用例分组

近来, 通过聚类分析法挖掘用例间的关系在 TCP 中得到广泛研究, 但聚类分析法常涉及较复杂的数据处理操作, 与基于 Additional 策略的 TCP 方法结合会增加算法复杂性. 然而, 利用用例间的关系来优化用例排序仍值得深入研究.

本文实现了基于邻接表的粗粒度用例分组算法以对用例排序进行优化. 与 Li 等人[8]类似, 我们使用代码覆盖率分析工具 Clover (https://bitbucket.org/atlassian/clover/src/master) 收集各数据集的覆盖信息, 生成的覆盖分析报告会以矩阵形式表示. 为降低获取用例间隐藏关系的复杂度, 本文将覆盖矩阵转换为用例-语句邻接表来存储用例覆盖信息. 此外, 采用相比于聚类算法更简单、高效的集合判断操作以计算用例的相关性, 并基于用例间的这种隐藏关系对测试用例进行粗粒度分组. 这有助于在下一阶段的排序过程中根据测试用例重要性采用多样性的排序策略, 通过减少在覆盖信息和排序方面的处理时间, 以优化排序. 一方面, 有助于减少用例相关性的计算次数, 以提高挖掘用例潜在关系效率; 另一方面, 也可在下一阶段减少对覆盖信息的冗余访问, 以提高排序效率.

基于邻接表挖掘用例间的隐藏关系, 将每一个待排用例的覆盖元素视作一个集合, 通过集合判断两用例间的隐藏包含关系. 这样, 将用例进行粗粒度分组的主要思想为: ① 当某个用例的覆盖信息可真包含另一用例信息时, 选择可覆盖更多程序元素的用例加入关键用例组, 被其他用例包含的用例放入普通用例组; ② 当两个测试用例的覆盖信息完全相同时, 表示它们的检测功能是一致的, 我们将其视为相同的用例, 为保持关键用例组的简洁性, 我们仅保留序号较小的用例在关键用例组中, 将序号较大的用例划分到普通用例组; ③ 当两个测试用例的覆盖信息不存在包含关系时 (即, 除①②情况外), 将它们均加入关键用例组. 隐藏包含关系的表示见公式 (1).

$$\begin{cases} K=\begin{cases} K\cup t_i, & S_i\supset S_j,\ \forall i,j \\ K\cup t_i, & S_i=S_j,\ i<j \\ K\cup t_i\cup t_j, & S_i\not\subseteq S_j\wedge S_j\not\subseteq S_i,\ \forall i,j \end{cases} \\ N=\begin{cases} N\cup t_j, & S_i\supset S_j,\ \forall i,j \\ N\cup t_j, & S_i=S_j,\ i<j \end{cases} \end{cases} \tag{1}$$

其中, K 为关键用例组, N 为普通用例组, S_i 为用例 t_i 覆盖的语句集, S_j 为 t_j 覆盖的语句集.

例 1: 本文以表 2 为例, 将邻接表作为算法输入. 通过集合判断操作, 可发现用例 t_3 的覆盖信息 $S_3=\{e_1, e_3, e_4, e_6\}$可真包含 t_6 的覆盖信息 $S_6=\{e_1, e_3, e_4\}$, 故将 t_3 加入关键用例组, t_6 加入普通用例组; 用例 t_1 的覆盖信息 $S_1=\{e_1, e_2, e_3, e_4\}$与 t_7 的覆盖信息 $S_7=\{e_1, e_2, e_3, e_4\}$相同, 且 t_1 的序号小于 t_7 的序号, 故将 t_1 加入关键用例组, t_7 加入普通用例组; 其余用例 t_2、t_4、t_5、t_8 间不存在包含关系, 故均加入关键用例组. 因此, 关键用例组 $K=[t_1, t_2, t_3, t_4, t_5, t_8]$, 普通用例组 $N=[t_6, t_7]$.

通过对用例进行粗粒度分组, 有助于在后续排序阶段减少冗余覆盖信息的访问, 也简化了测试用例排序和执行的流程. 粗粒度用例分组的具体过程见算法 1.

算法 1. 基于邻接表的粗粒度用例分组过程.

输入: 无序用例集 T, 用例覆盖语句集 S;
输出: 关键用例集 K, 普通用例集 N, 关键用例覆盖语句集 S_k, 普通用例覆盖语句集 S_n.

Begin
1. 关键用例集 $K \leftarrow \varnothing$;
2. 普通用例集 $N \leftarrow \varnothing$;
3. **for** t_i in T // 遍历无序用例集 T
4. **if** t_i in K **then** // 当前用例为关键用例
5. **continue**; // 接下来无需寻找与 t_i 覆盖语句具有包含关系的用例
6. **end if**
7. **for** t_j in T
8. **if** t_j in K **then**
9. **continue**;
10. **end if**
11. **if** $S[i] \subseteq S[j]$ **then** // 用例 t_j 的覆盖语句可包含 t_i 的覆盖语句
12. Add t_j into K; // 将可包含 t_i 覆盖语句的用例 t_j 加入关键用例集
13. **break**; // 搜寻下一个能包含当前用例覆盖语句的用例
14. **end if**
15. **if** $S[j] \subseteq S[i]$ **then** // 用例 t_i 的覆盖语句可包含 t_j 的覆盖语句
16. Add t_i into K; // 将可包含 t_j 覆盖语句的用例 t_i 加入关键用例集
17. **end if**
18. **if** $S[i] \not\subseteq S[j]$ and $S[i] \not\supseteq S[j]$ **then** // t_i 与 t_j 的覆盖语句不存在包含关系
19. Add t_i and t_j into K; // 将 t_j 与 t_i 加入关键用例集
20. **end if**
21. $j = j + 1$;
22. **end for**
23. $i = i + 1$;
24. **end for**
25. $N \leftarrow T - K$; // 将未分组用例视为普通用例
26. **Output** K, N, S_k, S_n;
End

在算法 1 中, 第 1, 2 行初始化关键用例集 K 和普通用例集 N. 第 3–26 行通过集合判断操作检测各用例间的语句包含关系. 先判断当前检测用例是否已在关键用例集 K 和普通用例集 N 中, 若在, 则跳过检测该用例与其他用例间的包含关系以加快检测效率. 再判断是否存在包含关系, 将可包含其他用例覆盖信息的用例与覆盖信息互不存在包含关系的用例加入关键用例集 K. 第 25 行, 将无序用例集 T 中不属于关键用例集 K 的用例加入普通用例集 N. 第 26 行输出关键用例集 K 和普通用例集 N 以及对应的覆盖语句集 S_k 和 S_n.

2.2 细粒度测试用例分组排序

为同时提高排序的有效性和效率, 我们对测试用例进行细粒度分组排序. 一方面, 在 Zhang 等人[11]分区排序算法基础上, 提出以用例潜力度为二次选择指标的 TP-Additional 策略对部分关键用例进行排序, 实现 TPG-TCP 方法; 另一方面, 受 Li 等人[8]利用多列表储存排序数据的启发, 我们增加了语句-用例数表 SC, 以加快计算待排用

例的用例潜力度效率, 并根据迭代次数对关键用例进行细粒度分组.

我们使用多列表储存排序过程所需的数据, 在后续排序过程中会动态更新这些列表, 减少冗余数据访问. 各列表相关描述如表 3 所示.

表 3 各列表信息描述

名称	功能
语句-用例数表SC	记录每条语句被测试用例覆盖的次数
用例-语句数表TC	记录每个测试用例覆盖的语句数
语句标记表S_{cov}	记录每条语句是否被已选择测试用例覆盖
用例标记表T_{cov}	记录测试用例是否被选择

本文设计语句-用例数表 SC 用于存储计算用例潜力度所需数据, 加快排序的效率. 其中, 用例潜力度是我们提出的一个二次排序指标, 用于表示一个测试用例发现重要问题快慢的能力.

对于单个测试用例, 统计它覆盖的每条程序语句在所有测试用例中被覆盖的次数, 从中选取次数的最小值, 并将该最小值的倒数定义为测试用例的“用例潜力度”, 见公式 (2).

$$tp_i = \frac{1}{\min(SC_i)} \tag{2}$$

其中, $SC_i = [cov_1, cov_2, \ldots, cov_n]$ 表示测试用例 t_i 的语句-用例数表, cov_j 表示语句 j 被用例覆盖的次数. $\min(SC_i)$ 表示用例 t_i 覆盖的程序语句分别被全体用例覆盖次数的最小值, tp_i 表示 t_i 的用例潜力度.

一方面, 一个测试用例覆盖的程序元素被其他用例覆盖的次数越少, 意味着该用例覆盖的代码部分相对独特, 暴露潜在重要问题的可能性更高, 用例潜力度越大. 优先考虑执行用例潜力度较大的测试用例可更快地检测程序中难以发现的问题. 另一方面, 根据用例潜力度, 我们能利用原始的覆盖信息帮助排序, 当两个测试用例的语句覆盖数 (未被已选择的用例覆盖的剩余语句) 相等, 即额外覆盖数一样时, 将优先选择更具代表性和影响力的用例, 即潜力度更大的用例, 可加快发现重要问题的速度, 同时提高测试的有效性和效率.

