软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2022,33(5):1674−1698 [doi: 10.13328/j.cnki.jos.006555]

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

# 面向类集成测试序列生成的强化学习研究*

丁艳茹[1,2], 张艳梅[1,2,3], 姜淑娟[1,2], 袁 冠[1,2], 王荣存[1,2], 钱俊彦[4]

[1](中国矿业大学 矿山数字化教育部工程研究中心, 江苏 徐州 221116)
[2](中国矿业大学 计算机科学与技术学院, 江苏 徐州 221116)
[3](桂林电子科技大学 广西可信软件重点实验室, 广西 桂林 541004)
[4](广西师范大学 广西多源信息挖掘与安全重点实验室, 广西 桂林 541004)
通信作者: 张艳梅, E-mail: ymzhang@cumt.edu.cn; 姜淑娟, E-mail: shjjiang@cumt.edu.cn

**摘 要**: 集成测试是软件测试过程中不可缺少的步骤, 针对在集成测试中如何对系统中的类合理排序的问题, 国内外研究者提出了多种生成类集成测试序列的方法, 然而他们大多没有将测试桩复杂度作为评估测试代价的指标. 针对该问题, 提出面向类集成测试序列生成的强化学习研究方法, 以总体测试桩复杂度为评价测试代价的指标, 生成测试代价尽可能低的类集成测试序列. 首先, 定义强化学习任务, 根据任务设定算法的追求目标; 其次, 进行程序的静态分析, 根据分析得到的结果计算测试桩复杂度; 然后, 将测试桩复杂度的计算融入奖励函数的设计中, 为选择下一步动作提供信息和依据; 最后, 通过奖励函数反馈值函数, 通过值函数的设定保证累计奖励最大化. 当智能体完成规定训练次数, 系统会选择获得最大累计奖励值的类集成测试序列进行输出, 即为我们追求的测试代价尽可能低的结果. 实验结果表明, 与现有方法相比, 在以总体测试桩复杂度为评价指标时, 提出的方法结果更优.
**关键词**: 类集成测试序列; 强化学习; 测试桩; 测试代价; 奖励函数
**中图法分类号**: TP311

中文引用格式: 丁艳茹, 张艳梅, 姜淑娟, 袁冠, 王荣存, 钱俊彦. 面向类集成测试序列生成的强化学习研究. 软件学报, 2022, 33(5): 1674–1698. http://www.jos.org.cn/1000-9825/6555.htm
英文引用格式: Ding YR, Zhang YM, Jiang SJ, Yuan G, Wang RC, Qian JY. Generation Method of Class Integration Test Order Based on Reinforcement Learning. Ruan Jian Xue Bao/Journal of Software, 2022, 33(5): 1674–1698 (in Chinese). http://www.jos.org.cn/1000-9825/6555.htm

## Generation Method of Class Integration Test Order Based on Reinforcement Learning

DING Yan-Ru[1,2], ZHANG Yan-Mei[1,2,3], JIANG Shu-Juan[1,2], YUAN Guan[1,2], WANG Rong-Cun[1,2], QIAN Jun-Yan[4]

[1](Engineering Research Center of Mine Digitalization (China University of Mining and Technology), Ministry of Education, Xuzhou 221116, China)
[2](College of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221116, China)
[3](Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin 541004, China)
[4](Guangxi Key Laboratory of Multi-source Information Mining & Security, Guangxi Normal University, Guilin 541004, China)

**Abstract**: Integration testing is an indispensable step in the software testing process. In response to the problem of how to rationally sort the classes in the system in integration testing, researchers worldwide have proposed a variety of methods to generate class integration test orders. However, most of them didn't take the stubbing complexity as the indicator, which is an important factor in evaluating the test cost. In order to solve this problem, this study proposes a method of generating a class integration test order based on reinforcement learning, using the overall stubbing complexity as the indicator to evaluate the test cost, and generating a class integration test order with

