软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(6):2708−2726 [doi: 10.13328/j.cnki.jos.006506]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种基于强化学习的持续集成环境中测试用例排序技术*

赵逸凡[1,2], 郝 丹[1,2]

[1](高可信软件技术教育部重点实验室 (北京大学), 北京 100871)
[2](北京大学 计算机学院, 北京 100871)
通信作者: 郝丹, E-mail: haodan@pku.edu.cn

摘 要: 在软件交付越来越强调迅速、可靠的当下, 持续集成成为一项备受关注的技术. 开发人员不断将工作副本集成到代码主干完成软件演化, 每次集成会通过自动构建测试来验证代码更新是否引入错误. 但随着软件规模的增大, 测试用例集包含的测试用例越来越多, 测试用例的覆盖范围、检错效果等特征也随着集成周期的延长而变化, 传统的测试用例排序技术难以适用. 基于强化学习的测试排序技术可以根据测试反馈动态调整排序策略, 但现有的相关技术不能综合考虑测试用例集中的信息进行排序, 这限制了它们的性能. 提出一种新的基于强化学习的持续集成环境中测试用例排序方法——指针排序方法: 方法使用测试用例的历史信息等特征作为输入, 在每个集成周期中, 智能体利用指针注意力机制获得对所有备选测试用例的关注程度, 由此得到排序结果, 并从测试执行的反馈得到策略更新的方向, 在“排序-运行测试-反馈”的过程中不断调整排序策略, 最终达到良好的排序性能. 在 5 个规模较大的数据集上验证了所提方法的效果, 并探究了使用的历史信息长度对方法性能的影响, 方法在仅含回归测试用例的数据集上的排序效果, 以及方法的执行效率. 最后, 得到如下结论: (1) 与现有方法相比, 指针排序方法能够随着软件版本的演化调整排序策略, 在持续集成环境下有效地提升测试序列的检错能力. (2) 指针排序方法对输入的历史信息长度有较好的鲁棒性, 少量的历史信息即可使其达到最优效果. (3) 指针排序方法能够很好地处理回归测试用例和新增测试用例. (4) 指针排序方法的时间开销不大, 结合其更好、更稳定的排序性能, 可以认为指针排序方法是一个非常有竞争力的方法.
关键词: 持续集成; 测试用例排序; 强化学习
中图法分类号: TP311

中文引用格式: 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. http://www.jos.org.cn/1000-9825/6506.htm
英文引用格式: 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). http://www.jos.org.cn/1000-9825/6506.htm

Test Case Prioritization Technique in Continuous Integration Based on Reinforcement Learning

ZHAO Yi-Fan[1,2], HAO Dan[1,2]

[1](Key Laboratory of High Confidence Software Technologies of Ministry of Education (Peking University), Beijing 100871, China)
[2](School of Computer Science, Peking University, Beijing 100871, China)

Abstract: As software delivery increasingly emphasizes speed and reliability, continuous integration (CI) has attracted more and more attention these years. Developers continue to integrate working copies into the mainline to realize software evolution. Each integration involves automated tests to verify whether the update introduces faults. However, as the scale of software increases, test suites contain more and more test cases. As software evolves, the coverage and fault detection ability of test cases also change among different CI cycles. As a result, the traditional test case prioritization techniques may be inapplicable. Techniques based on reinforcement learning can

* 基金项目: 国家自然科学基金 (61872008)
收稿时间: 2021-07-13; 修改时间: 2021-09-08; 采用时间: 2021-10-08; jos 在线出版时间: 2022-11-16
CNKI 网络首发时间: 2022-11-18

adjust prioritization strategies dynamically according to test feedback. But the existing techniques based on reinforcement learning proposed in recent years do not comprehensively consider information in the test suite during prioritization, which limits their effectiveness. This study proposes a new test case prioritization method in CI, called pointer ranking method. The method uses features like history information of test cases as inputs. In each CI cycle, the agent uses the attention mechanism to gain attention to all candidate test cases, and then obtains a prioritization result. After test execution, it obtains the updating direction from the feedback. It constantly adjusts its prioritization strategy in the process “prioritization, test execution, test feedback” and finally achieves satisfied prioritization performance. This study verifies the effectiveness of the proposed method on five large-scale datasets, and explores the impact of history length on method performance. Besides, it explores the model’s effectiveness on datasets which only contain regression test cases and the model’s execution efficiency. Finally, the study comes to the following conclusions. First, compared to existing techniques, pointer ranking method can adjust its strategy along with the evolution of the software, and effectively enhance the fault detection ability of test sequence in CI. Second, pointer ranking method has good robustness to history length. A small amount of history information can make it achieve the optimal performance. Third, pointer ranking method can handle regression test cases and newly-added test cases well. Finally, pointer ranking method has little time overhead. Considering its better and more stable prioritization performance, pointer ranking method is a very competitive method.

Key words: continuous integration (CI); test case prioritization; reinforcement learning

1 引 言

随着工业软件交付向着敏捷、持续的趋势发展, 为了使得软件迭代更加迅速且可靠, 工业界提出了持续集成方法. 在持续集成中, 软件项目的现有状态称为“主线”, 为了多人同时开发以提高软件交付的速度, 开发人员通过代码管理系统将软件项目拷贝到本地的开发机器上, 这份拷贝称为“工作副本”. 通过对工作副本代码的更新, 开发人员为软件引入新功能或修复旧版本的错误, 并与此同时添加或改变测试用例, 以及时检测改动后的软件功能是否正常, 在编译、链接、运行并通过所有自动化测试后, 变动被集成到“主线”, 该项目被重新构建, 完成一个持续集成周期. 持续集成需要开发者不断提交更新代码, 将工作副本与共享主体部分代码合并, 通过不断提交小的修改并将其整合到主目录中, 完成新功能的添加, 高频率的提交使得软件得以更快地进行版本的更新迭代, 并能够通过软件测试及时发现软件演化过程中引入的新错误, 降低积累错误的风险和修复错误的成本. 近年来, Travis CI[1]、Circle CI[2]、Jenkins[3]、Buildbot[4]等多个持续集成平台发展迅速, 成千上万的软件使用持续集成作为软件交付方式, 逾 60 万个用户已选择在 Travis CI 上进行项目测试[1], 并从其快速建立、及时测试的特性之中受益, 公司如 Google、Facebook、Microsoft 和 Amazon 也使用了持续集成来更好地匹配他们的开发速度和规模[5].

然而, 持续集成也面临着挑战, 其最受关注、最重要的特性就是为开发者提供快速的反馈, 但实际应用中系统构建和测试运行可能耗费大量的时间和资源, 这与人们所期望的相悖. 据报道, 亚马逊工程师每天进行 13.6 万次系统部署, 平均每 12 s 进行一次[5]. 在 Google, “开发人员必须等待 45 分钟至 9 个小时才能接收测试结果”[6]. 如前所述, 每个持续集成周期涉及多个任务, 包括版本控制, 软件配置管理, 新软件候选版本的自动构建和回归测试, 在企业应用中, 通常持续集成的时间瓶颈在于回归测试[7], 特别是涉及到外部服务如数据库的测试. 回归测试是持续集成尤为重要的一环, 通过运行确定的测试用例集, 观察软件能否通过全部测试用例, 可以确保更新的代码没有引入新的错误, 且没有影响到原先软件功能. 但越来越大的测试用例集规模对回归测试的时间成本和资源成本提出了更高的要求, 同时, 在代码不断迭代更新的条件下, 持续集成中的回归测试可获取的信息和测试用例特性也不同于传统的回归测试, 新环境下的回归测试为研究人员提出了新的挑战. 为了解决该问题, 持续集成中的测试用例排序技术被提出, 其基本思想是对测试用例集中的元素进行排序, 先执行潜在可能揭示错误的、用时更短的测试用例, 后执行不会揭示错误的、用时较长的测试用例, 及时暴露软件的缺陷, 给予开发者最快的反馈.

在持续集成环境中, 由于开发人员的频繁提交, 软件代码在不断发生改变, 传统的白盒测试用例排序技术[8−11]需要使用的测试用例语句覆盖信息、代码依赖关系等需要频繁更新, 而近年来一些新的测试用例排序技术也需要用到这些信息[12−15], 这需要消耗大量资源和时间, 不能满足持续集成对快速反馈的需求. 除此以外, 每个集成周期中可能存在测试用例的增删, 一些测试系统过时特性的测试用例可能被删去, 而一些测试新特性或被改变特性的

© 中国科学院软件研究所 http://www.jos.org.cn

测试用例可能被加入, 单纯依靠历史执行结果的测试用例排序方法不能够捕捉到代码上下文的变化, 因而不能够很好地进行适应性调整. 同时, 随着集成周期数增加, 测试用例的重要性也会发生变化, 非适应性算法不能够捕捉到测试用例重要程度随时间的变化. 综上, 一些传统的测试用例排序技术不能很好地捕捉软件代码和测试用例的变化, 不能合理地利用持续集成中测试用例的历史执行信息 (包括历史执行结果、历史执行时长) 等特征, 因而不适用于持续集成环境, 为此, 新的持续集成测试用例排序算法亟待开发. 近年来, 有不少针对持续集成测试用例排序的系统性研究[16,17].

机器学习算法通过构建模型, 完全自动化地生成排序, 也被应用于测试用例排序任务中[18−20], 而强化学习是机器学习方法的一个分支, 学习者在与环境的交互中做出序列决策, 选择动作并得到奖励指引, 然后根据执行结果不断调整自己的策略, 强化得到奖励的行为以获得最大效益. 持续集成中每个集成周期都需要确定测试用例的执行次序, 且由于代码改变、测试用例增删, 不同集成周期的测试特性也会发生变化, 强化学习技术能够解决持续集成测试用例排序这个序列决策问题, 也适用于持续集成这个环境不断发生变化的场景[18,21−23]. Spieker 等人提出了第一个基于强化学习的方法 RETECS[18], RETECS 方法构建的模型根据单个测试用例特征, 计算并输出其优先级, 根据该测试用例的执行结果及最终排序位次, 计算得到该测试用例的奖励信号, 据此调整模型. 本文提出了一种新的基于强化学习的持续集成环境中测试用例排序方法——指针排序方法, 该方法构建的模型将测试用例集中所有用例的特征一起输入模型, 模型在赋予某个测试用例优先级时, 使用注意力机制, 综合考虑其他所有测试用例的特征, 捕捉测试用例间的相对关系并据此排序. 强化学习通过奖励信号激励模型改进策略, 通过给予智能体合适的反馈信号, 引导模型朝着最大化预期目标的方向前进, 为了与输入相匹配, 本文还提出了两种新的奖励函数, 它们是对测试用例集整体排序效果的评价, 为整个测试用例集而非单个测试用例的排序结果定义激励信号, 引导模型实现更好的排序. 本文在 5 个数据集 (Paint Control、IOF/ROL、GSDTSR、Bcel、Nfe) 上将本方法与 RETECS 方法进行效果比较, 并探究了历史信息长度等输入信息对本方法的表现影响, 本方法在回归测试用例数据集上的表现效果, 以及本方法的执行效率.