例 2: 见表 2, $SC = [8, 4, 5, 5, 2, 2, 2, 1]$, 即 e_1–e_8 元素被覆盖的次数分别为 8、4、5、5、2、2、2、1. 其中, t_1 覆盖了 e_1、e_2、e_3、e_4 这 4 个元素, 由 SC 表可知 $SC_1 = [8, 4, 5, 5]$, 故 t_1 的用例潜力度为 SC_1 表中最小值的倒数, 即结果为 1/4; t_2 覆盖了 e_1、e_5、e_7、e_8 这 4 个元素, 由 SC 表可知 $SC_2 = [8, 2, 2, 1]$, 故 t_2 的用例潜力度为 1; t_3 覆盖了 e_1、e_3、e_4、e_6 这 4 个元素, 由 SC 表可知 $SC_3 = [8, 5, 5, 2]$, 故 t_3 的用例潜力度为 1/2. 因此, 在 t_1、t_2、t_3 均未被选择且其覆盖元素个数相同时, 根据 t_1、t_2、t_3 的用例潜力度大小关系 $tp_2 > tp_3 > tp_1$, 优先选择 t_2.

此外, 我们还根据迭代次数对关键用例进行细粒度分组. 每完成一轮迭代, 根据 TP-Additional 策略部分关键用例会被选择, 我们将这些用例视为能更快暴露问题的用例组; 剩余关键用例自动成组. 每当某个测试用例被选择, 将其覆盖的语句标记为“已覆盖”, 持续地选择测试用例直到所有可能语句均被覆盖并已标记完, 根据给定策略进行测试用例排序, 然后将所有可能语句重置为“未覆盖”的过程, 我们称其为一次迭代, 即语句标记表 S_{cov} 的值由初始为全 0 经过多次更新转化为全 1, 再重置为全 0 的过程, 见公式 (3):

$$S_{cov} = [a_1, a_2, \ldots, a_n] \text{ 且 } a_i = \begin{cases} 1, \text{ if 语句}i\text{被已选用例覆盖} \\ 0, \text{ if 语句}i\text{未被已选用例覆盖} \end{cases} \tag{3}$$

例 3: 以表 2 和例 1 中的关键用例组为例. 初始时, 语句标记表 $S_{cov} = [0, 0, 0, 0, 0, 0, 0, 0]$, 关键用例组 $K = [t_1, t_2, t_3, t_4, t_5, t_8]$, t_1、t_2、t_3、t_4、t_5、t_8 的覆盖语句数分别为 4、4、4、3、3、4. 先选出覆盖语句数多的用例, 即 t_1、t_2、t_3、t_8, 再根据用例潜力度进行选择, t_1、t_2、t_3、t_8 的用例潜力度分别为 1/4、1、1/2、1/2, 故排序时优先选择 t_2, 并更新 $S_{cov} = [1, 0, 0, 0, 1, 0, 1, 1]$; 接下来, 根据 S_{cov} 计算 t_1、t_3、t_8 的额外覆盖语句数分别为 3、3、2, 通过覆盖语句数选出 t_1 和 t_3, 再比较 t_1 和 t_3 的用例潜力度大小优先选择 t_3, 并更新 $S_{cov} = [1, 0, 1, 1, 1, 1, 1, 1]$; 继续根据 S_{cov} 计算 t_1 和 t_8 的额外覆盖语句数分别为 1 和 0, 故直接选择 t_1, 无需考虑用例潜力度, 并更新 $S_{cov} = [1, 1, 1, 1, 1, 1, 1, 1]$. 这样, 当依次选择 t_2、t_3、t_1 后, 所有可能语句已被全覆盖, 在下一轮迭代之前将 S_{cov} 重置为 0. 在本轮迭代

中, 将 t_2、t_3、t_1 视为能更快暴露问题的用例组, 剩余 t_4、t_5、t_8 成为一组.

根据迭代次数将关键用例进行分组的原因是, 经过多轮迭代后, 所有可能语句都已被覆盖了足够的次数. 若持续迭代, 被覆盖的语句再暴露问题的可能性就会降低. 一方面, 若多轮迭代后所有故障均被发现, 则剩余迭代对于故障检测而言只会增加时间成本; 另一方面, 若经过多轮迭代后仍存在部分故障, 则它们较难被发现, 故剩余迭代只是偶然地、个别地揭示错误.

因此, 在已达到足够迭代次数的情况下, 将关键用例针对性地分为能更快地暴露问题的用例组和剩余关键用例组, 并利用多种排序策略以应对不同复杂程度的排序场景, 即对不同分组使用不同策略进行排序. 分组排序使排序过程更加灵活和高效, 有助于保证有效性的同时提高方法效率. 细粒度分组排序的一次迭代过程见算法 2.

算法 2. 基于 TP-Additional 策略的细粒度分组排序.

输入: 关键用例集 K, 关键用例覆盖语句集 S_k;
输出: 主用例序列 P'.

Begin
1. 根据算法 1 得到的 S_k 初始化 TC、SC、S_{cov}、T_{cov};
2. $P' \leftarrow \varnothing$, $G \leftarrow \varnothing$, $WG \leftarrow \varnothing$; // 初始化主用例序列 P'、分区集 G、待排分区集 WG
3. $g_{\text{count}} = |S_{cov}|$; // 当前分区覆盖语句数初始为语句标记表 S_{cov} 的长度
4. *UpdateG*(G); // 更新分区
5. **while** $G.size > 0$
6. **for** each (t_i, t_{count}) in G
7. **if** $t_{\text{count}} = g_{\text{count}}$ **then** // 当前分区覆盖语句值与搜索用例覆盖语句值相等
8. $temp = TC[i]$; // 从用例-语句数表 TC 读取当前搜索用例的覆盖语句值
9. **if** $temp \neq g_{\text{count}}$ **then** // 当前搜索用例的覆盖语句值与分区覆盖语句不相等
10. *UpdateG*(G); // 更新分区
11. **else**
12. $WG \leftarrow WG \cup \{(t_i, t_{\text{count}})\}$; // 将当前搜索用例及覆盖语句值加入待排分区
13. **end if**
14. **end if**
15. **end for**
16. **if** $g_{\text{count}} > 0$ **then**
17. **if** WG.size > 0 **then**
18. **for** each (t_i, t_{count}) in WG
19. $tp_i = \dfrac{1}{\min(SC_i)}$; // 计算用例 t_i 的用例潜力度, 见公式 (2)
20. Add tp_i to *tpList*; // 将用例潜力度加入潜力度列表
21. **end for**
22. $index = tpList.index(\max(tpList))$; // 获取单个分区中用例潜力度最大的一个用例位置
23. Add t_{index} to P';
24. *Delete*(t, t_{count}) in G; // 在分区 G 中删除已选用例及其覆盖值
25. **for** each a_i in S_{cov} // 见公式 (3)
26. **if** not a_i **then** // 仍存在未被覆盖的语句
27. Update SC, TC; // 更新语句-用例数表 SC 和用例-语句数表 TC
28. Update S_{cov}, T_{cov}; // 更新语句标记表 S_{cov} 和用例标记表 T_{cov}

```
29.            else
30.               Reset SC, TC, S_cov; // 重置 SC、TC、S_cov 表
31.            end if
32.         end for
33.      else
34.         g_count = g_count − 1;
35.      end if
36.   end if
37. end while
38. Output P′;
    /* 更新分区子函数 UpdateG */
39. Method UpdateG(G)
40.   for t_count in TC // 遍历用例-语句数表 TC
41.      if G.containsKey(t_count) // 分区集 G 中存在语句覆盖值键
42.         Add (t_i, t_count) to G[t_count]; // 将用例加入分区覆盖语句数为 t_count 的子分区
43.      else // G 中不存在语句覆盖值键
44.         G.put(t_count, [(t_i, t_count)]); // 新建分区覆盖语句数为 t_count 的子分区
45.      end if
46.   end for
End
```

在算法 2 中, 第 1, 2 行是初始化操作. 第 3 行将当前分区覆盖语句数的初始值设为语句标记表 S_{cov} 的长度. 第 4 行是子函数 *UpdateG*(), 用于新建分区 G 并将测试用例加入子分区. 第 5–37 行对部分关键用例进行优先级排序. 每轮迭代均尝试找到与当前检索分区覆盖语句数相同的测试用例, 并将其添加到主用例序列 P'中. 其中, 第 6–15 行按分区覆盖语句数从大到小选择子分区. 若当前搜索用例的覆盖语句数与分区覆盖语句数不相等, 则对分区进行更新; 若相等, 则将子分区中的所有用例添加到待排分区集 WG 中. 第 18–24 行选择待排分区集 WG 中具有最大潜力度的用例加入主用例序列 P', 并将此用例从分区 G 中删除. 第 25–32 行检查是否存在语句未被 P'中的用例覆盖. 若仍存在语句未被覆盖, 则根据 P'中的用例更新 SC、TC、S_{cov}、T_{cov} 表; 若所有语句均已被覆盖, 则重置 SC、TC、S_{cov} 表. 在第 34 行, 若待排分区集 WG 无测试用例可选择, 则检索下一个分区. 第 38 行输出主用例序列 P'. 值得注意的是, 虽然用例潜力度被作为二次排序的指标, 但也有可能出现两个用例潜力度相等的情况, 在这种情况下, 与 Additional 策略类似, 我们的方法随机选择某一测试用例.