* 基金项目: 国家自然科学基金 (61673384, 71774159, 62162004, 51874292); 广西可信软件重点实验室开放课题 (kx201608); 广西自然科学基金重点项目 (2018GXNSFDA138003)
本文由“领域软件工程”专题特约编辑汤恩义副教授、江贺教授、陈俊洁副教授、李必信教授以及唐滨副教授推荐.
收稿时间: 2021-08-10; 修改时间: 2021-10-09; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

the stubbing complexity as low as possible. First, the reinforcement learning task is defined, and the pursuit goal of the algorithm is set according to the task. Second, the static analysis of the program is performed and the stubbing complexity is calculated according to the results of the analysis. Then, the calculation of the stubbing complexity is integrated into the design of the reward function to provide information and basis for choosing the next action. Finally, the value function is fed back through the reward function, and the value function is set to ensure that the cumulative reward is maximized. When the agent completes the specified number of training times, the system will select the class integration test order that obtains the largest cumulative reward value for output, which costs the lowest stubbing complexity pursued. The experimental results show that the results obtained by this method are better than those obtained by other existing methods in terms of the overall stubbing complexity as the evaluation indicator.

**Key words**: class integration test order; reinforcement learning; test stub; test cost; reward function

软件测试阶段主要包括单元测试、集成测试、系统测试以及回归测试等. 其中, 集成测试是在单元测试的基础上进行的步骤, 主要任务是将所有的软件单元组装成模块、子系统或系统, 检测各部分工作是否达到或实现相应技术指标及要求, 以保证各单元组合之后能够按既定意图协作运行等. 但是, 面向对象的程序没有明显的层次划分, 其间的调用关系表现为错综复杂的网状结构, 传统的集成测试策略并不能很好地应用其中. 因此, 需要提出符合面向对象程序特点的新的集成测试策略[1], 这种新的策略以类为对象、以生成最优的类集成测试序列为目的, 进而确定测试方式.

根据面向对象程序的类间依赖性, 软件工程领域的研究者们提出了基于类集成测试序列的集成策略. 在测试过程中, 这些策略往往需要为面向对象程序中的某些类构造测试桩, 以代替其完成某些功能. 这项任务的代价很大, 并且一般来说没有办法避免. 因此, 如何降低这种代价成为了集成测试中一项关键性的问题. 研究过程中, 学者们通过计算测试桩复杂度衡量测试桩的代价, 对于不同的类集成测试序列, 测试桩复杂度不尽相同, 其测试代价也不相同. 合理地对待测程序中的类进行排序, 得到可行的类集成测试序列, 可以大大降低需要构建的测试桩的总体复杂度, 尽可能地减小测试代价. 总的来说, 提出合理的类集成测试序列生成技术对于集成测试来说具有相当重要的意义.

通过对国内外相关算法的研究, 我们发现目前解决类集成测试序列的生成问题普遍采用基于图论与基于搜索的方法, 极少涉及到强化学习方面的相关知识, 强化学习适用于解决序列决策问题, 已经成功应用于游戏问题 (如 AlphaGo、AlphaZero 在围棋方面的突破)[2]、面向对象测试用例生成问题[3]以及类集成测试序列问题[4]中. 然而, 已有的基于强化学习的类集成测试序列生成方法评估确定类集成测试序列花费的总体代价的指标不够精确, 忽略了测试桩复杂度这一重要的评价指标, 这些方法假设每个类间关系依赖程度相同, 即, 每个测试桩具有相同的复杂度[4]. 但是, 不同的测试桩具有不同的复杂度, 构建的测试桩越少不能表示确定一个类集成测试序列花费的测试代价越低. 因此, 需要研究基于强化学习的面向类集成测试序列生成的确定技术, 满足总体测试桩复杂度尽可能低.

针对上述问题, 本文提出了一种面向类集成测试序列生成的强化学习研究方法, 重点探究集成测试中如何生成测试代价最低的类集成测试序列. 该方法通过强化学习中的 Q-学习算法[4], 构建智能体, 设计奖励函数, 使智能体在环境中探索, 逐步选择可以降低测试桩复杂度的动作, 进而得到最优的的学习路径, 获取最终的类集成测试序列. 这可以为实际生产生活中测试人员开展测试工作提供更为准确的度量方法, 提高工作效率.