本文的主要贡献如下.

(1) 提出了一种新的用于持续集成环境中的测试用例排序方法, 该方法综合考虑测试用例集中所有用例信息进行排序, 并使用了评价测试用例集的整体排序效果的奖励函数, 直接用全局目标激励模型, 避免排序陷入局部最优.

(2) 在 5 个较大规模的实际数据集上验证技术的有效性, 验证了提出的方法具有良好的测试用例排序性能, 在其中 4 个数据集上, 所提方法的平均表现优于现有方法, 在全部数据集上, 所提方法在最坏情况下的排序效果远好于现有方法.

(3) 探究了影响指针排序方法效果的因素: 历史信息长度、新增测试用例. 得到结论: 指针排序方法受历史信息长度影响较小, 能够很好地处理新增测试用例和回归测试用例.

(4) 探究了指针排序方法的执行效率. 得到结论: 指针排序方法具有较小时间开销的同时, 有优秀的排序性能, 是一个具有竞争力的排序方法.

本文第 1 节介绍了本文的研究背景、研究目的及研究内容. 第 2 节简要介绍了测试用例排序的相关工作. 第 3 节介绍了本文提出的指针排序方法的原理, 所构建模型各个部分的结构和计算公式. 第 4 节介绍了本文提出的 4 个研究问题、实验设置、实验结果与分析、以及实验有效性威胁. 第 5 节对本文工作进行总结和讨论, 阐述本文的研究意义, 展望未来工作.

2 相关工作

测试用例优先排序最先由 Wong 等人在 1997 年提出[24], 即对给定的测试用例集, 将最有可能揭示错误的测试用例排到前面最先执行, 从而减少开发人员得到测试反馈的时间, 提升开发效率. Total 和 Additional 策略是两种基于贪心算法的传统排序技术[9], 它们的提出是基于假设: 覆盖程序语句数多的测试用例检错能力强. 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, 而 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, 据此, 多种在语句粒度、分支粒度、方法粒度上应用 Total 或 Additional 策略的技术[8−10,25,26]被设计出来. 基于搜索的排序技术最早由 Li 等人提出[27], 这种技术是通过在测试用例的所有排序序列构成的解空间中, 使用遗传排序算法等进行搜索, 寻找到较优的排序序列. 此外, 利用整数线性规划的测试用例排序技术[28]亦被提出, 这种技术利用线性规划求解得到满足约束的测试用例子集, 然后在该子集上继续进行排序. 然而, Elbaum 等人[29]提出传统的测试用例排序技术需要依赖代码插桩, 来获得信息如测试用例覆盖语句等, 需要耗费大量时间, 只适用于离散的、完整的测试用例集, 而持续集成下的测试用例排序则更为频繁和连续, 代码插桩等收集数据的方式的时间开销大太, 不足以迅速地应对高频率的代码变动.

持续集成环境下的测试用例选择与排序研究近年来相当活跃, Yu 等人[30]通过分析逾 1 000 个 GitHub 项目的变动提交, 来探究测试用例选择技术应该如何在持续集成环境下使用, 同时, 多个可在持续集成环境中应用的测试用例排序技术已被提出[18,29,31−35], 理想的可应用于持续集成环境的回归测试技术需要同时满足可靠性和省时性, 主要可分为两类: 启发式方法[36,37]以及完全自动化的机器学习算法[18−23]. Busjaeger 等人利用机器学习算法提出了一个测试用例排序技术[19], 并在大规模的持续集成环境中进行了实验, 他们提取出有用的测试用例特征, 并用 SVM 分类器为测试用例赋予优先级, 然而他们的技术在训练后得到的是一个固定的策略, 不能够随集成环境中代码变动进行排序策略的调整. Bertolino 等人[20]提供了一个全面的比较机器学习算法在持续集成测试用例优先级排序任务中的效果的研究, 他们首先通过构建依赖图的方式选择出与代码改动相关的测试用例, 并且在此过程中提取代码变动和测试用例特性的量化指标, 然后使用 10 种不同的机器学习算法, 将这些选择出的测试用例按照重要性从高到低进行排序, 运行测试并比较算法表现的差异, 据研究显示, 强化学习算法的转移熵更小, 排序效果对于代码变动更具有鲁棒性, 在高度动态的环境中相对于静态的机器学习算法更具优势. Spieker 等人[18]首先将强化学习技术应用于持续集成中的测试用例排序问题, 提出了一种在持续集成中自动学习测试用例优先级排序的新方法 RETECS, 其根据测试用例的历史执行信息、预估执行时长来选择测试案例并确定其优先级, 通过智能体与环境的交互, 以预先设定的奖励函数激励智能体, 使其做出有益达到设定目标的动作. 每个集成周期中, 智能体每次获得单个测试用例的特征, 输出其对应优先级, 最后根据所有测试用例的优先级进行排序. Spieker 等人[18]提出了 3 种奖励函数: failure count reward, test case failure reward 和 time-ranked reward, 并在 3 个工业级程序上进行了实验. He 等人[21]则基于上述工作, 对持续集成测试中强化学习方法的奖励函数和奖励策略进行了研究, 在奖励函数方面, 使用测试用例的完整历史执行信息, 提出了两种新的强化学习奖励函数 APHF 和 HFC, 其中 APHF 函数考虑了测试用例的历史失效分布信息, HFC 函数通过计算测试用例的历史失效总次数, 给予失效次数多的测试用例更多奖励, 而在奖励策略方面, He 等人[21]提出了整体奖励和部分奖励两种策略, 根据测试用例执行结果决定是否给予智能体反馈. Yang 等人[22]进行了持续集成测试中强化学习奖励函数的系统研究, 探究了 APHF、HFC 和 TF 这 3 种奖励函数的效果, 并提出使用时间窗计算奖励以提升计算效率, 此外他们还提出了一种新的奖励策略——模糊奖励策略. 以上文献提出的强化学习算法, 对于每个测试用例仅依据它们各自的特征预测测试用例的优先级, 却忽视了在排序过程中选择测试用例排序位次时, 其他测试用例的特征也是需要考虑的.

3 基于强化学习的指针排序方法

3.1 方法概述

强化学习通过构建智能体, 利用其做出动作决策, 其在不断与环境进行交互的过程中获得奖励, 并根据奖励进行策略的调整[38]: 如果执行某个行为获得正的奖赏, 那么智能体产生这个行为的趋势会增强; 如果执行某个行为获得负的奖赏, 那么智能体产生这个行为的趋势会减弱. 智能体通过“试错”并获得环境反馈的方式, 学习到最大化奖励的策略, 这里的奖励信号起到引导智能体的作用, 是对智能体行为的评价, 而并非直接告诉其正确的行为是什么.

在持续集成场景中, 智能体不断做出决策, 即对测试用例集进行排序, 不同于传统的测试用例排序技术, 其不需要设立可能与最终目标不符的中间目标, 如通过最大化测试用例覆盖范围、最大化测试用例多样性来达到最终

© 中国科学院软件研究所 http://www.jos.org.cn

目标[39]: 尽可能多地揭示错误, 而是利用每个集成周期测试用例集执行的结果计算奖励, 据此进行策略的动态调整并应用于下一个周期, 因此可以直接优化最终目标, 从这个维度避免了陷入局部最优的可能, 有助于算法达到最优. 同时, 传统测试用例排序技术仅通过计算某个指标来指引排序, 不能根据环境变化调整策略, 而强化学习中的智能体能够不断得到测试反馈, 这有利于智能体感知环境的变化 (如测试用例的重要性、开发者的关注点变化), 随着软件演化适应性地调整策略. 实际应用中的流程如图 1 所示, 开始时随机初始化强化学习策略, 然后输入测试用例集, 模型基于测试用例特征计算各测试用例的优先级, 然后依据优先级对测试用例进行排序, 接着依据排序结果进行测试调度和执行, 测试结果给予强化学习模型以反馈, 模型依据反馈动态调整自己的策略, 以最大化预设定的奖励函数.

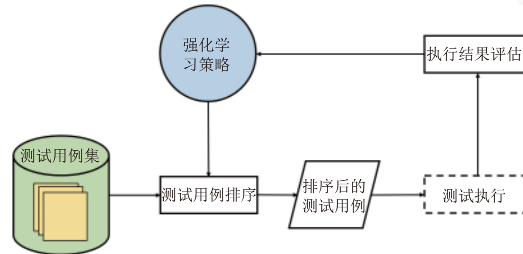

图 1 基于强化学习的持续集成测试[18]

如图 1 所示, 指针排序算法会根据每个周期的排序结果反馈, 调整自己的排序策略, 在这个过程中反映了软件代码和测试用例的变化. 现有的算法大多在排序时已经形成了固定的排序策略, 不会再随集成周期变动, 在过去相当长时间后, 输入相同的测试用例特征仍然会得到同样的优先级. 而因为同一个测试用例在不同集成周期中测试的代码不同, 测试用例也有可能修改, 因此测试用例的特征和执行结果可能发生变化, 优先级也就会发生改变. 指针排序方法根据每一个周期的反馈动态调整自己的策略, 目前周期排序策略的形成是由之前所有周期的实验数据训练所得, 训练数据中包含了执行历史中距当前时间近的测试用例特征及执行结果, 由于相近周期之间的代码变动不大, 这些数据可以用于预测当前测试用例的执行结果. 本文工作的排序策略是随着集成周期延长不断改变的, 由于不断有新的测试用例数据添加, 因此反映了代码和测试用例的变化.

对于每个集成周期的测试, 模型的输入是测试用例集中所有测试用例的信息, 包含其历史执行结果、预估执行时长, 模型的输出是测试用例的一个排序序列, 按照该排序执行测试用例应能够尽快检测出软件中的错误, 即揭错能力更强的测试用例先被执行, 以及时给予开发人员有关软件缺陷的反馈.