2.3 测试用例优先级排序过程

已有工作[8]表明, 少数测试用例的顺序变化才会对故障百分比指标产生较大影响, 这些用例更具有代表性且更能暴露问题. 基于此, 首先, 我们采用基于多种指标排序的 TP-Additional 策略对第 2.2 节中能更快暴露问题的用例组进行排序, 得到主用例序列 $P'=<t_1,t_2,\ldots,t_n>$. 然后, 针对普通用例 (见第 2.1 节) 与经过多轮迭代未被选择的测试用例 (见第 2.2 节) 随机合并后的大多数用例, 使用更加简单高效的 Total 策略进行排序, 即按照用例的语句覆盖数大小排序, 得到次用例序列 $P''=<t'_1,t'_2,\ldots,t'_n>$. 最后, 我们将主用例序列 P'与次用例序列 P''进行线性拼接得到 $P_{temp}=P'\oplus P''$, 即$<t_1,t_2,\ldots,t_n,t'_1,t'_2,\ldots,t'_n>$.

例 4: 以表 2、例 1 的普通用例组, 以及例 3 的分组结果为例. 普通用例组 $N=[t_6, t_7]$, t_1、t_2 与 t_3 已通过 TP-Additional 策略完成排序, 得到主用例序列 $P'=<t_2, t_3, t_1>$, 剩余用例组为 $[t_4, t_5, t_8]$. 将普通用例组 N 与剩余用例组的用例随机合并得到待排集$\{t_4, t_5, t_6, t_7, t_8\}$, 它们的语句覆盖数分别为 3、3、3、4、4. 按照语句覆盖数大小对其排序 (若覆盖数相等, 则随机选择), 可得到次用例序列 $P''=<t_8, t_7, t_5, t_4, t_6>$. 将 P'与 P''进行线性拼接得到 $P_{temp}=$

<t_2, t_3, t_1, t_8, t_7, t_5, t_4, t_6>, 即为一轮迭代产生的优先级排序序列.

拼接完成后, 用 P_{temp} 的测试用例进行故障检测, 根据故障检测结果可得到最佳优先级排序序列 P. 面向两阶段分组的具体过程见算法 3.

算法 3. 面向两阶段分组的测试用例优先级排序过程.

输入: 关键用例集 K, 普通用例集 N, 主用例序列 P', 迭代阈值 M, 用例标记表 T_{cov};
输出: 最佳优先级排序序列 P.

Begin
1. 获得关键用例集 K 和普通用例集 N // 见算法 1
2. $m = 0$;
3. **while** $T_{cov}.include(0)$
4. **if** $m \leqslant M$ **then**
5. $P' = TP\_Additional(K)$; // 对关键用例集 K 的部分用例排序, 见算法 2
6. $m = m + 1$;
7. Update T_{cov}; // 将已排序的用例在用例标记表 T_{cov} 中标记为“已标记”
8. **else**
9. **for** $i = 1$ to $|T_{cov}|$
10. **if** $T_{cov}[i] = 0$ **then**
11. $R_k \leftarrow R_k \cup K[i]$; // 关键用例中剩余未排用例加入待排序列 R_k
12. **end if**
13. **end for**
14. **end if**
15. $R = R_k \cup N$; // 合并关键用例中剩余未排用例集 R_k 与普通用例集 N
16. $P'' = Total(R)$; // 对 R 中所有用例使用 Total 策略排序
17. $P_{\text{temp}} = P'.append(P'')$; // 将 P''中的所有用例拼接在 P'后
18. $Result \leftarrow Fault\_test(P_{\text{temp}})$; // 用优先级序列 P_{temp} 对故障进行检测
19. **end while**
20. $P \leftarrow Best\_P(Result)$; // 根据故障检测结果 $Result$ 输出最佳优先级排序序列 P
21. **Output** P;
End

在算法 3 中, 第 1 行根据基于邻接表的粗粒度用例分组算法获得关键用例集 K 和普通用例集 N. 第 2 行迭代次数初始化为 0, 第 3–19 行对测试用例进行优先级排序. 其中, 第 3–7 行根据迭代阈值 M 对部分关键用例 K 使用基于 TP_Additional 策略排序得到主序列 P', 并将已排序用例在用例标记表 T_{cov} 标记为“已标记”. 第 8–14 行将关键用例中剩余未排用例加入待排序列 R_k. 第 15 行合并关键用例中剩余未排用例 R_k 与普通用例集 N. 第 16 行对合并后的用例使用 Total 策略排序得到次序列 P''. 第 17 行将次序列 P''中的所有用例拼接在主序列 P'后. 第 18 行用优先级序列 P_{temp} 对故障进行检测, 用 $Result$ 记录每轮检测结果. 第 20 行根据故障检测结果得到最佳优先级排序序列 P, 并在第 21 行输出.

3 实验设计与分析

3.1 问题提出

为验证 TPG-TCP 方法的有效性与高效性, 本节通过多种综合实验进行验证与对比分析, 主要回答以下 3 个方

面的问题.

RQ1: 本文方法的不同构件对排序性能有何影响? 是否有存在的必要性?

针对此问题, 在第 3.4.1 节设置了消融实验, 根据粗粒度用例分组、细粒度用例分组排序这 2 大模块中的 3 个构件组成的 6 种变体方法, 分别分析这些构件对 TPG-TCP 方法的整体影响. 实验结果表明, 这 6 种变体方法的排序性能均在不同程度上不如所提方法 TPG-TCP, 说明这 3 个构件在 TCP 方面是有效的.

RQ2: 与经典的、较新的相关方法对比, 本文方法有何优势? 效果如何?

针对此问题, 第 3.4.2 节设置了对比实验, 分别将所提方法 TPG-TCP 在 6 个数据集上与 8 种相关方法做对比. 实验结果表明, 在平均缺陷检测百分比 (average percentage of fault detect, *APFD*) 和测试执行时间成本 (test execution time cost, *TETC*) 这两个评价指标上, TPG-TCP 方法均优于其他对比方法. 这主要得益于粗粒度分组阶段用例筛选过程与细粒度排序策略的改进, 可在提高排序速度的同时提高其有效性.

RQ3: 迭代次数对本文方法的性能有何影响?

针对此问题, 在第 3.4.3 节进行了迭代次数划定的实验, 分析不同迭代次数在不同规模数据集下对方法性能的影响, 以便更好地平衡方法的有效性与效率, 进一步优化排序方法.

3.2 数据集与评价指标

实验均在 Windows 10 操作系统下进行, 环境为 8 GB 内存、1.8 GHz 主频的 AMD Ryzen 7 4800U 处理器 (16 核), 编程语言使用 Python 3. 为兼顾计算资源和时间成本等因素, 并减少随机性因素带来的影响, 每组实验运行 50 次, 最终结果取其平均值 (表 7 数据以其下方说明为准).

3.2.1 数据集

我们从近期的研究[8,11,13,15–17]中筛选了 6 个常用数据集, 它们涵盖了从 3k 行代码的小型项目到超过 100k 行代码的大型项目, 且为了对比的公平性, 尽量选择具有相似评估指标的数据集. 数据集来源如下: ① Flex、Tcas、Schedule2 这 3 种数据集源自著名测试数据库 SIR (software-artifact infrastructure repository)[18], SIR 库中的项目被广泛用于评估 TCP 技术[5,10,11,19]; ② Jsoup 与 Jacksoncore 数据集源于 Defects4J 数据库[20], 常用于评估 TCP 技术, 在最新的 TCP 研究[15,21]中也得到应用; ③ Camel-core 是来源于 GitHub 上的大型开源工业项目, 其集成逻辑复杂, 功能多样化, 是评估 TCP 有效性的重要选择[8]. 这 6 种开源数据集的统计信息如表 4 所示.

表 4 数据集统计信息

数据集	代码行数	测试用例数	突变故障数
Flex	79200	567	1289
Tcas	7093	1608	876
Schedule2	3740	2710	780
Jsoup	4512	41	346
Jacksoncore	27908	135	351
Camel-core	120248	5623	13005

已有工作[8,11,15]表明突变故障适用于软件测试评估, 且该技术已被广泛应用于 TCP 评估, 故本文引入突变故障技术以评估不同 TCP 性能. 使用 PIT 突变工具[22]中的 11 个突变运算符以生成突变体, 并利用 TCE[23]消除部分重复与等价的突变体. 而等价性是不可判定的, 故 TCE 不能去除所有等价突变体, 但有助于缓解过多重复突变体对方法有效性造成的影响, 最终在剩余的突变体集中为每个项目构建一个突变组. 各数据集的突变故障数见表 4.

3.2.2 评价指标

APFD 是当前测试用例优先级研究领域中的一个核心指标[8,11,15–17], 已被广泛应用于衡量测试用例排序序列检测缺陷的能力. 本文与上述工作类似, 采用 *APFD* 作为评价指标, 以评估各 TCP 方法的有效性. 此外, 增加 *TETC* 指标用于评估各 TCP 方法的效率.

1) *APFD* 是平均缺陷检测百分比, 用于衡量测试用例排序序列检测缺陷的能力, 该能力越强说明检测方法越有效. 给定一个包含 n 个用例的测试用例集 T, P 是 T 的排序后序列, P 的 *APFD* 计算如公式 (4) 所示:

$$APFD = 1 - \frac{TF_1 + TF_2 + \ldots + TF_m}{nm} + \frac{1}{2n} \tag{4}$$

其中, TF_i 表示检测到第 i 个 ($1 \leqslant i \leqslant m$) 缺陷的测试用例在 T 中的位置, m 是检测到的缺陷总数. 因在同一测试中任何测试序列的 n 和 m 都是固定的, 故 $APFD$ 值越高表明按测试序列 P 检测缺陷的速度越快, 其对应的 TCP 方法越有效.