本文第 1 节介绍了类集成测试序列生成过程中涉及到的一些基本概念; 第 2 节介绍了面向类集成测试序列生成的强化学习研究策略, 并给出了一个实例分析; 第 3 节介绍了为验证本文方法的效果所做的实验设计, 并对设计内容展开实验, 分析实验结果; 第 4 节总结了近年来国内外研究者在类集成测试序列方面的相关工作; 第 5 节对本文的研究方法进行总结, 并对未来类集成测试序列生成领域的研究方向进行展望.

## 1 基本概念

类集成测试序列的生成过程中涉及到的基本概念可以概括为 3 部分, 分别是集成测试相关、类间依赖关系分类以及测试代价度量.

### 1.1 集成测试相关

整个软件系统的生命周期都存在测试, 根据在生命周期中所处的阶段, 软件测试可以分为以下几个类别[5].

(1) 单元测试, 存在于软件系统的编码阶段, 针对组成程序的最小单元进行检验;

(2) 集成测试, 当上述经过单元测试的程序单元组装到一起, 变为更大的子系统时, 对集成过程进行的测试;

(3) 系统测试, 当系统集成完毕, 针对目前软件所处的实际环境进行的测试;

(4) 回归测试&β 测试, 针对系统后期维护阶段以及后续版本的发布等进行的测试.

其中, 按面向对象程序中的层级划分又可以将集成测试视为类簇级测试. 类簇指一组类的集合, 它们之间由于各种方法的调用往往形成复杂的类间关系, 类簇级测试的功能就是检验系统内的各种关系的实现、消息传递和系统功能正常运作与否.

一个模块单独工作时运行正常, 但是集成之后是否还能完成其功能需要进一步测验, 因此, 为了使单元测试中看不到的问题暴露出来, 类簇级测试应运而生. 经过了初步的单元测试, 类簇级测试需要验证所需单元按照设计需求进行整合、组装成待测子系统的过程是否符合前期设计需求. 并且, 由于在某些开发系统中, 代码的编写与测试都是在不断迭代的过程中完成的, 因此, 作为确保各单元的正常结合、完成所需求的功能的测试步骤, 类簇级测试就彰显了其重要性.

本文研究的类集成测试序列的生成问题属于面向对象的类簇级测试中极为重要的一个环节. 类簇中类间关系复杂, 通过消息传递产生的依赖关系可以抽象为一个复杂的面向对象关系图, 程序中出现的类间关系越复杂, 越使得同时测试完系统中所有的类变的无法实现. 因此, 如何确定测试这些类之间的先后顺序以使测试整个系统的代价降到最低成为了一个关键性问题.

### 1.2 类间依赖关系分类

要对待测系统中的类进行排序, 就要先分析类间的各种依赖关系. 类间依赖关系[6]可以依据程序运行与否分为类间静态依赖关系与类间动态依赖关系. 程序无需运行即可进行分析的称为类间静态依赖关系, 需要程序运行才能形成的关系称为类间动态依赖关系. 本文涉及的主要是静态依赖关系.

类间静态依赖关系主要分为 5 种, 分别为: 使用 (use)、关联 (association)、聚合 (aggregation)、组合 (composition) 以及继承/泛化 (generalization)[6].

使用 (use): 使用关系也称为依赖关系 (dependency), 表示类间某个类依赖于另一个类的关系, 属于依赖强度最弱的类间关系. 类 A 和类 B 分别表示两个独立的类, 当由于类 A 引入了类 B 时, 称类 A 依赖于类 B, 但是 A 中不包含 B 的属性, B 的实例以参量形式传入 A 的方法中. Java 中使用关系主要表现为方法形参 (B 的实例表现为 A 中的方法或者参数)、局部变量 (B 的对象视作 A 的局部变量) 以及静态方法的调用 (A 调用 B 的静态方法) 等.

关联 (association): 关联关系的依赖强度稍强于依赖关系, 表示某个类了解另一个类的方法属性, 这种关系可以是单向的也可以是双向的. 单向关联关系可以表示为: 当类 A 调用了 B 内某实例, A 单方面了解 B 中的方法属性, 但是 B 不了解 A 中的信息, 就称为 A 单向关联于 B. 双向关联关系就是指 A 和 B 彼此都知道对方的方法属性. Java 中关联关系一般表示为类里的成员变量.