我们想让模型学习到策略, 使得给定测试用例集 s 中所有测试用例, 其选择最优的 (能在最短时间内揭示最多错误的) 测试用例排序 π 的概率 $p(\pi|s)$ 为最大. 对于任意一个排序, 根据链式法则, 有公式 (1):

$$p(\pi|s)=\prod_{i=1}^{n}p(\pi(i)|\pi(<i),s) \tag{1}$$

其中, n 为测试用例集中的测试用例数, $\pi(i)$ 为排序 π 中排在第 i 个位置的测试用例, $\pi(<i)$ 为排序 π 中排在第 i 个位置之前的所有测试用例.

获得某个测试用例集的最优排序序列是一个 NP 难问题[40], 但由上分析, 可以将获得序列的过程看成一个连续的决策过程, 问题的关键在于如何从已获得的序列推测出下一个应该选择的测试用例. 这个序列到序列的问题可以使用编码器-解码器结构的机器翻译模型解决, 然而, 通常的编码器-解码器模型只能处理输出类别数量固定的问题, 在测试用例排序任务中, 输出序列中的元素是与输入序列中的元素 (即测试用例) 相关的离散标识, 输出类别数量随输入序列长度变化而变化, Pointer Networks[41]首先被提出用于解决输出类别数量可变的问题, 如旅行商问题, 但是模型需要给定最优序列进行监督学习, 而后 Bello 等人[42]提出使用强化学习方法改进 Pointer Networks 的训练方法, 从而不需要人为给定最优序列, 通过设计奖励信号, 使得模型可以自己学习到如何找到较优解, 并且经过训练可以泛化解决不同规模的组合优化问题. 本文将 Pointer Networks 应用到测试用例排序任务上,

© 中国科学院软件研究所 http://www.jos.org.cn

并且对模型做出相应调整, 使其能够给任意大小的测试用例集进行排序, 并在与环境交互的过程中得到预先设定的奖励信息.

训练的目标是给定测试用例集 S, 最大化其排序的 NAPFD 值, 该值被广泛应用于衡量技术的排序性能[43], 越大说明技术的排序效果越好, 序列检错能力越强, 关于 NAPFD 值的详细介绍见第 3.3 节. 于是训练目标计算公式如下:

$$L(\theta|s)=\mathbb{E}_{\pi\sim p_\theta(\cdot|s)}N(\pi|s) \tag{2}$$

其中, $N(\pi|s)$ 为排序 π 的 NAPFD 值, θ 为模型参数.

为了优化模型参数 θ, 本文使用强化学习中的策略梯度算法和随机梯度下降法让模型在与环境的交互中更新参数, 公式 (2) 的梯度为:

$$\nabla_\theta L(\theta|s)=\mathbb{E}_{\pi\sim p_\theta(\cdot|s)}[N(\pi|s)\nabla_\theta \log p_\theta(\pi|s)] \tag{3}$$

在梯度更新过程中, 由于随机初始化和随机梯度下降会引入噪声, 可以在计算期望时减去一个基准: 这个基准只与 s 相关, 这样的好处是减小方差, 使得模型的训练过程更加稳定. 于是梯度可以使用公式 (4) 计算:

$$\nabla_\theta L(\theta|s)=\mathbb{E}_{\pi\sim p_\theta(\cdot|s)}[(N(\pi|s)-b(s))\nabla_\theta \log p_\theta(\pi|s)] \tag{4}$$

其中, p_θ 是以 θ 作为参数的模型策略, $b(s)$ 为基准.

对于这个基准, 可以选择该测试用例集的排序评价期望值 $\mathbb{E}_{\pi\sim p_\theta(\cdot|s)}N(\pi|s)$, 一个最简易的估计方式是利用已得数据计算指数移动平均值, 然而使用移动平均值不能够很好地对每个测试用例集进行区分, 于是我们引入一个辅助的评价者网络, 利用其对测试用例集 s 的 NAPFD 预测值作为基准, 评价者网络的输入是测试用例集, 输出为估计的评价期望值 $b_{\theta_v}(s)$, 它的损失函数为估计的期望值与真实值的评价值之间的均方误差, 如下:

$$\mathcal{L}(\theta_v)=\|b_{\theta_v}(s)-N(\pi|s)\|_2^2 \tag{5}$$

由此, 本文提出的模型结构如图 2 所示.

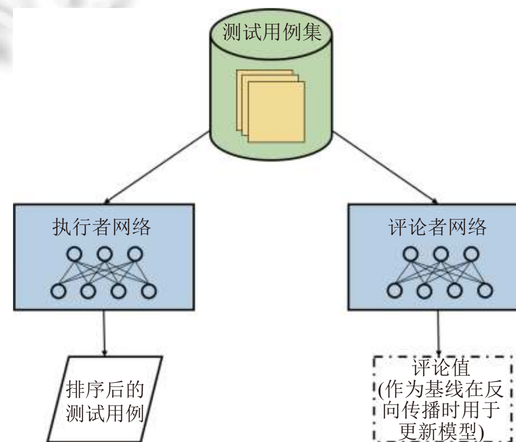

图 2 指针排序模型

模型使用强化学习的优势动作评论算法 (advantage actor critic, A2C)[44], 测试用例集为观察对象, 执行者网络 (actor network) 根据观察到的数据, 产生对应的“动作”, 即对测试用例进行排序, 而评论者网络 (critic network) 则根据观察到的测试用例产生一个评论值, 这个评论值作为基线用于更新执行者网络以减小方差.

在软件的每一次集成中, 执行者网络和评论者网络都会得到此时测试用例集的信息, 即集合中每个测试用例的特征向量 (由其预测执行时长、上一次执行时间、历史执行结果组成), 经过神经网络的计算, 执行者网络得到对测试用例的排序, 评论者网络则根据测试用例信息计算基线评论值, 在执行该排序得到奖励信息, 也即得到排序结果的反馈 (直接使用排序的评价指标作为奖励函数) 后, 两个网络分别据此更新自己的参数, 然后进行下一次的排序与更新. 如此, 随着软件的演化, 网络模型能够不断通过排序结果反馈来更新自己的参数, 对策略进行动态调

© 中国科学院软件研究所 http://www.jos.org.cn

整, 适应新的代码环境.

3.2 模型具体设计

3.2.1 执行者网络

执行者网络用来产生对于测试用例的排序, 如图 3 所示, 其主要由编码器和解码器两个部分组成, 编码器为 Bi-LSTM 模型, 解码器为 LSTM 模型. 测试用例的特征向量 (包含其预估执行时长, 上一次执行时间以及历史执行结果) 首先通过一个嵌入层, 得到其隐层表示 $V=\{v_1,v_2,\ldots,v_n\}$, 其中 v_i 为第 i 个测试用例的嵌入向量, 然后在第 i 步将 v_i 输入编码器进行编码, 直到测试用例集中的 n 个测试用例输入完成. 在解码时, 首先输入一个开始向量 $<g>$, 在之后每一步中使用指针注意力机制产生选择下一个测试用例的概率分布, 然后指向概率最大的测试用例, 当这个测试用例被选择后, 其嵌入向量作为解码器的下一步输入, 如此逐步得到模型对测试用例的下一步选择, 最终输出对测试用例集的排序结果.

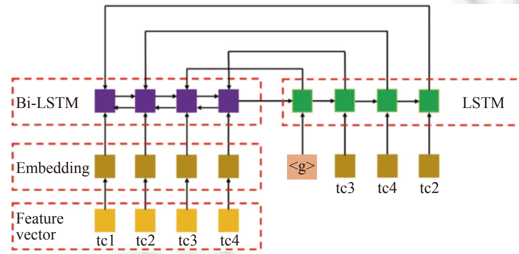

图 3 执行者网络

在测试用例排序场景下, 测试用例的输入顺序理应不影响排序结果, 因而本文使用 Bi-LSTM 作为模型的编码器. 如图 4 所示, 在 Bi-LSTM 中, 每一个时间步, 隐藏状态由前向和后向两个方向的 LSTM 所得结果拼接起来, 也即 $h_t=\{\overrightarrow{h_t},\overleftarrow{h_t}\}$, 这样使得每个时间步能够同时得到输入的过去和未来状态信息, 减少了输入顺序对结果造成的影响, 最终得到每一步的编码 $\{h_i\}_{i=1}^n$.

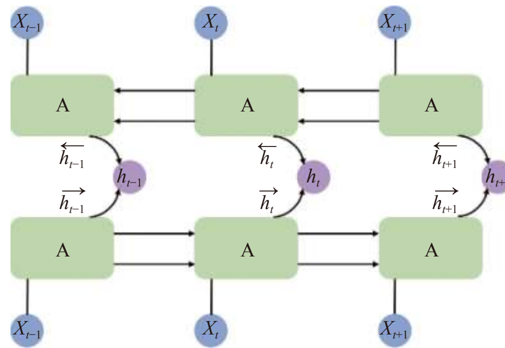

图 4 Bi-LSTM

传统的序列到序列模型的输出的类别数量是固定的, 其取决于设计的输出类别数量. 在测试用例排序这个应用场景下, 因为测试用例集中的测试用例数量是不固定的, 对于不同的输入序列长度, 传统的序列到序列模型都需要进行重新训练, 这不利于实际应用: 本文希望模型的输出类别数量是可变的, 且直接依赖于输入序列长度. 为了解决这个问题, 本文利用指针注意力机制, 将注意力当作指针, 对输入序列直接输出一个概率分布, 指示每个输入序列的元素在该时刻出现的概率, 这样输出类别数量即直接取决于输入序列的长度, 模型就能够完成对不同序列长度的样本的求解.

具体地, 指针注意力机制可以计算对于某个查询向量 q, 键向量 $\{k_i\}_{i=1}^n$ 的注意力分布, 于是可以将注意力权值当作指针, 计算出的权重大的键需要得到更多注意, 也即指示了该时刻该键对应的输入应该在该位次出现的概率

更大, 对于查询向量 q, 键向量 k_i 的注意力权重计算公式如下[42]:

$$u_i = \begin{cases} v^{\top} \cdot \tanh(W_{key} \cdot k_i + W_q \cdot q), & \text{if } i \neq \pi(j) \text{ for all } j < i \\ -\infty, & \text{otherwise} \end{cases} \quad \text{for } i = 1, 2, \ldots, k \tag{6}$$

$$A(key, q; W_{key}, W_q, v) = Softmax(u) \tag{7}$$

其中, W_{key} 、 W_q 为可训练的权重矩阵, v 为可训练的参数向量, u_i 表示了查询向量 q 对键向量 k_i 的关注程度, $Softmax$ 函数归一化向量 u 得到对各个查询向量的注意力 A. 因此, 选择排在第 j 位的测试用例时, 每个测试用例被选择的概率即为以第 j 步解码器的输出作查询向量, 以编码器每一步输出作键向量 (与每一个输入相对应) 的注意力权值, 计算公式如下:

$$p(\pi(j)|\pi(<j), s) = A(enc_{1:n}, dec_j) \tag{8}$$

其中, enc_i 为编码器的第 i 步输出, dec_j 为解码器的第 j 步输出, p 为各个测试用例在第 j 步排序时被选择的概率.