2) $TETC$ 是测试执行时间成本, 用于衡量测试的执行效率, 如公式 (5) 所示:

$$TETC = t_{\text{last}} - t_{\text{first}} \tag{5}$$

其中, t_{first} 是测试的开始时间, t_{last} 是测试的结束时间. $TETC$ 表示使用 TCP 方法生成排序序列花费的总时间, $TETC$ 值越低, 则其对应的 TCP 方法效率越高.

3.3 对比方法

这里将所提方法 TPG-TCP 与 8 个经典的、先进的方法做对比, 对比方法主要分为如下 4 大类. ① 基于贪心策略的 TCP 方法. GA[6]是最先将 Additional 策略用于 TCP 中的方法, 而 OCP[11]在其基础上融入部分注意力思想, 并增加二次排序指标, 以提高排序性能. 此外, AGA[8]通过增加额外数据结构并减少迭代次数的方法提高了 GA 的效率. OCP 和 AGA 方法与本文方法的细粒度用例分组模块有相似性. ② 基于群智能算法的方法. CGWO-TCP[16]和 WOA-TCP[17]是近年来将先进的群智能算法应用于 TCP 领域的方法. ③ 基于故障的 TCP 方法. FB-TCP[15]考虑多种排序因素, 将多个单一的 TCP 技术集成以提高 TCP 的有效性, 与本文方法的细粒度用例分组模块有相似性. ④ 基于机器学习的方法. KS-TCP[13]运用聚类方法挖掘测试用例间的相似性以提高 TCP 性能, 与本文方法的粗粒度用例排序模块具有一定相关性. SAT-TCP[14]将 TCP 过程分为两个阶段, 基于语义相似度利用信息检索和预训练模型分别对测试用例进行粗粒度和细粒度排序, 与本文方法的两阶段排序框架相似.

1) GA: 一种优先选择可最大覆盖剩余未被覆盖代码数测试用例的方法.

2) OCP: 一种基于部分注意机制的 TCP 方法, 其采用历史优先级避免考虑所有候选用例.

3) AGA: 通过增加数据结构存储覆盖信息以减少冗余数据访问, 并探究迭代次数对 TCP 效率的影响.

4) CGWO-TCP: 该方法使用混沌函数提高现有灰狼优化 (GWO) 算法在收敛速度方面的性能, 以确定用例优先级.

5) WOA-TCP: 一种使用 n 维有向搜索空间改进鲸鱼优化算法以优化 TCP 决策的方法.

6) FB-TCP: 提出 4 种基于故障的优先级排序方法以生成用例的初始排名, 并选择基于 Averaging、Kendall Tau 以及 Schulze 这 3 种排名聚合技术来整合 4 种排序方法生成的排名, 其中 Kendall Tau 集成技术在排序上表现更出色.

7) KS-TCP: 一种基于聚类的静态黑盒测试用例优先级排序方法, 先采用 K-medoids 方法捕获用例的相似性进行更精细的划分, 再使用贪婪策略对各个簇进行优先级排序, 并通过轮询方式选择最终的执行顺序.

8) SAT-TCP: 一种基于语义感知的两阶段 TCP 框架, 利用基于信息检索对测试用例进行初步排序和过滤, 并使用预训练的 Siamese 语言模型基于语义相似性对用例进行细粒度排序.

3.4 实验结果与分析

下面通过各实验验证本文方法的优势. 第 3.4.1 节针对本文方法的 2 大模块中的 3 个构件设计了 6 种变体方法进行消融实验, 以回答 RQ1; 第 3.4.2 节将本文方法在 6 个数据集上与 8 种经典的、最新的相关方法进行对比, 详细分析本文方法的优势, 以回答 RQ2; 第 3.4.3 节探讨所提方法的迭代次数是如何划定的, 以回答 RQ3.

3.4.1 消融实验 (RQ1)

为验证粗粒度用例分组和细粒度用例分组排序这 2 大模块中的构件对 TCP 排序性能的影响, 设计 6 种变体方法, 将本文方法 TPG-TCP 与它们进行对比. 其中, ① Ours-1 变体方法不进行粗粒度分组, 直接采用 TP-Additional 策略对全部测试用例排序, 并通过减少迭代次数加快排序, 主要考察本文方法的粗粒度分组模块的作用; ② Ours-2 变体方法在使用 TP-Additional 策略对粗粒度用例分组后得到的关键用例组排序时, 不减少迭代次数, 主要验证本

文方法迭代次数划定 (即, 减少迭代次数) 的有效性; ③ Ours-3 变体方法不采用 TP-Additional 策略, 而采用 Additional 策略对粗粒度分组后的关键用例排序, 并减少迭代次数, 主要阐明本文方法提出的 TP-Additional 策略的合理性; ④ Ours-4 变体方法仅保留粗粒度用例分组模块, 并采用 Additional 策略, 此变体主要说明本文方法的粗粒度用例分组模块与其他构件组合的效果如何; ⑤ Ours-5 变体方法仅在采用 Additional 策略排序时通过减少迭代次数提高排序性能, 不对用例集进行粗粒度分组也不采用 TP-Additional 策略, 此变体主要阐述对迭代次数划定与其他构件组合的有效性; ⑥ Ours-6 变体方法仅使用 TP-Additional 策略排序, 不进行粗粒度用例分组也不减少迭代次数, 此变体主要阐述 TP-Additional 策略与其他构件组合的有效性. 各方法的构件组成情况如表 5 所示.

表 5　变体方法组件构成情况

变体方法	粗粒度用例分组	细粒度用例分组排序		
		迭代次数划定	TP-Additional策略	Additional策略
Ours-1	×	○	○	×
Ours-2	○	×	○	×
Ours-3	○	○	×	○
Ours-4	○	×	×	○
Ours-5	×	○	×	○
Ours-6	×	×	○	×
Ours	○	○	○	×

注: ① “×”表示相应方法不包含该构件, “○”表示相应方法包含该构件; ② Additional策略并非本文方法构件, 在表中列出意在表明, 当变体方法中未采用本文提出的TP-Additional策略排序时, 使用Additional策略替代以指导排序

在 6 种不同规模数据集上进行消融实验, 并使用 *APFD* 和 *TETC* 作为评价指标, 以验证本文方法各构件对排序性能的影响. 结果如表 6 所示.

表 6　消融实验结果对比

评价指标	数据集	变体名称						
		Ours-1	Ours-2	Ours-3	Ours-4	Ours-5	Ours-6	Ours
APFD	Flex	0.9648	0.9698	0.9506	0.9506	0.9506	0.9574	**0.9698**
	Tcas	0.9412	0.9633	0.9224	0.9224	0.9216	0.9351	**0.9633**
	Schedule2	0.9354	0.9477	0.9187	0.9187	0.9187	0.9269	**0.9477**
	Jsoup	0.9461	0.9588	0.9328	0.9328	0.9328	0.9459	**0.9586**
	Jacksoncore	0.9485	0.9532	0.9237	0.9237	0.9237	0.9376	**0.9532**
	Camel-core	0.9613	0.9645	0.9544	0.9544	0.9544	0.9590	**0.9645**
TETC (s)	Flex	0.1064	0.0936	0.0881	0.0944	0.0982	0.1387	**0.0862**
	Tcas	0.1381	0.1236	0.1153	0.1245	0.1262	0.1421	**0.1132**
	Schedule2	0.1082	0.0779	0.0772	0.0867	0.0863	0.1132	**0.0764**
	Jsoup	0.2324	0.2064	0.2062	0.2093	0.2122	0.2768	**0.2058**
	Jacksoncore	0.2865	0.2536	0.2436	0.2563	0.2541	0.3352	**0.2422**
	Camel-core	21.2238	18.2775	17.4281	22.7642	23.6743	25.7614	**15.7693**

注: 本文方法TPG-TCP的数据加粗标示, 为便于比较, 利用下划线突显变体方法中表现最佳的数据

图 2 是表 6 结果的更直观表示, 图 2(a)–(f) 分别表示 TPG-TCP 相比于各变体方法在数据集 Flex、Tcas、Schedule2、Jsoup、Jacksoncore 与 Camel-core 上的性能比较情况.

由表 6 和图 2 可看出, 粗粒度用例分组构件对实验结果的影响最为显著, 未使用粗粒度用例分组的变体方法 Ours-1 在 *APFD* 与 *TETC* 这 2 个指标上的表现不如使用粗粒度用例分组的本文方法 Ours 以及其他变体方法; 迭代次数划定构件会导致实验结果产生明显变化; TP-Additional 策略构件对各方法性能影响较大.

1) 粗粒度用例分组构件对方法效果的整体影响最大 (变体方法 Ours-1). 为高效捕获并利用用例间的潜在关系, TPG-TCP 在进行正式排序前考虑对用例进行粗粒度用例分组, 以减少后期排序中对覆盖信息的冗余访问以及

用例相关性的计算次数. 由图 2 可看出, 与变体方法 Ours-1 相比, 所提方法整体性能提升最大, *APFD* 与 *TETC* 指标分别平均提升 1.06% 和 19.84%. 其中, 在 *TETC* 指标上至少提升 11.45%, 最多提升可达 29.39%. 由此可知, 本文对用例进行粗粒度分组, 可减少在覆盖信息和排序方面的处理时间, 更有效地提升排序方法的效率.