3.2.2 评论者网络

如图 5 所示, 评论者网络由编码器、处理模块和解码器 3 部分组成, 编码器是 Bi-LSTM, 处理模块利用注意力机制对测试用例特征进行综合计算, 解码器是全连接的神经网络 (一个两层的 ReLU 神经网络). 与执行者网络相同, 测试用例特征向量首先要经过编码器的编码生成隐藏状态序列, 然后通过数轮处理模块的计算, 在每轮的处理模块计算中, 通过计算编码器各步输出的隐藏状态的注意力权重, 然后根据注意力权重计算隐藏状态的加权线性组合, 更新隐藏状态, 最终的处理结果能够很好地包含相关的重要输入信息, 处理模块的计算公式如下 (注意力权值计算公式同公式 (6)、公式 (7))[42]:

$$p = A(key, q; W^g_{key}, W^g_q, v^g) \tag{9}$$

$$G(key, q; W^g_{key}, W^g_q, v^g) = \sum_{i=1}^{k} k_i p_i \tag{10}$$

$$g_0 = q \tag{11}$$

$$g_l = G(key, g_{l-1}; W^g_{key}, W^g_q, v^g) \tag{12}$$

其中, W_{key} 、 W_q 为可训练的权重矩阵, v^g 为可训练的参数向量, G 是键向量使用注意力权值加权后的线性组合, l 为重复处理的轮数, 上一轮处理结果 g_{l-1} 可以作为下一轮处理过程的查询向量, 第 1 轮的 g_0 即为查询向量 q, 该过程可以进行数轮以更好地综合输入信息. 经过数个处理过程后, g_l 被送入两层维度分别是 d 和 1 的神经网络中解码产生评论值 (也即基线预测值), 该评论值与真实值求均方误差后作为损失函数用于优化网络, 更新网络参数使评论者网络预测的期望值更准确.

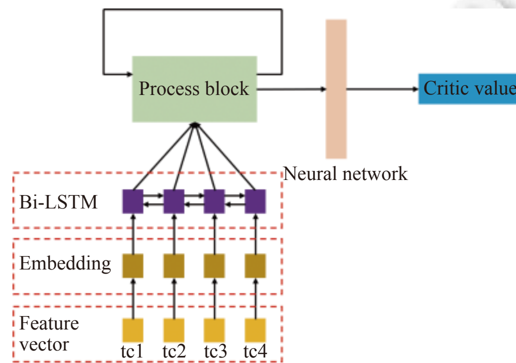

图 5 评论者网络

3.3 奖励函数设计

为了给予模型合适的反馈, 引导其产生更好的排序, 需要使用合理的奖励函数, 在排序后的测试用例集得到执

© 中国科学院软件研究所 http://www.jos.org.cn

行后, 模型依据测试执行结果, 强化得到奖励的行为, 规避得到惩罚的行为, 利用梯度下降算法 (见公式 (4)), 逐渐学习到有利于揭示错误的排序策略. 不同于 RETECS[18]给予每个测试用例奖励, 凸显每个特定的测试用例对于揭示错误的作用, 本文使用的奖励函数以排序后的测试用例集作为整体给予奖励, 这样的好处是能视测试调度为整体, 因为测试用例在不同测试用例集中可能有不同的优先级, 仅根据特征给予单个测试用例奖励可能会忽视其相对其他测试用例揭错能力的高低, 并且, 直接以测试用例集作整体给予奖励能够以排序的整体效果作为奖励信息, 直接优化最终目标, 于是本文提出使用如下两个函数计算奖励.

定义 1. NAPFD[43].

$$NAPFD(TS_i)=p-\frac{\sum_{t\in TS_i^{\text{fail}}} rank(t)}{|TS_i^{\text{fail}}|\times|TS_i|}+\frac{p}{2\times|TS_i|} \tag{13}$$

其中, $p=\frac{|TS_i^{\text{fail}}|}{|TS_i^{\text{total,fail}}|}$, $rank(t)$ 为测试用例 t 所在位次, TS_i^{fail} 为测试用例集中能够揭示错误的测试用例子集, 因为测试用例不是全部被执行, 存在没有被揭示的错误, 因此公式中包含比例 p 进行了标准化.

对于测试用例的一个排序, 当揭错的测试用例被排到前面, 通过的测试用例被排到后面, 我们就认为这个序列能够在执行更少测试用例的条件下, 发现更多的错误, NAPFD 就是据此衡量序列检错性能的一个指标, 本文使用 NAPFD 作为奖励函数. 排序并调度后的测试用例集能够揭示的错误越多, 揭错的测试用例被排在越前面, NAPFD 值就越大, 表明测试用例的排序越成功, 该奖励信息就会给予模型更积极的反馈.

定义 2. NAPFDc.

$$NAPFDc(TS_i)=\frac{\sum_{i=1}^{m}\left(\sum_{j=TF_i}^{n} t_j-\frac{1}{2}t_{TF_i}\right)}{m\sum_{j=1}^{n} t_j} \tag{14}$$

其中, n 为执行的测试用例数, m 为执行的测试用例中揭示错误的用例数, t_i 为测试用例 i 的实际执行时长, TF_i 为排序后的测试用例集中第 i 个揭错的测试用例.

如前所述, NAPFD 的计算仅考虑了揭错测试用例所排位次, 并没有考虑测试用例执行的时长, 在实际情况中, 能够在越短时间内揭示出错误, 开发者就能更快地得到反馈. 于是类比于 APFD 与 APFDc 的关系, 本文对 NAPFD 做一个外推提出 NAPFDc. 相较于 NAPFD 只考虑揭错测试用例的排序位置, NAPFDc 的计算则进一步考虑了测试用例的执行代价, 因为对于两个均能揭错的测试用例而言, 先执行用时更短的可以用迅速地揭示出错误, 揭错效率更高. 整体上, 在越短的时间内执行越多的揭错测试用例, 该排序的 NAPFDc 值就越大, 给予模型的奖励就越多. NAPFDc 相较 NAPFD 包含了执行时长, 能够更好地指示出排序的时间代价, 更有方向性地激励模型.

3.4 算法描述

算法 1. 决策过程.

For each cycle:

获得该集成周期内的环境状态, 即 n 个测试用例的特征向量 $\{f_{tc_1}, f_{tc_2}, \ldots, f_{tc_n}\}$

环境状态输入强化学习模型, 输出个体动作, 即对所有测试用例进行排序

(根据排序结果执行测试)

根据排序结果、测试执行结果, 计算该排序对应的奖励

奖励反馈给强化学习模型, 模型利用梯度下降算法更新参数

决策过程的伪代码描述如上. 对于每个集成周期, 模型首先获得测试用例的特征作为环境状态, 将其输入模型, 通过神经网络计算对测试用例进行排序 (对应于强化学习中的“个体动作”, 具体过程详见第 3.2.1 节), 得到排

© 中国科学院软件研究所 http://www.jos.org.cn

序结果, 按照排序执行测试用例后, 根据排序结果以及每个测试用例的执行结果 (通过或失败), 按照公式 (13) 或公式 (14) 计算奖励, 该奖励用于模型的反向传播以更新参数, 于是模型获得了新的排序策略. 在每个周期中, 模型都会使用最新的数据进行训练, 以确保排序策略不断调整.

4 实验验证

4.1 研究问题

本文提出的研究问题如下.

(1) 指针排序方法是否能够有效地提升测试用例序列检错能力.

(2) 不同的测试用例历史信息长度对指针排序方法效果的影响.

(3) 指针排序方法在仅含回归测试用例的数据集上的排序效果.

(4) 指针排序方法的执行效率如何.

第 1 个研究问题是通过指针排序方法与 RETECS 的排序效果的比较, 研究指针排序方法在工业数据集上的性能表现, 探究其是否能够很好地提升测试用例序列的检错能力, 达到尽早揭示错误的效果, 第 2 个研究问题是通过改变输入的历史信息长度, 探究该因素对方法的效果有何影响. 第 3 个研究问题是探究去除集成周期中新增的无历史执行信息的测试用例后, 排序有历史信息的回归测试用例, 方法表现如何, 这也能间接验证指针排序方法能否很好地处理新增测试用例. 第 4 个研究问题是探究指针排序方法的执行效率, 由于持续集成环境对测试时间有较强的限制, 而机器学习方法需要一定时间进行训练与预测, 因此需要探究方法的执行效率表现, 以确保方法的可用性.

4.2 实验设置

(1) 数据集: 为验证排序方法在实际场景上的可用性, 实验在 3 个工业数据集 (Paint Control, IOF/ROL, GSDTSR[18]) 和两个开源项目数据集 (Bcel, Nfe) 上进行, 其中 Paint Control 和 IOF/ROL 来自 ABB Robotics Norway, 是复杂的工业机器人的测试数据, GSDTSR 是 Google 公司的产品测试数据, Bcel 和 Nfe 是从 GitHub 上收集的高星数项目的测试数据, 这些数据集是从实际的测试过程中得到的, 包含了测试用例的历史执行结果、预估执行时长、上一次执行时刻、当前集成周期的执行结果, 每个数据集包含了超过 300 个集成周期的数据, 具体统计数据如表 1 所示.

表 1 工业数据集的统计数据

数据集	测试用例数	集成周期数	执行结果数	失败率 (%)	执行结果数/测试用例数	执行结果数/集成周期数
Paint Control	89	352	25594	19.36	287.57	72.71
IOF/ROL	1941	320	32260	28.79	16.62	100.81
GSDTSR	5555	336	1260618	0.25	226.93	3751.83
Bcel	60	308	3114	2.67	51.90	10.11
Nfe	235	318	20090	10.03	85.49	63.18

表 1 中, “测试用例数”表示在所有集成周期中所出现的不同的测试用例的总数, “集成周期数”表示代码经过了多少个周期的集成和测试, “执行结果数”表示所有测试用例在全部集成周期中执行的总次数, “失败率”表示在所有测试执行结果中失败的测试所占比例. 从统计数据可以看出, Paint Control、IOF/ROL、Bcel、Nfe 数据集的规模相对较小, 总的测试结果数相较 GSDTSR 少, 但它们的失败率高, 能够揭错的测试用例多, 其中 IOF/ROL 的失败率最高, 达 28.79%. 此外, 观察执行结果数/测试用例数的值, 可以得知, Paint Control 和 GSDTSR 中每个测试用例被执行的平均次数更多, 这表明在这两个数据集中测试用例的历史执行信息更为丰富, 而在 IOF/ROL、Bcel、Nfe 数据集中每个测试用例被执行的平均次数更少, 这表明在该数据集中, 测试用例的历史执行信息相对较少. 从执行结果数/集成周期数的值可以看出, Paint Control、IOF/ROL、Bcel 和 Nfe 数据集在每个集成周期中需要执行

的测试数量较少, 而 GSDTSR 数据集在每个集成周期中需要执行的测试数量较多, 也即需要排序的测试用例数更多, 考虑到 GSDTSR 相对更少的失败率, 其对于排序技术的考验更大, 是一个更具有挑战性的任务. 由以上分析知, 这 5 个数据集的差异性大, 在项目规模、失败率、测试用例历史执行信息多少、集成所需执行的测试数量方面均有差异, 因此实验结论具有普遍意义. 为了减少算法随机性造成的影响, 我们在 Paint Control、IOF/ROL、Bcel、Nfe 数据集上进行了 30 次实验取其平均值, 而在 GSDTSR 上进行 3 次实验取其平均值 (由于 GSDTSR 的测试用例数太多, 有百万级的执行结果数, 平均每个周期需要执行的测试用例数达 3000 以上, 受限于实验时间和计算资源, 我们重复实验 3 次).

(2) 对比技术: RETECS[18]是最早提出的将强化学习应用于持续集成环境测试用例排序的算法, 其也是目前在工业数据集上表现效果最好的强化学习方法[23]. 与本文提出方法不同的是, RETECS 仅使用浅层全连接神经网络, 根据单个测试用例的特征估计其失败概率, 然后将失败概率作为排序优先级进行排序, 其设计的奖励函数也是针对单个测试用例给予反馈, 而没有给予关于测试用例集的整体排序效果的反馈.

(3) 参数设置: 参照文献 [18], RETECS 算法的隐藏层神经元数为 12, 使用的奖励函数为表现效果最佳的 TF 奖励函数. 指针排序方法中, 使用不同参数组合在 IOF/ROL 数据集上的实验结果如表 2 (使用 NAPFDc 作为奖励函数). 由于时间所限, 我们对于各个参数进行了局部探索, 各个参数的探索范围为, 嵌入层维度{4, 8, 16}, 隐藏层维度{16, 32, 64}, Bi-LSTM 层数{1, 2}, 处理模块轮数{1, 2, 3}, 学习率{0.001, 0.0001, 0.00001}, 由实验结果, 我们设定模型的嵌入层维度为 8, 隐藏层维度为 16, 编码器的 Bi-LSTM 层数为 2, 解码器的 Bi-LSTM 层数为 1, 处理模块轮数为 1, 学习率为 0.0001.

表 2 不同参数设置的结果

嵌入层维度	隐藏层维度	Bi-LSTM层数	处理模块轮数	学习率	NAPFD平均值
8	16	2	1	1E–4	0.410
4	16	2	1	1E–4	0.383
16	16	2	1	1E–4	0.385
8	32	2	1	1E–4	0.387
8	64	2	1	1E–4	0.374
8	16	1	1	1E–4	0.372
8	16	2	2	1E–4	0.384
8	16	2	3	1E–4	0.383
8	16	2	1	1E–3	0.384
8	16	2	1	1E–5	0.382

已有的研究[18]在实验中模拟实际的持续集成测试情况: 较强的时间限制不一定能够允许所有的测试用例得到执行, 因此作者限制每个集成周期只有总测试用例执行时长的一半用于测试调度, 只有排在总执行时长前半段的测试用例才能够得到执行. 本文参照这样的做法进行实验.

4.3 评价指标

本文使用 NAPFD 作为评价指标, 且假设每个测试用例揭示的错误不同, 计算公式见公式 (13). APFD 首先被提出[45], 用于衡量测试用例排序技术的效果, 它通过揭错的测试用例在最终排序结果中出现的位次来衡量排序的质量, 揭错用例排序越靠前, 其值越大, 代表排序越有效, 但其假设的是测试用例能够全部得到执行, 不适用于只有一部分测试用例被选择执行的情况. 而 NAPFD 则是 APFD 指标的一个拓展, 它包含了执行部分测试用例检测到的错误占所有错误的比例, 因此适用于只有一部分测试用例能够得到执行的情况.

此外, 本文还使用了 TTF (test to fail) 值作为另一个衡量指标, 其意义是排序后的测试用例集中第 1 个揭错测试用例所在的位次, 测试用例排序的目的就是尽早揭示错误, 如果开发人员及时得到了此次集成中有错误的反馈 (取决于第 1 个揭错测试用例的位次), 那么就会更早地对代码进行深入的检查修改. 第 1 个揭错测试用例排的越靠前, TTF 值就越小, 表明排序效果越好.

© 中国科学院软件研究所 http://www.jos.org.cn

为了验证排序的效果, 召回率 (recall) 也是一个重要的衡量指标, 它衡量的是检测出的错误 (即排在总执行时长前半段的揭错测试用例) 占所有错误的比例, 召回率越大, 证明测试出的错误就越多, 表明排序效果越好.

4.4 实验结果与分析

4.4.1 对研究问题 1 的分析

图 6、图 7 分别为 NAPFDc 奖励函数和 NAPFD 奖励函数的实验结果, 均将指针排序方法与 RETECS 方法做了比较. 图中横坐标表示持续集成的周期数, 纵坐标表示在每个周期方法排序结果的 NAPFD 值, 蓝色曲线、红色曲线分别展示了指针排序方法 (pointer ranking method, PRM) 和 RETECS 的排序效果随集成周期数增加的变化, 黑色实线和虚线分别是蓝色、红色曲线的一次拟合, 指示了 NAPFD 值随周期数变化的趋势, 也表示了模型的学习趋势. 为了展现方法真实的排序效果, 测试用例集中只有一个测试用例的集成周期, 其结果没有被画在图中, 因为此时不需要进行排序. 在实验中, 我们使用 NAPFD 作为奖励函数评价指标, 是为了更好地展示排序效果随集成周期推移的变化情况: 若使用 NAPFDc 作为衡量指标, 由于其计算考虑了测试用例的执行代价, 而每个集成周期包含执行时长各异的测试用例, 且同一个测试用例在不同集成周期中执行时长也会有差异, 因而不能够公平地比较每个集成周期的排序序列检错效果, 而 NAPFD 值的计算关注的是揭错测试用例的排序位次, 这在各个集成周期中是可比较的, 因此我们使用 NAPFD 作为衡量指标, 比较模型在不同集成周期的排序效果.

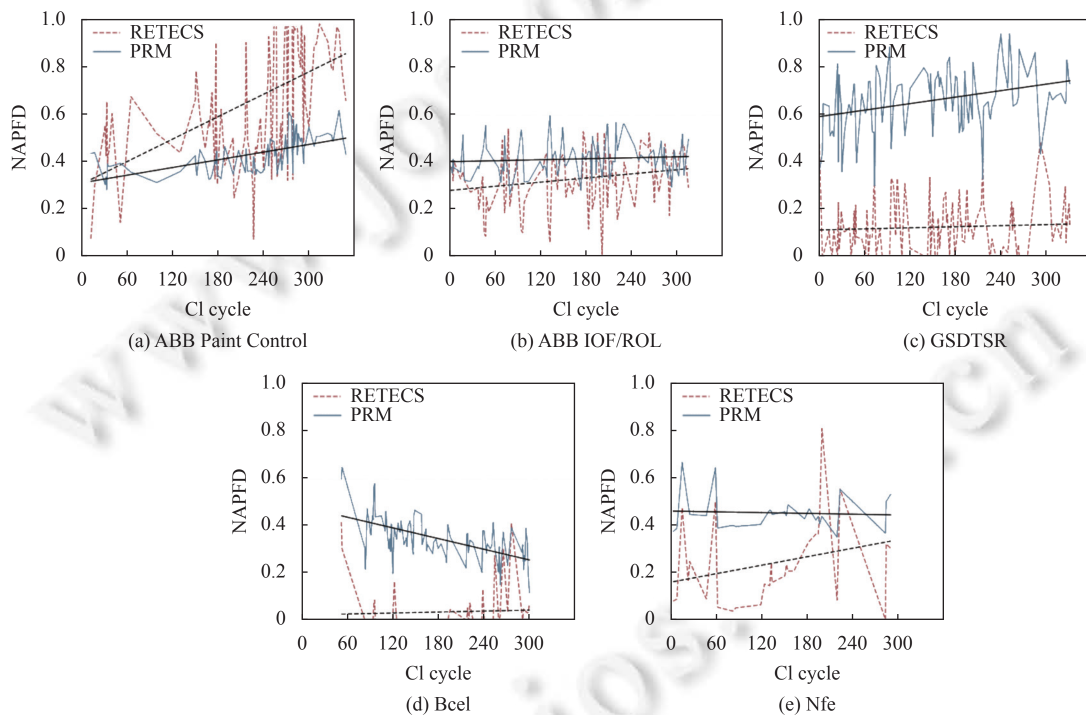

图 6 使用 NAPFDc 作奖励函数的指针排序方法与 RETECS 实验结果比较

在 5 个数据集上, 观察学习曲线, 从图 6 和图 7 的排序方法效果的对比可以看出, 选择 NAPFDc 作为奖励函数对模型的激励效果更好, 因为其考虑了测试用例的执行代价, 相对于单纯考虑测试用例排序位次的 NAPFD 来说, 其包含了测试用例的执行时长, 能够指示模型尽可能将执行时间短的测试用例排在前面, 而这在有时间限制的测试用例排序场景下是较为重要的. 在图 6 中, 模型初始时没有学习到对测试用例进行排序的模式, 因此表现效果较差, 随着集成周期数的增加, 模型根据排序结果反馈调整策略, 学习到排序的模式和如何应对环境的变化, 排序效果逐渐提升, 最终达到了一个较好的排序效果, 这证明了强化学习技术可以在变化的环境中学习到排序模式, 并进行策略的适应性调整. 在图 7 中, 使用 NAPFD 作为奖励信息时, 指针排序方法在 Paint Control、GSDTSR、Nfe、

Bcel 数据集上表现效果逐渐下降, 这可能是由模型的欠拟合或是奖励信息不够明确, 对模型更新产生误导所致, 因为方法的特殊性, 模型是将整个测试用例集作为一个样本, 在每个集成周期结束时才获得一次反馈并更新一次参数, 这使得模型得到的奖励较为稀疏, 在如此稀疏的奖励引导下学习, 奖励函数的设计便至关重要.