2) 迭代次数划定构件对方法性能的影响明显 (变体方法 Ours-2). 因经过多轮迭代后, 所有可能语句均已被覆盖足够的次数, 若持续迭代, 被覆盖的语句再暴露问题的可能性会大大降低, 再使用精确的策略对剩余用例排序, 必会降低排序效率. TPG-TCP 在已达到足够覆盖次数情况下, 通过迭代次数划定操作将关键用例针对性地分为能更快地暴露问题的用例组和剩余关键用例组, 对于剩余用例使用简单高效的策略进行排序, 以进一步提高排序效率. 由图 2 可看出, 与变体方法 Ours-2 相比, 所提方法在效率方面提升明显, *TETC* 指标平均提升 6.13%, 至少提升 0.29%, 最多提升可达 13.72%. 之所以在 *APFD* 指标上未提升, 是因为迭代次数的划定对于有效性的影响非常小. 由此可知, 通过迭代次数划定对用例进行分组排序可使排序过程更加灵活和高效, 在提高排序性能上作用较大.

3) TP-Additional 策略构件对方法性能影响较大 (变体方法 Ours-3). 针对现有方法仅根据用例覆盖标准的整体状态选择用例, 以及 Additional 策略随机性的选择会降低故障检测有效性这两个问题, TPG-TCP 采用以用例潜力度为二次选择指标的 TP-Additional 策略, 可快速暴露潜在的重要问题. 由图 2 可看出, 与变体方法 Ours-3 相比, 所提方法性能提升较大, *APFD* 与 *TETC* 指标分别平均提升 2.77% 和 2.55%. 其中, 在 *APFD* 指标上至少提升 1.06%, 最多提升可达 4.43%. 由此可知, 采用 TP-Additional 策略在一定程度上减少了 Additional 策略中随机因素的干扰, 可有效地提升排序方法的有效性.

4) 三构件相互配合使得排序方法性能达到最优. 由图 2 可看出, 与仅采用粗粒度用例分组构件的变体方法 Ours-4 相比, 所提方法在 *APFD* 与 *TETC* 指标分别平均提升 2.77% 和 11.26%; 与仅采用迭代次数划定构件的变体方法 Ours-5 相比, 所提方法在 *APFD* 与 *TETC* 指标分别平均提升 2.79% 和 12.51%; 与仅采用 TP-Additional 策略变体方法 Ours-6 相比, 所提方法在 *APFD* 与 *TETC* 指标分别平均提升 1.69% 和 30.48%. 本文方法 TPG-TCP 同时运用粗粒度用例分组、迭代次数划定构件与 TP-Additional 策略这 3 个构件, 其评价指标在各数据集中均有所提升. 由此可知, 这 3 个构件相互配合可使 TPG-TCP 方法排序效果达到最优.

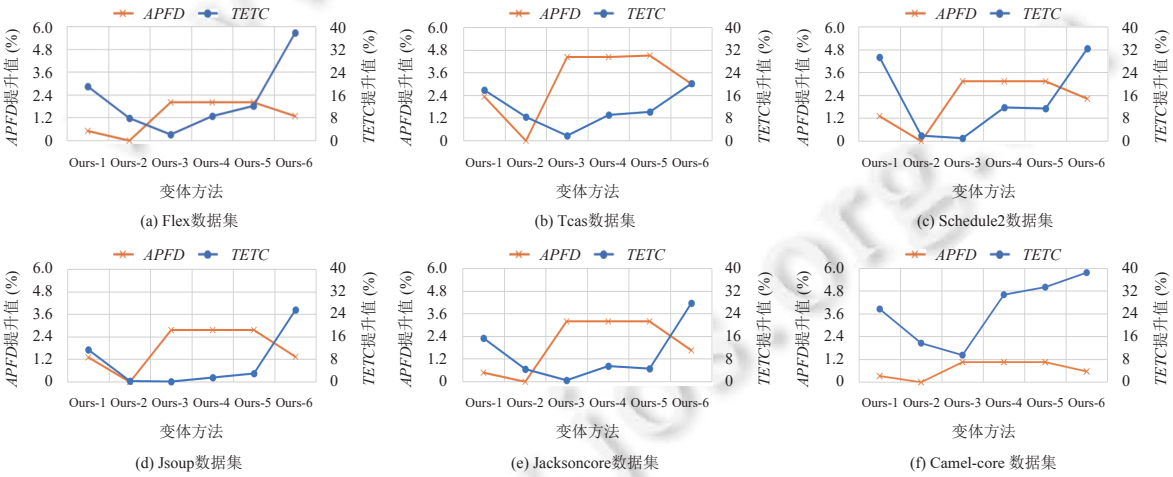

图 2 TPG-TCP 与各变体方法在不同数据集上的性能比较

综上分析可知: ① 与直接对原始用例集进行排序方法相比, 经过粗粒度用例分组后的用例充分利用了用例间的覆盖信息, 使得在后续排序过程中对于用例信息的访问更加高效, 可更快速地优化排序过程. ② 通过迭代次数划定, 可对关键用例集进一步划分, 在保证检测质量的同时, 提升检测效率. ③ TP-Additional 策略在 Additional 策略中加入用例潜力度这个二次排序指标, 可有效缓解 Additional 策略随机性的选择对故障检测有效性造成的影响, 提升排序方法的有效性. ④ 粗粒度用例分组构件、迭代次数划定构件与 TP-Additional 策略构件的组合是合理、有效的, 对于本文方法性能的提升具有非常重要的作用, 尤其在排序效率方面.

3.4.2 对比实验 (RQ2)

在对比时, 本文的迭代次数设置为 5 (迭代次数划定实验见第 3.4.3 节), 而对于含有参数的对比方法, 其参数值均采用对应文献中的原始设定. 就部分重要参数而言, 在 AGA 中, 迭代次数为 10; 在 CGWO-TCP 中, 循环混沌函数中的控制参数 a 为 0.5, b 为 0.2; 在 WOA-TCP 中, 螺旋攻击的控制参数 b 为 1; 在 SAT-TCP 中, 学习率为 2E–5, batch_size 为 68, epoch 为 5.

本文与 8 种相关经典的、最新的方法在 6 个不同规模的数据集上就 *APFD*、*TETC* 指标进行详细对比, 结果如表 7 所示.

表 7 各方法对比结果

指标	数据集	基于贪心策略方法			基于群智能算法		基于故障方法	基于机器学习方法		本方法	性能提升
		GA	OCP	AGA	CGWO-TCP	WOA-TCP	FB-TCP	KS-TCP	SAT-TCP	TPG-TCP	(%)
APFD	Flex	0.9506	0.9569	0.9506	0.9601	0.9542	N/A	N/A	0.9472	**0.9698**	**1.02**
	Tcas	0.9216	0.9341	0.9216	0.9470	N/A	N/A	0.9213	0.9425	**0.9633**	**1.72**
	Schedule2	0.9187	0.9258	0.9187	N/A	0.9302	N/A	0.9096	0.9365	**0.9477**	**1.20**
	Jsoup	0.9328	0.9459	0.9328	N/A	N/A	0.9350	N/A	0.9316	**0.9586**	**1.34**
	Jacksoncore	0.9389	0.9365	0.9389	N/A	N/A	0.9400	N/A	0.9109	**0.9532**	**1.40**
	Camel-core	0.9544	0.9536	0.9544	N/A	N/A	N/A	N/A	0.9390	**0.9645**	**1.06**
TETC (s)	Flex	0.3685	0.1386	0.0982	N/A	2.7287	N/A	N/A	1.6782	**0.0862**	**12.22**
	Tcas	1.0569	0.1415	0.1270	N/A	N/A	N/A	0.7180	2.0547	**0.1132**	**10.86**
	Schedule2	1.5218	0.1120	0.0867	N/A	10.6269	N/A	1.2830	2.7756	**0.0764**	**11.88**
	Jsoup	5.7624	0.2639	0.2145	N/A	N/A	29.0670	N/A	9.4432	**0.2058**	**4.06**
	Jacksoncore	1.4103	0.3402	0.2558	N/A	N/A	46.1960	N/A	3.5758	**0.2422**	**5.32**
	Camel-core	1288.1519	75.4517	18.1040	N/A	N/A	N/A	N/A	486.4658	**15.7693**	**12.09**

注: ① 本文方法TPG-TCP的数据加粗标示, 为便于比较, 利用下划线突显对比方法中表现最佳的数据, 最后一列给出本文方法相对于某一最佳对比方法的性能提升情况 (粗体标示). ② CGWO-TCP、WOA-TCP、FB-TCP、KS-TCP数据来源于各原始文献, 因这些文献中不同方法选择的实验对象不同, 故存在部分数据集数据缺失的情况. 其余方法的数据通过运行50次实验, 并取其平均值, 将所得平均值与对应方法文献中的原实验数据进行比较, 最终实验数据以较优结果为准填入

1) 与基于贪心策略的 TCP 方法相比

① GA 在 TCP 领域是非常经典的方法, 反馈机制的加入使其在排序有效性方面取得不错的效果, 但该方法包含大量重复的数据访问, 极大地降低了其速度. 此外, 该策略存在的随机因素会在一定程度上影响其有效性. 相较于 GA 方法, TPG-TCP 在 *APFD* 和 *TETC* 指标上分别最少提升 1.06% 和 76.61%.