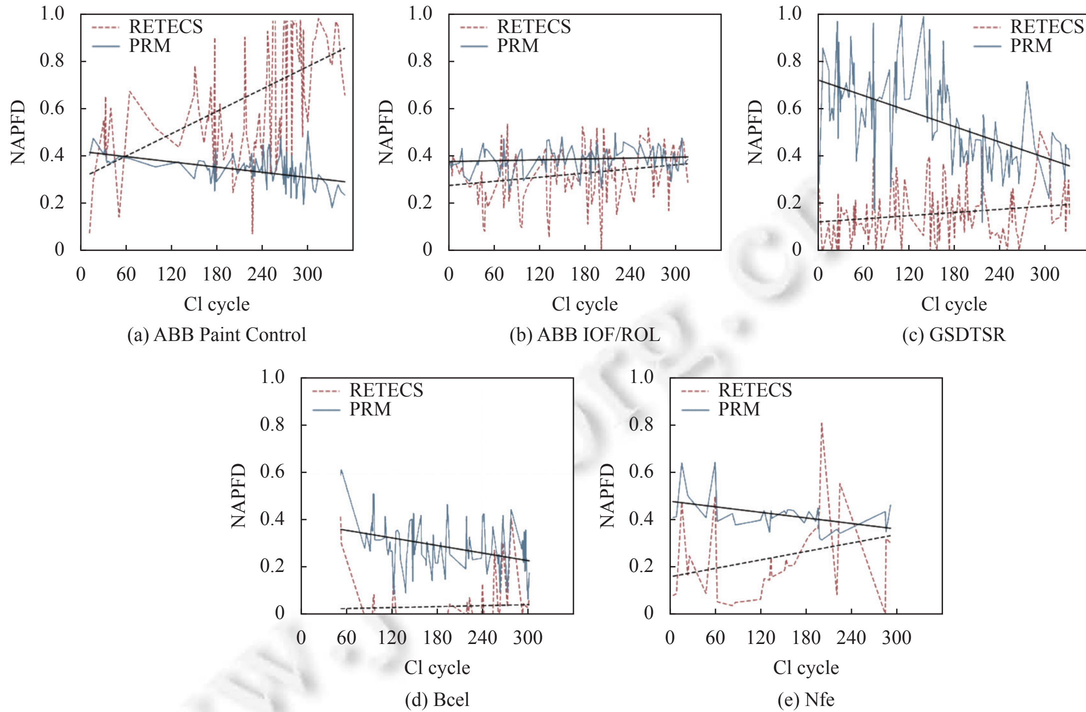

图 7　使用 NAPFD 作奖励函数的指针排序方法与 RETECS 实验结果比较

RETECS 算法在 Paint Control、IOF/ROL、GSDTSR、Bcel、Nfe 这 5 个数据集上的平均 NAPFD 值分别为 0.6535、0.3254、0.1214、0.0331、0.2418, 而指针排序方法在这 5 个数据集上的平均 NAPFD 值分别为 0.4289、0.4105、0.6603、0.3448、0.4535, 由图 6 和 NAPFD 平均值知, 在 IOF/ROL、GSDTSR、Bcel、Nfe 这 4 个数据集上, 使用 NAPFDc 作为奖励函数的指针排序方法的表现好于 RETECS, 不仅其 NAPFD 平均值更高, 且其在大多数集成周期上的表现也更好, 由拟合的直线可看出其良好的学习趋势. 从算法稳定性来看, RETECS 算法在 Paint Control、IOF/ROL、GSDTSR、Bcel、Nfe 这 5 个数据集上的 NAPFD 值的方差分别为 0.2611、0.1141、0.1141、0.0849、0.1798, 指针排序方法在这 5 个数据集上的 NAPFD 值的方差分别为 0.0805、0.0709、0.1293、0.0939、0.0706, 由此可知指针排序方法在所有数据集上表现稳定. 在 Paint Control 数据集上, 指针排序方法表现略差, 平均 NAPFD 值较低, 但 NAPFD 值的方差更小, 排序效果更为稳定, 由剧烈抖动的曲线和较大的方差可知, RETECS 的表现效果很不稳定, 偶尔会达到很差的排序效果, RETECS 算法在 Paint Control、IOF/ROL、GSDTSR、Bcel、Nfe 这 5 个数据集上的最差 NAPFD 值分别为 0.0707、0、0、0、0.0015, 指针排序方法在这 5 个数据集上的最差 NAPFD 值分别为 0.2804、0.2611、0.2965、0.1147、0.3502, 由此可知, 在最坏情况下, RETECS 在 5 个数据集上的排序效果微乎其微, 在原有测试用例集包含揭错测试用例的情况下, 经过排序、选择、执行的测试用例集甚至有可能检测不出错误, 而指针排序方法在最坏情况下仍能够进行较为合理的排序, 将揭示错误的部分测试用例识别出来并前排, 排序效果尚可. 方法的稳定性在实际应用中也是至关重要的, 一个在某些情况下表现极差的排序算法可能为后续的开发埋下隐患, 因而指针排序方法可能在实际使用中更有价值.

表 3、表 4 展示了各个算法在的 TTF 值和召回率的比较, 加粗的为最优值. 由表 3 知, 除了 Paint Control 和 Bcel, 在其余数据集上指针排序方法均有更小的平均 TTF 值, 特别是在 GSDTSR 数据集上, 指针排序方法与 RETECS

© 中国科学院软件研究所 http://www.jos.org.cn

方法的表现差距十分显著, 这表明指针排序方法能够有效地将第 1 个揭错的测试用例排序位次提前, 使得开发者能够更早地发现错误. 由表 4 知, 在除 Paint Control 外的所有数据集上, 指针排序方法均有更高的召回率, 表明在有时间限制的情况下, 指针排序方法能够检测出更多的错误, 而在 Paint Control 数据集上, 则是 RETECS 方法能够检测出更多的错误.

表 3　不同方法的 TTF 平均值比较

方法	Paint Control	IOF/ROL	GSDTSR	Bcel	Nfe
RETECS	**3.2052**	9.4358	665.5383	**1.1494**	31.5184
PRM (NAPFDc)	6.1885	**3.7954**	**24.4486**	1.7438	**7.3589**

表 4　不同方法的召回率平均值比较

方法	Paint Control	IOF/ROL	GSDTSR	Bcel	Nfe
RETECS	**0.7761**	0.4404	0.1676	0.0964	0.5053
PRM (NAPFDc)	0.5374	**0.5411**	**0.7239**	**0.4185**	**0.5874**

4.4.2　对研究问题 2 的分析

本文接着改变了测试用例输入特征中的历史信息长度的大小, 探究这个变量对方法效果的影响. 历史信息即为某个测试用例过去被执行所得的结果 (成功通过或失败), 基于以前失败过的测试用例很有可能再次失败的假设, 基于历史信息的排序方法[36]使用测试用例的历史执行结果作为排序依据, 其基本思想是将过去频繁失败的测试用例排在前面, 而越长的历史信息则包含了测试用例的更多次的执行结果.

本文在 IOF/ROL 和 Paint Control 数据集上改变使用的测试用例历史信息长度进行实验, 图 8 中横坐标表示使用的历史信息长度, 纵坐标表示在 18 个结果中, 每个结果相对于最好结果的百分比, 由图 8 看出, 指针排序方法受历史信息长度影响相对较小, 在两个数据集上, 历史信息长度由 2 增加到 10 的过程中, 其排序效果整体变化不大, 在 IOF/ROL 数据集上历史信息长度为 2 时即可达到最优效果, 在 Paint Control 数据集上历史信息长度为 5 时即可达到最优的效果, 这是因为指针排序方法在排序时不单考虑到单个测试用例的特征, 还参考了其他测试用例的特征, 故而其对单个测试用例的历史信息依赖相对较小, 可以综合考虑全局信息, 而 RETECS 受历史信息长度的影响较大, 由图 8(a) 知, 在 IOF/ROL 数据集上, 历史信息长度由 8 增加到 9 时, 其性能有了较大的提升, 且随着历史信息长度的增加, 方法效果在不断变好, 表明在该数据集上, RETECS 方法需要较多的历史信息进行测试用例优先级的判断, 而由图 8(b) 知, 在 Paint Control 数据集上, RETECS 在历史信息长度为 3、4 时表现较好, 而后排序性能随着历史信息长度增加而降低, 这表明在该数据集上过多的历史信息又对 RETECS 造成了干扰, 这可能是因为浅层神经网络难以处理如此复杂的信息. 这些结果表明, 指针排序方法不需要很长的历史信息就可以在数据集上达到良好的效果, 且对于所利用的历史信息长度变化不敏感, 能够综合考虑多个测试用例特征进行排序, 而 RETECS 因为基于单个测试用例特征赋予其优先级, 对所利用的历史信息变化敏感, 在使用时需要进行更多的参数调整.

4.4.3　对研究问题 3 的分析

因为在集成过程中新增测试用例没有历史执行结果, 模型无法获得这部分特征, 因此可能学习不到排序策略, 于是本文去除数据集中在集成周期中新增的测试用例, 只保留那些含有历史执行信息的测试用例, 即回归测试用例, 探究方法在这些有历史执行记录的测试用例上的表现如何, 图 9 是方法在 3 个处理后的工业数据集上 (Paint Control、IOF/ROL、GSDTSR) 的排序效果曲线, 表 5、表 6 展示了各个算法 (其中指针排序方法使用 NAPFDc 作为奖励函数) 在处理后的数据集上, 所有集成周期上的 TTF 平均值和召回率平均值的比较.

由图 9、表 5、表 6 可知, 使用 NAPFDc 作为奖励函数时, 指针排序方法仍旧具有良好的学习曲线, 且在 IOF/ROL 和 GSDTSR 数据集上排序效果好于 RETECS, 在 Paint Control 数据集上表现逊于 RETECS 但更为稳定. 与图 6、表 3、表 4 比较, 指针排序方法在这 3 个处理后仅包含回归测试用例的数据集上, 表现与其在原先数据集

© 中国科学院软件研究所 http://www.jos.org.cn

上的表现相差不大, 而 RETECS 在处理后的 Paint Control 和 IOF/ROL 数据集上表现与原先相差不大, 在处理后的 GSDTSR 数据集上模型效果却远好于原数据集, 模型训练曲线的学习趋势更为明显, 这表明 GSDTSR 中新增的测试用例阻碍了 RETECS 的学习, 因为在测试用例的特征表示中, 为了保证历史信息长度的一致, 没有历史信息或历史信息长度小于预先设定值的测试用例, 其缺少的历史执行结果会被当作“成功”对待, 用“成功”的结果填充至固定长度, 这就导致新增测试用例与历史执行结果均为成功的测试用例特征表示没有不同, 使得模型难以区分这二者区别, 并很难据此预测测试用例的失败概率, 这是考虑单个测试用例特征的缺点, 而指针排序方法在排序时能够综合考虑所有测试用例特征, 即便有个别的新增测试用例, 也能够通过执行时长等其他特征的相对比较, 将其放到合适的位置上, 不至于被历史执行信息直接误导. 实验结果表明, 新增测试用例没有对指针排序方法的效果产生很大影响, 该方法在仅含回归测试用例的数据集上表现良好, 能够较好地处理不含历史信息的新加入的测试用例.