② OCP 通过设定历史优先级指标以避免考虑所有候选测试用例, 并降低 GA 随机因素对有效性的影响. 相较于 OCP 方法, TPG-TCP 在 *APFD* 和 *TETC* 指标上分别最少提升 1.14% 和 20.00%.

③ AGA 相较于 OCP 方法更侧重对 Additional 策略效率的提升, 通过增加存储结构并减少迭代次数共同加快排序速度. 相较于 AGA 方法, TPG-TCP 在 *APFD* 和 *TETC* 指标上分别最少提升 1.06% 和 4.06%.

这 3 种基于贪心策略的方法 GA、OCP 和 AGA 虽然在一定程度上提高了 TCP 排序的有效性和效率, 但未考虑用例间的关系以及用例信息的个体性. 因此, 本文方法 TPG-TCP 采用粗粒度用例分组以及 TP-Additional 策略应对这些问题.

2) 与基于群智能算法的 TCP 方法相比

① CGWO-TCP 将近年来较流行的灰狼优化算法改进后应用于 TCP 领域, 提升了排序的有效性. 相较于 CGWO-TCP 方法, TPG-TCP 在 *APFD* 指标上最少提升 1.02%.

② WOA-TCP 对先进的鲸鱼优化算法改进后, 将其用于解决 TCP 问题, 在减少算法迭代次数的同时提升了排序的有效性. 相较于 WOA-TCP 方法, TPG-TCP 在 *APFD* 和 *TETC* 指标上分别最少提升 1.63% 和 96.84%.

这两种基于群智能算法的方法将较新的优化算法应用于 TCP 领域, 利用群智能算法强大的搜索能力来解决排序问题, 创新性较高. 但此类方法建立的搜索空间规模较大, 导致在搜索时间成本方面与基于贪心策略的方法相比过高. 本文方法 TPG-TCP 采用粗粒度用例分组与细粒度分组排序, 在保证排序有效性的同时提高了排序的效率.

3) 与基于故障的 TCP 方法相比

FB-TCP 侧重于组合多种基于故障的方法以解决 TCP 问题. 相较于 FB-TCP 方法, TPG-TCP 在 *APFD* 和 *TETC* 指标上分别最少提升 1.40% 和 99.29%.

此方法根据排序过程中的多种因素对同一用例集进行排序, 将不同排序结果进行整合, 该方法在有效性方面取得不错的提升. 但该方法在采用多种排序方法时未区分用例, 只是对不同排序方法的结果进行调整组合, 这极大地增加了排序时间. 本文方法 TPG-TCP 利用用例间的关系对用例进行粗粒度划分, 并根据用例重要性采用差异化的排序策略, 较好地提升了方法的效率.

4) 与基于机器学习的 TCP 方法相比

① KS-TCP 通过 K-medoids 聚类挖掘用例间的相似性, 以提高排序有效性. 相较于 KS-TCP 方法, TPG-TCP 在 *APFD* 和 *TETC* 指标上分别最少提升 4.19% 和 84.23%.

② SAT-TCP 将 TCP 排序分为两个阶段, 在将用例进行过滤与排序后, 采用预训练模型捕获用例间的语义相似性, 进一步细化排序, 在有效性方面取得极大提升. 相较于 SAT-TCP 方法, TPG-TCP 在 *APFD* 和 *TETC* 指标上分别最少提升 1.2% 和 93.23%.

这两种基于机器学习的方法均在一定程度上提高了排序的有效性, 但由于聚类方法和预训练过程复杂, 其在时间性能上的提升非常有限. 本文方法 TPG-TCP 采用简单但有效的集合操作以利用用例间的相似性将用例分组, 并在后续排序过程中, 根据用例的重要性采用不同复杂程度的策略排序, 可见 TPG-TCP 方法在效率方面有更好的表现.

综上分析, 相比于其他方法, TPG-TCP 在 Flex、Tcas、Schedule2、Jsoup、Jacksoncore 和 Camel-core 这 6 个数据集上的性能均有所提升. ① 从数据集的角度看: 对于较小规模数据集, 在 Tcas 上, *APFD* 与 *TETC* 指标分别提升 1.72% 和 10.86%; 在 Schedule2 上, *APFD* 与 *TETC* 指标分别提升 1.20% 和 11.88%; 在 Jsoup 上, *APFD* 与 *TETC* 指标分别提升 1.34% 和 4.06%. 对于较大规模数据集, 在 Flex 上, *APFD* 与 *TETC* 指标分别提升 1.02% 和 12.22%; 在 Jacksoncore 上, *APFD* 与 *TETC* 指标分别提升 1.40% 和 5.32%; 在 Camel-core 上, *APFD* 与 *TETC* 指标分别提升 1.06% 和 12.90%. 在较小规模数据集 Tcas、Schedule2、Jsoup 中, 本文方法在 Schedule2 上表现更好; 在较大规模数据集 Flex、Jacksoncore、Camel-core 中, 本文方法在 Camel-core 上表现更好. 因为当代码规模较小时, 测试用例规模对时间成本的影响更大; 当代码规模较大时, 代码规模对时间成本的影响会超过测试用例规模的影响. 可见, TPG-TCP 对于代码规模较大或测试用例规模较大的数据集表现更好, 原因在于, 粗粒度用例分组构件与迭代次数划定构件能减少冗余信息的处理, 可减少在覆盖信息和排序方面的处理时间, 使排序过程更灵活和高效. ② 从指标的角度看: 在 *APFD* 与 *TETC* 指标上, TPG-TCP 较对比方法分别平均提升 1.29% 和 9.54%. 这是因为 TPG-TCP 加入了用例潜力度这个二级指标, 打破了 Additional 策略陷入平局的局面, 在一定程度上提高了排序的有效性, 并将排序过程分为粗细粒度两个阶段, 针对不同重要性的用例采用差异化的策略进行排序, 在保证有效性的同时提高排序效率.

考虑到各方法之间 *APFD* 指标值的差异较小, 并排除随机干扰的影响, 图 3 展示了 TPG-TCP 方法与其他 TCP 方法在 6 个数据集上实验 50 次的 *APFD* 值. 对于 CGWO-TCP、WOA-TCP、FB-TCP 和 KS-TCP 方法, 其数据来源于各原始文献 (见表 7 说明), 故其箱型图的中值即对应文献中的原始数据, 其余方法的数据在实验中未出现误报或漏报情况. 与大多数其他方法相比, 所提方法 TPG-TCP 的 *APFD* 指标表现出相对较高的中值, 这表明 TPG-TCP 在多数情况下更能有效地检测出故障, 准确性和稳定性更高.

此外, 在 6 个数据集上, 我们采用单侧 Wilcoxon 符号秩检验进行统计分析, 以评估 TPG-TCP 方法相对于其他对比方法在 *APFD* 指标上的改进情况. 单侧 Wilcoxon 符号秩检验适用于配对样本的比较, 在我们的实验中, 每一对配对样本来自同一数据集下的 2 种方法的性能结果. Wilcoxon 符号秩检验通过对这些配对样本的差异进行排序并计算符号秩, 来评估 2 种方法之间是否存在显著差异. 我们将 p 值的阈值设置为 0.05, 若 p 值小于该阈值则表明应拒绝原假设, 即本文方法的分布差异显著高于对比方法. 检验结果如表 8 所示. 从检验结果来看, 本文提出的 TPG-TCP 方法均优于其他对比方法, 这也说明本文方法的故障检测有效性更优.