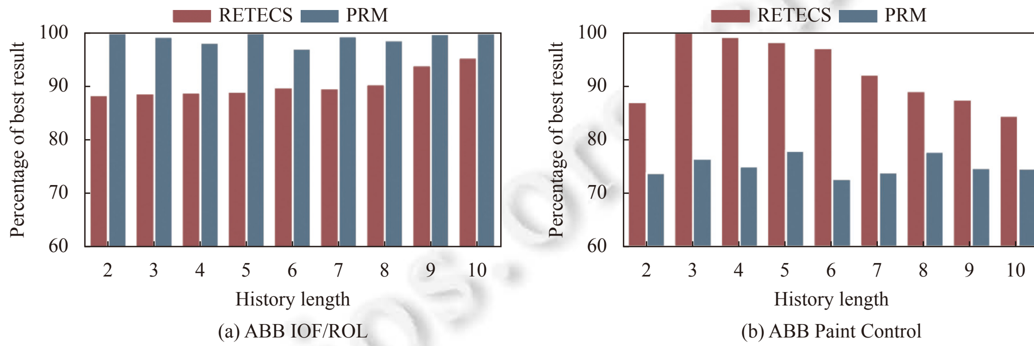

图 8 输入不同历史信息长度的情况下方法的相对表现

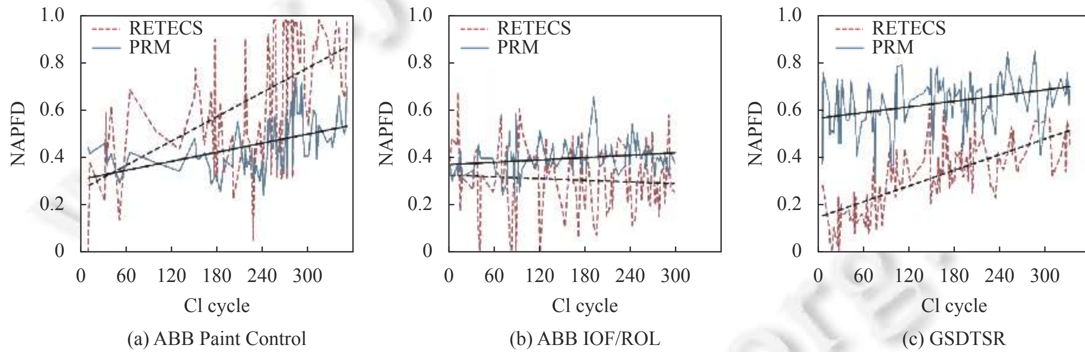

图 9 处理后数据集上不同方法的实验结果

表 5 处理后数据集上不同方法的 TTF 平均值比较

方法	Paint Control	IOF/ROL	GSDTSR
RETECS	**2.786 0**	7.934 7	415.448 6
PRM (NAPFDc)	5.977 4	**4.002 1**	**45.480 4**

表 6 处理后数据集上不同方法的召回率平均值比较

方法	Paint Control	IOF/ROL	GSDTSR
RETECS	**0.768 5**	0.417 3	0.383 8
PRM (NAPFDc)	0.565 9	**0.521 0**	**0.699 3**

4.4.4 对研究问题 4 的分析

我们进一步比较了指针排序算法和 RETECS 算法的训练与预测的平均时间, 实验结果如表 7、表 8 所示.

© 中国科学院软件研究所 http://www.jos.org.cn

表 7　不同方法的平均预测时间的比较 (s)

方法	Paint Control	IOF/ROL	GSDTSR	Bcel	Nfe
RETECS	**0.0142**	**0.0176**	**0.6469**	**0.0017**	**0.0109**
PRM (NAPFDc)	0.1581	0.2115	21.9759	0.0235	0.1296

表 8　不同方法的平均训练时间的比较 (s)

方法	Paint Control	IOF/ROL	GSDTSR	Bcel	Nfe
RETECS	**0.0045**	**0.0033**	**0.4002**	**0.0030**	**0.0017**
PRM (NAPFDc)	0.2204	0.3768	307.4974	0.0352	0.1739

训练时间包含了计算奖励函数和模型根据所得奖励更新参数的时间, 预测时间包含了模型对每个测试用例的优先级进行判定并据此排序的时间. 由结果可知, 总体而言, RETECS 算法的执行效率比指针排序方法要高, 其具有更短的测试时间以及训练时间, 本文认为这是指针排序模型复杂的网络结构所致, 相较 RETECS 模型, 其有更多的网络参数, 因此在排序和训练时需要进行更多的计算. 但是, 在大部分数据集上 (4/5), 指针排序方法的训练/预测时间与 RETECS 相差不多, 如在 Paint Control、IOF/ROL、Bcel、Nfe 数据集上, 指针排序方法的训练与预测时间比 RETECS 长不超过 0.4 s. 虽然前者在 GSDTSR 数据集上的时间开销 (包括预测和排序时间) 要比后者长得多 (由表 1 知, 该数据集有百万级的执行结果数, 平均每个周期需要执行的测试用例数达 3000 以上. 因此, 指针排序方法的时间开销大.), 但持续集成环境中的测试用例排序模型的训练是离线的, 而其平均预测时间仅有 20 s 左右. 因此, 指针排序方法的时间开销不大, 结合其更好、更稳定的排序性能, 本文认为指针排序方法是一个非常有竞争力的方法.

4.5　有效性威胁

在本文场景下, 可能威胁实验有效性的因素分为两类: 内部有效性威胁和外部有效性威胁, 我们对其分析如下, 并阐述了本文如何减轻这些威胁.

(1) 内部有效性的威胁. 这一方面来自于算法实现中可能出现的错误, 为了减轻这方面的威胁, 本文使用了成熟的开源机器学习库 PyTorch、Scikit-learn 构建模型, 并且仔细检查了模型训练过程, 确保模型按照预先设想的网络结构搭建, 按照合理的训练方法进行训练.

另一方面的威胁来自于机器学习算法的随机性对结果造成的影响, 因为初始的策略是通过随机化参数产生, 为了减轻这种威胁, 本文对每个实验都重复进行多次并取其平均值, 确保实验结果不是出于偶然.

在机器学习算法中, 参数的选择十分重要, 不同的参数可能在不同的环境下导致模型表现有较大差异. 为了减轻这个威胁, 本文没有改变文献 [18] 对于模型参数的设置, 并且对于指针排序方法, 在多个数据集上使用的是同一套参数, 没有对其做出调整, 使用的测试用例特征也和 RETECS 保持相同, 以便进行公平的比较.

(2) 外部有效性的威胁. 主要来自于使用的 5 个数据集, 其中 Paint Control、IOF/ROL、GSDTSR 都是被研究者公布并且在相关工作中[18,21–23]得到使用的数据集, 这证明研究者们相信其能够较好地反映真实的持续集成环境, 能够成为衡量算法效果的试验田, 而 Bcel、Nfe 则是来自于 GitHub 的两个项目, 它们是高星数的大型项目 (星数大于 100, 代码行数大于 60000), 具有真实的持续集成环境下的测试数据. 但为了验证算法的有效性, 更多数据集上的实验应该被添加.

5　总结与未来展望

本文提出了一种新的持续集成环境中的测试用例排序技术——基于强化学习的指针排序技术, 它利用了强化学习技术构建了一个模型, 其能够随着软件的演化适应性地调整自己的策略, 不同于以往的关注单个测试用例特征的强化学习方法, 它将测试用例集整体作为模型输入, 在每一步排序时都利用注意力机制考虑所有测试用例的特征, 并且得到对排序整体的结果反馈, 据此动态地调整策略. 相较于以往的机器学习算法, 它在排序时考虑的信

息更加全面, 不容易被单个测试用例的特征误导而陷入局部最优, 且能够根据整体的排序效果反馈直接优化最终目标, 不易被人为设立的中间目标误导.

本文基于 5 个数据集进行了实验, 以便验证方法在真实场景下的排序效果和探究影响其排序效果的因素. 实验结果证明了指针排序方法能够有效地提升测试的质量, 缩短发现软件错误的时间, 且随着集成周期数的增多, 模型会越来越多地学习到如何排序、如何根据环境变化动态调整策略. 本文设计了两个奖励函数: NAPFD 和 NAPFDc, 由实验结果本文发现, 使用 NAPFDc 作为奖励函数, 方法能够更有效地给予揭错且用时短的测试用例更高的优先级, 排序效果好于 RETECS 且表现更为稳定, 因为 NAPFDc 奖励函数考虑了测试用例的执行代价, 能够给予模型更为明确的指示. 通过对输入的改变, 本文探究得到结论: 指针排序方法不需要很长的历史执行信息就可以达到较好的排序效果, 在可获得的历史信息长度改变时方法表现稳定, 因为指针排序方法不只考虑单个测试用例的特征, 还参考了其他测试用例的特征, 故而其对单个测试用例的历史信息特征依赖较小. 此外, 通过对数据集的处理本文发现, 得益于做决策时能够获得所有用例的信息, 方法在回归测试用例集上表现良好, 能够很好地处理新增的无历史信息的测试用例. 通过记录模型的训练和预测时间, 本文发现指针排序方法的时间开销不大. 这些都表明, 指针排序方法是一个有效的测试用例排序技术.

对于未来的展望, 本文有如下考虑: (1) 在更多的持续集成数据集上对指针排序技术的有效性进行验证; (2) 现阶段的编码器设计使得测试用例的输入顺序仍对结果有影响, 且就结果来看方法的效果仍然欠佳, 考虑使用对输入顺序依赖更小的、能提取更深层次特征的神经网络而非 RNN 作为编码器, 设计更为精细的模型; (3) 考虑加入更多有用的测试用例特征, 帮助方法提升排序性能.

References:

[1] Travis CI. 2018. https://travis-ci.org

[2] Circle CI. 2018. https://circleci.com

[3] Jenkins. 2018. https://jenkins.io

[4] Buildbot. 2018. https://buildbot.net

[5] Liang JJ, Elbaum S, Rothermel G. Redefining prioritization: Continuous prioritization for continuous integration. In: Proc. of the 40th IEEE/ACM Int’l Conf. on Software Engineering (ICSE). Gothenburg: ACM, 2018. 688–698. [doi: 10.1145/3180155.3180213]

[6] Memon A, Gao ZB, Nguyen B, Dhanda S, Nickell E, Siemborski R, Micco J. Taming Google-scale continuous testing. In: Proc. of the 39th IEEE/ACM Int’l Conf. on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). Buenos Aires: IEEE, 2017. 233–242. [doi: 10.1109/ICSE-SEIP.2017.16]

[7] Fowler M. Continuous integration, 2006. https://martinfowler.com/articles/continuousIntegration.html

[8] Henard C, Papadakis M, Harman M, Jia M, Traon YL. Comparing white-box and black-box test prioritization. In: Proc. of the 38th IEEE/ACM Int’l Conf. on Software Engineering (ICSE). Austin: IEEE, 2016. 523–534. [doi: 10.1145/2884781.2884791]

[9] Zhang L, Hao D, Zhang L, Rothermel G, Mei H. Bridging the gap between the total and additional test-case prioritization strategies. In: Proc. of the 35th Int’l Conf. on Software Engineering (ICSE). San Francisco: IEEE, 2013. 192–201. [doi: 10.1109/ICSE.2013.6606565]

[10] Hao D, Zhang LM, Zhang L, Rothermel G, Mei H. A unified test case prioritization approach. ACM Trans. on Software Engineering and Methodology, 2014, 24(2): 10. [doi: 10.1145/2685614]

[11] Hao D, Zhao X, Zhang L. Adaptive test-case prioritization guided by output inspection. In: Proc. of the 37th IEEE Annual Computer Software and Applications Conf. Kyoto: IEEE, 2013. 169–179. [doi: 10.1109/COMPSAC.2013.31]

[12] Huang RB, Zhang QJ, Towey D, Sun WF, Chen JF. Regression test case prioritization by code combinations coverage. Journal of Systems and Software, 2020, 169: 110712. [doi: 10.1016/j.jss.2020.110712]

[13] Mondal S, Nasre R. Colosseum: Regression test prioritization by delta displacement in test coverage. IEEE Trans. on Software Engineering, 2022, 48(10): 4060–4073. [doi: 10.1109/TSE.2021.3111169]

[14] Chi JL, Qu Y, Zheng QH, Yang ZJ, Jin WX, Cui D, Liu T. Relation-based test case prioritization for regression testing. Journal of Systems and Software, 2020, 163: 110539. [doi: 10.1016/j.jss.2020.110539]

[15] Chen JJ, Lou YL, Zhang LM, Zhou JY, Wang XL, Hao D, Zhang L. Optimizing test prioritization via test distribution analysis. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Lake Buena Vista: ACM, 2018. 656–667. [doi: 10.1145/3236024.3236053]

© 中国科学院软件研究所 http://www.jos.org.cn

[16] Lima JAP, Vergilio SR. Test case prioritization in continuous integration environments: A systematic mapping study. Information and Software Technology, 2020, 121: 106268. [doi: 10.1016/j.infsof.2020.106268]

[17] Jin XH, Servant F. What helped, and what did not? An evaluation of the strategies to improve continuous integration. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Madrid: IEEE, 2021. 213–225. [doi: 10.1109/ICSE43902.2021.00031]

[18] Spieker H, Gotlieb A, Marijan D, Mossige M. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In: Proc. of the 26th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Santa Barbara: ACM, 2017. 12–22. [doi: 10.1145/3092703.3092709]

[19] Busjaeger B, Xie T. Learning for test prioritization: An industrial case study. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Seattle: ACM, 2016. 975–980. [doi: 10.1145/2950290.2983954]

[20] Bertolino A, Guerriero A, Miranda B, Pietrantuono R. Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering. Seoul: IEEE, 2020. 1–12. [doi: 10.1145/3377811.3380369]

[21] He LL, Yang Y, Li Z, Zhao RL. Reward of reinforcement learning of test optimization for continuous integration. Ruan Jian Xue Bao/Journal of Software, 2019, 30(5): 1438−1449 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5714.htm [doi: 10.13328/j.cnki.jos.005714]

[22] Yang Y, Li Z, He LL, Zhao RL. A systematic study of reward for reinforcement learning based continuous integration testing. Journal of Systems and Software, 2020, 170: 110787. [doi: 10.1016/j.jss.2020.110787]

[23] Bagherzadeh M, Kahani N, Briand L. Reinforcement learning for test case prioritization. IEEE Trans. on Software Engineering, 2022, 48(8): 2836–2856. [doi: 10.1109/TSE.2021.3070549]

[24] Wong WE, Horgan JR, London S, Agrawal H. A study of effective regression testing in practice. In: Proc. of the 8th Int'l Symp. on Software Reliability Engineering. Albuquerque: IEEE, 1997. 264–274. [doi: 10.1109/ISSRE.1997.630875]

[25] 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]

[26] Rothermel G, Untch RH, Chu CY, Harrold MJ. Prioritizing test cases for regression testing. IEEE Trans. on Software Engineering, 2001, 27(10): 929–948. [doi: 10.1109/32.962562]

[27] Li Z, Harman M, Hierons RM. Search algorithms for regression test case prioritization. IEEE Trans. on Software Engineering, 2007, 33(4): 225–237. [doi: 10.1109/TSE.2007.38]

[28] Zhang L, Hou SS, Guo C, Xie T, Mei H. Time-aware test-case prioritization using integer linear programming. In: Proc. of the 18th Int'l Symp. on Software Testing and Analysis. Chicago: ACM, 2009. 213–224. [doi: 10.1145/1572272.1572297]

[29] Elbaum S, Rothermel G, Penix J. Techniques for improving regression testing in continuous integration development environments. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Hong Kong: ACM, 2014. 235–245. [doi: 10.1145/2635868.2635910]

[30] Yu TT, Wang T. A study of regression test selection in continuous integration environments. In: Proc. of the 29th IEEE Int'l Symp. on Software Reliability Engineering (ISSRE). Memphis: IEEE, 2018. 135–143. [doi: 10.1109/ISSRE.2018.00024]

[31] Kim JM, Porter A. A history-based test prioritization technique for regression testing in resource constrained environments. In: Proc. of the 24th Int'l Conf. on Software Engineering. Orlando: IEEE, 2002. 119–129. [doi: 10.1109/ICSE.2002.1007961]

[32] Marijan D, Gotlieb A, Sen S. Test case prioritization for continuous regression testing: An industrial case study. In: Proc. of the 2013 IEEE Int'l Conf. on Software Maintenance. Eindhoven: IEEE, 2013. 540–543. [doi: 10.1109/ICSM.2013.91]

[33] Strandberg PE, Sundmark D, Afzal W, Ostrand TJ, Weyuker EJ. Experience report: Automated system level regression test prioritization using multiple factors. In: Proc. of the 27th IEEE Int'l Symp. on Software Reliability Engineering (ISSRE). Ottawa: IEEE, 2016: 12–23. [doi: 10.1109/ISSRE.2016.23]

[34] Srikanth H, Cashman M, Cohen MB. Test case prioritization of build acceptance tests for an enterprise cloud application: An industrial case study. Journal of Systems and Software, 2016, 119: 122–135. [doi: 10.1016/j.jss.2016.06.017]

[35] Hemmati H, Fang ZH, Mäntylä MV, Adams B. Prioritizing manual test cases in rapid release environments. Software Testing, Verification and Reliability, 2017, 27(6): e1609. [doi: 10.1002/stvr.1609]

[36] Haghighatkhah A, Mäntylä M, Oivo M, Kuvaja P. Test prioritization in continuous integration environments. Journal of Systems and Software, 2018, 146: 80–98. [doi: 10.1016/j.jss.2018.08.061]

[37] Gligoric M, Eloussi L, Marinov D. Practical regression test selection with dynamic file dependencies. In: Proc. of the 2015 Int'l Symp. on Software Testing and Analysis. Baltimore: ACM, 2015. 211–222. [doi: 10.1145/2771783.2771784]

[38] Sutton RS, Barto AG. Reinforcement Learning: An Introduction. Cambridge: MIT Press, 1998. 43–47.

© 中国科学院软件研究所 http://www.jos.org.cn

[39] Hao D, Zhang L, Zang L, Wang YB, Wu XX, Xie T. To be optimal or not in test-case prioritization. IEEE Trans. on Software Engineering, 2015, 42(5): 490–505. [doi: 10.1109/TSE.2015.2496939]
[40] Hao D, Zhang L, Mei H. Test-case prioritization: Achievements and challenges. Frontiers of Computer Science, 2016, 10(5): 769–777. [doi: 10.1007/s11704-016-6112-3]
[41] Vinyals O, Fortunato M, Jaitly N. Pointer networks. In: Proc. of the 28th Int'l Conf. on Neural Information Processing Systems. Montreal: NIPS, 2015. 2692–2700.
[42] Bello I, Pham H, Le QV, Norouzi M, Bengio S. Neural combinatorial optimization with reinforcement learning. arXiv:1611.09940, 2016.
[43] Qu X, Cohen MB, Woolf KM. Combinatorial interaction regression testing: A study of test case generation and prioritization. In: Proc. of the 2007 IEEE Int'l Conf. on Software Maintenance. Paris: IEEE, 2007. 255–264. [doi: 10.1109/ICSM.2007.4362638]
[44] Mnih V, Badia AP, Mirza M, Graves A, Lillicrap T, Harley T, Silver D, Kavukcuoglu K. Asynchronous methods for deep reinforcement learning. In: Proc. of the 33rd Int'l Conf. on Machine Learning. New York: PMLR, 2016. 1928–1937.
[45] 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. Oxford: IEEE, 1999. 179–188. [doi: 10.1109/ICSM.1999.792604]

附中文参考文献:

[21] 何柳柳, 杨羊, 李征, 赵瑞莲. 面向持续集成测试优化的强化学习奖励机制. 软件学报, 2019, 30(5): 1438−1449. http://www.jos.org.cn/1000-9825/5714.htm [doi: 10.13328/j.cnki.jos.005714]

赵逸凡(1999－), 男, 博士生, CCF 学生会员, 主要研究领域为软件测试.

郝丹(1979－), 女, 博士, 副教授, 博士生导师, CCF 杰出会员, 主要研究领域为软件测试.

© 中国科学院软件研究所 http://www.jos.org.cn