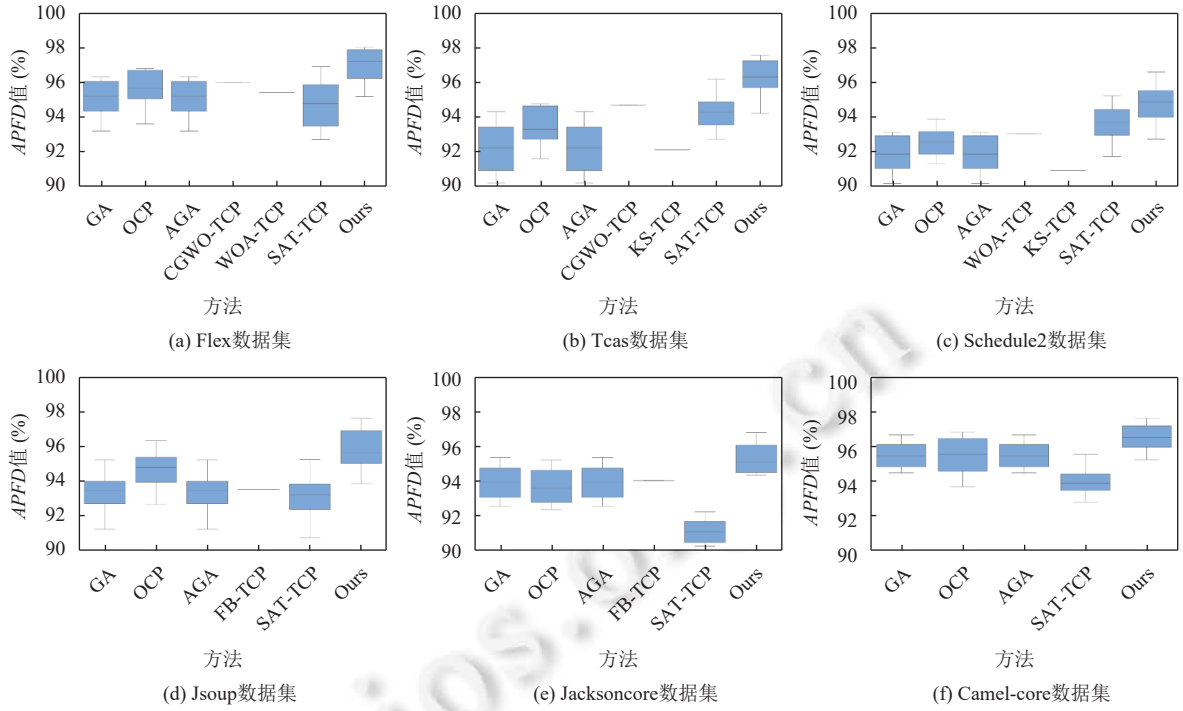

图 3　TPG-TCP 方法与其他对比方法在 6 个数据集上的 *APFD* 值箱线图

表 8　TPG-TCP 方法与其他对比方法在 6 个数据集上的 *p*(*APFD*) 指标值检验结果

数据集	GA	OCP	AGA	CGWO-TCP	WOA-TCP	FB-TCP	KS-TCP	SAT-TCP
Flex	2.31E–11	1.45E–09	2.31E–11	4.58E–08	1.90E–12	—	—	4.96E–11
Tcas	1.78E–15	1.78E–15	1.78E–15	5.86E–14	—	—	1.78E–15	2.63E–12
Schedule2	1.78E–15	9.52E–13	1.78E–15	—	2.49E–14	—	1.78E–15	3.92E–06
Jsoup	1.95E–13	2.10E–06	1.95E–13	—	—	1.78E–15	—	2.49E–14
Jacksoncore	2.31E–11	7.15E–11	2.31E–11	—	—	1.78E–15	—	1.78E–15
Camel-core	3.09E–09	2.42E–06	3.09E–09	—	—	—	—	1.24E–14

3.4.3　迭代次数划定实验 (RQ3)

这里针对如何选取合适的迭代次数展开讨论, 在不同数据集上探究不同迭代次数对方法性能的影响. 将 TPG-TCP 方法应用于 6 个数据集上, 对每个数据集使用 I 次 TP-Additional 策略, 对剩余未被排序的用例使用复杂度更低的 Total 策略排序, 记录迭代次数 I 与这 I 次排序的 *APFD* 和 *TETC* 指标值. 若在某数据集的一次迭代中选择的测试用例数量较多, 则 TPG-TCP 在该数据集的迭代次数就较小. 迭代次数的选取对排序的有效性与效率有不同程度的影响. 选取的迭代次数过大, 排序的时间成本会随之增加; 选取的迭代次数过小, 排序的有效性则会受到影响. 图 4(a)–(f) 分别表示 TPG-TCP 在数据集 Flex、Tcas、Schedule2、Jsoup、Jacksoncore 和 Camel-core 中不同迭代次数的实验结果.

从图 4 可看出, 随着迭代次数的增大, *APFD* 值会逐渐趋于稳定, 而 *TETC* 值在不断上升. 当迭代次数 $I \geqslant 5$ 时, 在 Tcas、Schedule2、Jsoup 以及 Jacksoncore、Camel-core 数据集上的 *APFD* 值不再变化. 而在 Flex 数据集上, 当迭代次数 $I \geqslant 3$ 时, *APFD* 值不再变化. 可见, 当 *APFD* 指标达到某一值时会趋于稳定, 若再增加迭代次数, 则时间成本会不断上升. 因此, 为了保证排序方法的有效性, 同时减少时间成本以提高效率, 将迭代次数划定为 5 最为合适.

3.4.4　复杂度分析

所提方法 TPG-TCP 通过对原始用例集进行粗细粒度分组并采用不同策略实现排序, 与现有方法相比, 排序步

骤相对增多, 尽管复杂度随之增加, 但其综合性能得到一定的提升. 下面对本文方法及部分具有代表性的对比方法就时间复杂度展开分析.

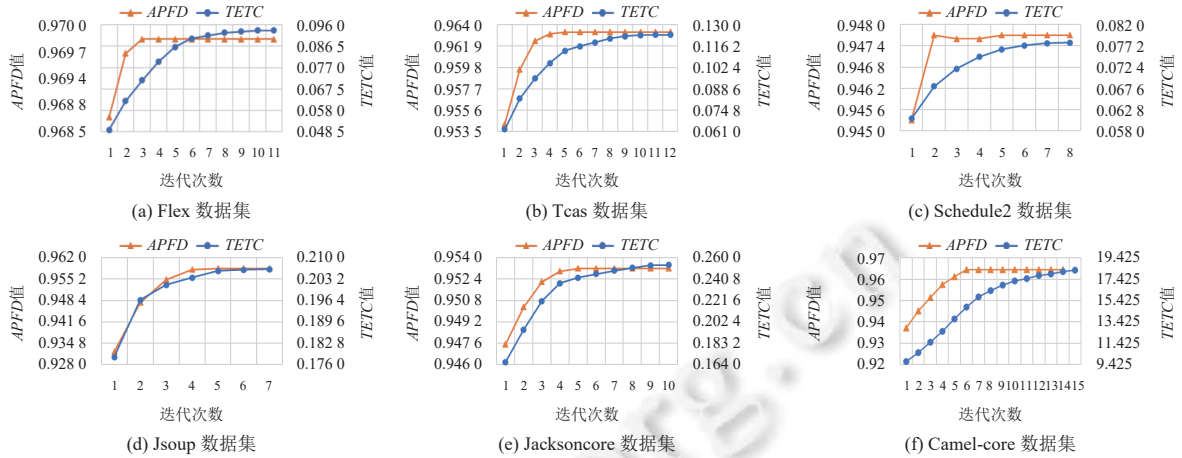

图 4 不同数据集中迭代次数的影响

1) 就本文方法 TPG-TCP 而言, 其时间复杂度主要体现在粗粒度用例分组与细粒度用例分组排序两个部分. 粗粒度用例分组的时间复杂度为 O(N^2M), 其中 N 为测试用例数, M 为覆盖元素的数量; 在细粒度用例分组排序的单轮迭代中, 更新用例-语句数表 TC 的次数等于语句标记表 S_{cov} 中的元素个数, 时间复杂度为 O(NM). 可知, 细粒度用例分组排序的时间复杂度大致为 O(NMI), I 是迭代次数. 由于 N 通常大于 I, 因此 TPG-TCP 的时间复杂度大致为 O(N^2M).

2) 就对比方法 AGA (较新的基于贪心策略的方法, 在对比方法中的排序效率最优, 与本文结构最为相似) 来说, 它的时间主要消耗在每次选择测试用例后的更新操作中, 时间复杂度大致为 O(NMI). 与方法 AGA 相比, TPG-TCP 的复杂度虽有所增加, 但 AGA 仅专注于效率的提升, 并未提高 Additional 策略的有效性, 而 TPG-TCP 在提高效率的同时也对排序策略进行了改进, 提升了排序的有效性.

3) 就对比方法 WOA-TCP (较新的基于群智能算法的方法, 与本文方法同为启发式算法) 来说, 它的时间主要消耗在搜索最优种群个体过程, 时间复杂度大致为 O(PNI), 其中 P 为种群规模, 迭代次数 I 通常为 1 000. 该方法将 TCP 视为 NP-hard 问题, 利用鲸鱼优化算法在用例序列的全排列中搜索最优解, 牺牲了时间复杂度, 但换取了一定的有效性. TPG-TCP 的复杂度与 WOA-TCP 方法相近, 但通过对比实验可知其在整体性能上优于 WOA-TCP 方法.

4) 就对比方法 FB-TCP (最新的基于故障的方法, 与本文方法同采用集成策略) 来说, 它的时间主要消耗在利用各个子方法生成用例序列, 并交换各序列中存在冲突的用例排名以生成最终优先级序列的过程, 时间复杂度大致为 O(N^3F) + O(NF) + O(N^2F) + O($N(N+F)$) + O(2^C), 即 O(N^3F), 其中 F 为故障数, C 为冲突的测试用例对, 最大值为 20. 该方法采用 3 种集成方式对 4 种基于故障方法的结果进行组合排序, 兼顾了排序过程的多种因素, 排序有效性较高, 但在效率上表现较差. TPG-TCP 的复杂度会高于 FB-TCP 方法, 且通过对比实验得知, 在整体性能上也会优于 FB-TCP 方法.

5) 就对比方法 KS-TCP (最新的基于机器学习的方法, 与本文结构较为相似) 来说, 它的时间主要消耗在聚类测试用例的过程, 时间复杂度为 O(NM^2/K), K 为簇的个数. 该方法利用 K-medoids 聚类挖掘用例间的相似性, 在用例相似度计算上耗费了大量时间. 由于通常 $M > N > K$, 故 TPG-TCP 在大多数情况下的复杂度低于 KS-TCP. 另外, 通过对比实验得知, TPG-TCP 在整体性能上相较于 KS-TCP 方法也有所提升.

综上分析可知, 各方法的时间复杂度大小关系为 FB-TCP (或 KS-TCP) > TPG-TCP (或 WOA-TCP) > AGA. 尽管从时间复杂度上来看, 本文方法 TPG-TCP 高于 AGA, 但就排序性能而言, 本文方法更优, 相较于 AGA, 所提方法在 *APFD*、*TETC* 指标上分别最少提升 1.06% 和 4.06%. 相较于 WOA-TCP、FB-TCP 与 KS-TCP 方法, 所提方法 TPG-TCP 的时间复杂度相近或更低, 排序性能均更佳, 在 *APFD* 指标上分别最少提升 1.63%、1.40% 和 4.19%,

在 *TETC* 指标上分别最少提升 96.84%、99.29% 和 84.23%. 可见, 本文方法的设计思路是合理而有意义的.

4 总结与下一步工作

本文提出一种面向两阶段分组的测试用例优先级排序方法 TPG-TCP. 第 1 阶段进行粗粒度测试用例分组, 第 2 阶段进行细粒度测试用例分组排序, 不仅提高了排序的效率, 也提升了有效性.

1) 为减少在覆盖信息和排序处理上的时间消耗, 提出粗粒度用例分组. 利用测试用例间的隐藏关系, 将待测用例集分为关键用例组与普通用例组, 减少对覆盖信息的冗余访问, 降低排序时间成本, 提高排序效率, 并为下一阶段采用多样性策略排序做准备.

2) 为同时提高排序效率与有效性, 采用细粒度用例分组排序. 为减少 Additional 策略中随机因素的干扰, 通过划定迭代次数将关键用例分组, 并提出基于用例潜力度的 TP-Additional 策略对一部分关键用例排序, 同时采用简单高效的 Total 策略对普通用例与另一部分关键用例排序. 这在提高方法排序效率的同时也提升了有效性.

3) 为验证所提方法效果, 展开综合实验分析. 所提方法 TPG-TCP 无论是与经典方法还是最新方法对比, 在 2 个常用 TCP 指标上均有明显提升. 同时通过消融实验, 验证了方法各构件的必要性.

虽然本文所提方法 TPG-TCP 较优, 但依旧存在一些待完善之处, 其中关键用例组的划分条件较为宽松, 未深入考虑用例间覆盖信息的相似程度. 在后续研究中, 我们会尝试探索合适的机器学习方法, 试图在捕获用例覆盖信息相似性的同时, 也能保证方法的高效性.

References

[1] Yu XL, Jia K, Hu WH, Tian J, Xiang JW. Black-box test case prioritization using log analysis and test case diversity. In: Proc. of the 34th IEEE Int'l Symp. on Software Reliability Engineering Workshops (ISSREW). Florence: IEEE, 2023. 186–191. [doi: 10.1109/ISSREW60843.2023.00072]

[2] Lima JAP, Vergilio SR. A multi-armed bandit approach for test case prioritization in continuous integration environments. IEEE Trans. on Software Engineering, 2022, 48(2): 453–465. [doi: 10.1109/TSE.2020.2992428]

[3] Khan A, Azim A, Liscano R, Smith K, Tauseef Q, Seferi G, Chang YK. Machine learning-based test case prioritization using hyperparameter optimization. In: Proc. of the 2024 IEEE/ACM Int'l Conf. on Automation of Software Test (AST). Lisbon: IEEE, 2024. 125–135.

[4] Felding E, Strandberg PE, Quttineh NH, Afzal W. Resource constrained test case prioritization with simulated annealing in an industrial context. In: Proc. of the 39th ACM/SIGAPP Symp. on Applied Computing. Avila: ACM, 2024. 1694–1701. [doi: 10.1145/3605098.3635971]

[5] Fan SP, Wan L, Yao NM, Zhang Y, Ma BY. Test case sorting method based on key use cases extracted. Acta Electronica Sinica, 2022, 50(1): 149–156 (in Chinese with English abstract). [doi: 10.12263/DZXB.20201284]

[6] Elbaum S, Malishevsky AG, Rothermel G. Test case prioritization: A family of empirical studies. IEEE Trans. on Software Engineering, 2002, 28(2): 159–182. [doi: 10.1109/32.988497]

[7] Rothermel G, Untch RH, Chu CY, Harrold MJ. Test case prioritization: An empirical study. In: Proc. of the 1999 IEEE Int'l Conf. on Software Maintenance (ICSM). Oxford: IEEE, 1999. 179–188. [doi: 10.1109/ICSM.1999.792604]

[8] Li F, Zhou JY, Li YZ, Hao D, Zhang L. AGA: An accelerated greedy additional algorithm for test case prioritization. IEEE Trans. on Software Engineering, 2022, 48(12): 5102–5119. [doi: 10.1109/TSE.2021.3137929]

[9] Zhao YF, Hao D. Test case prioritization technique in continuous integration based on reinforcement learning. Ruan Jian Xue Bao/Journal of Software, 2023, 34(6): 2708–2726 (in Chinese with English abstract). https://www.jos.org.cn/1000-9825/6506.htm [doi: 10.13328/j.cnki.jos.006506]

[10] Miranda B, Cruciani E, Verdecchia R, Bertolino A. FAST approaches to scalable similarity-based test case prioritization. In: Proc. of the 40th IEEE/ACM Int'l Conf. on Software Engineering. Gothenburg: IEEE, 2018. 222–232. [doi: 10.1145/3180155.3180210]

[11] Zhang QJ, Fang CR, Sun WS, Yu SC, Xu YT, Liu YL. Test case prioritization using partial attention. Journal of Systems and Software, 2022, 192: 111419. [doi: 10.1016/j.jss.2022.111419]

[12] Wang XL, Zhang SL. Cluster-based adaptive test case prioritization. Information and Software Technology, 2024, 165: 107339. [doi: 10.1016/j.infsof.2023.107339]

[13] Chen JF, Gu YC, Cai SH, Chen HB, Chen JY. A novel test case prioritization approach for black-box testing based on K-medoids clustering. Journal of Software: Evolution and Process, 2024, 36(4): e2565. [doi: 10.1002/smr.2565]

[14] Li YL, Wang ZA, Wang JJ, Chen J, Mou R, Li GB. Semantic-aware two-phase test case prioritization for continuous integration. Software Testing, Verification and Reliability, 2024, 34(1): e1864. [doi: 10.1002/stvr.1864]

[15] Biswas S, Bansal A, Mitra P, Mall R. Fault-based regression test case prioritization. IEEE Trans. on Reliability, 2023, 72(3): 1176–1190. [doi: 10.1109/TR.2022.3205483]

[16] Nayak G, Barisal SK, Ray M. CGWO: An improved grey wolf optimization technique for test case prioritization. Programming and Computer Software, 2023, 49(8): 942–953. [doi: 10.1134/S0361768823080169]

[17] Yang B, Li HL, Xing Y, Zeng FP, Qian CD, Shen YZ, Wang JB. Directed search based on improved whale optimization algorithm for test case prioritization. Int'l Journal of Computers Communications & Control, 2023, 18(2): 5049. [doi: 10.15837/ijccc.2023.2.5049]

[18] Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering, 2005, 10(4): 405–435. [doi: 10.1007/s10664-005-3861-2]

[19] Bajaj A, Sangwan OP, Abraham A. Improved novel bat algorithm for test case prioritization and minimization. Soft Computing, 2022, 26(22): 12393–12419. [doi: 10.1007/s00500-022-07121-9]

[20] Just R, Jalali D, Ernst MD. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proc. of the 2014 Int'l Symp. on Software Testing and Analysis. San Jose: ACM, 2014. 437–440. [doi: 10.1145/2610384.2628055]

[21] Wang HR, Cui ZQ, Yue L, Chen X, Zhang LW. Software fault localization based on redundant coverage information reduction. Acta Electronica Sinica, 2024, 52(1): 324–337 (in Chinese with English abstract). [doi: 10.12263/DZXB.20220820]

[22] Coles H, Laurent T, Henard C, Papadakis M, Ventresque A. PIT: A practical mutation testing tool for Java (demo). In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. Saarbrücken: ACM, 2016. 449–452. [doi: 10.1145/2931037.2948707]

[23] Papadakis M, Jia Y, Harman M, Le Traon Y. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: Proc. of the 37th IEEE/ACM IEEE Int'l Conf. on Software Engineering. Florence: IEEE, 2015. 936–946. [doi: 10.1109/ICSE.2015.103]

附中文参考文献

[5] 范书平, 万里, 姚念民, 张岩, 马宝英. 基于关键用例获取的测试用例排序方法. 电子学报, 2022, 50(1): 149–156. [doi: 10.12263/DZXB.20201284]

[9] 赵逸凡, 郝丹. 一种基于强化学习的持续集成环境中测试用例排序技术. 软件学报, 2023, 34(6): 2708–2726. https://www.jos.org.cn/1000-9825/6506.htm [doi: 10.13328/j.cnki.jos.006506]

[21] 王浩仁, 崔展齐, 岳雷, 陈翔, 郑丽伟. 基于冗余覆盖信息约简的软件缺陷定位方法. 电子学报, 2024, 52(1): 324–337. [doi: 10.12263/DZXB.20220820]

作者简介

钱忠胜, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为软件工程, 机器学习, 智能化软件.

秦朗悦, 硕士生, 主要研究领域为软件工程, 机器学习.

范赋宇, 硕士生, 主要研究领域为软件工程, 机器学习.

付庭峰, 硕士生, 主要研究领域为软件工程, 机器学习.