聚合 (aggregation): 也称聚集, 属于一种特殊的更强的关联关系, 主要表示整体和部分之间的关系. 类 A 代表整体, 类 B 代表部分, B 属于 A 中的一个个体, A 中的全局变量存在 B 中实例的声明, A 和 B 即为一种聚合关系. 这里 A 和 B 属于同一级别, 当部分脱离整体仍然可以单独作为一个类存在于整个系统. Java 中表现为 A 将 B 的已经实例化的部分作参数, 然后将该参数传递给构造函数.

组合 (composition): 组合与聚合同属于关联关系的一种形式, 但比聚合关系的依赖强度更深, 因此别名“强聚合”. 同上, 假设类 B 代表部分, 类 A 代表整体, B 是 A 的一个个体, 同聚合关系一样, 类 B 出现在类 A 的全局变量声明中, 不同的是, 如果 B 脱离于 A, 则 B 在系统中没有了存在性, 即部分不能脱离整体而存在. Java 中表现为 B 的对象属性创建在 A 的构造函数中.

继承/泛化 (generalization): 继承和泛化同属于子类父类关系的过程, 继承指父类具体出子类的过程, 泛化指子类抽象出父类的过程. 在这种关系里, 以类 A 代表父类, 类 B 代表子类, B 可以共享 A 中的部分属性以及方法, 并在重写父类方法的同时重新定义, 实现自己需要的属性以及方法, 这也属于 B 依赖于 A 的一种表现. Java 中, 某个

子类只可以继承单个父类, 继承关系通过 extends 关键字实现.

如图 1 所示, Briand 等人[7]在设计对象关系图 (object relation diagram, ORD)[8]时为了简化, 将 5 种关系缩减为 3 种[7], 继承关系仍表示为 I, 聚合与组合关系表示为 Ag, 关联和使用 (包括简单的聚合关系) 都表示为 As.

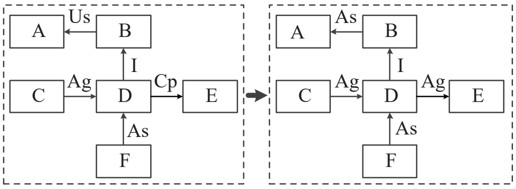



图 1 约减的类间关系图

### 1.3 测试代价度量

(1) 测试桩

在集成测试中, 测试桩用来模拟某个类所需的仍未被测试的服务组件, 其定义如下.

**定义 1**. 测试桩[6]. 假设将被集成测试的系统中存在两个独立的类, 分别为类 X 和类 Y, 若 X 调用了 Y 中的方法, 即 X 依赖于 Y, 则需要 Y 先于 X 被集成, 才可以令 X 的功能完整实现, 但是实际中并不能保证模块集成的顺序一定是 Y 先于 X. 因此当 X 先于 Y 被集成, 就需要构建一个组件代替模拟 X 需要的方法, 使得 X 可以顺利被测试, 这个被构建的组件或这段代码就被称为测试桩 (stub).

在生成类集成测试序列的过程中, 测试桩的构建较为复杂, 因此通常用建立测试桩的代价衡量效果优劣, 测试桩一般被分为通用测试桩 (generic stub) 和特效测试桩 (specific stub), 通用测试桩通常用来模拟整个类的行为, 容易增加额外开销, 而特效测试桩模拟类的一部分, 比如如何使用这个类所需的单个方法, 无法重用[6].

如图 2 所示, 图中包含 4 个类 A、B、C 和 D, A 调用 B 中某方法, B 调用 C 中某方法, C 调用 D 中某方法, D 调用 A 和 C 中某方法. 根据类间依赖关系, 可以得到需要建立的测试桩如下.

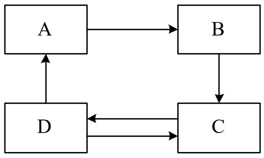



图 2 四类图

通用测试桩[6]: A 依赖于 B, 集成 A 时, B 尚未被集成, 需要为 A 建立 B 的通用测试桩, 同理需要为 B 建立 C 需要的通用测试桩, D 依赖于已集成的 A 和已建立通用测试桩的 C, 故无需再建. 因此图 2 需要的通用测试桩有 B 和 C 两个.

特效测试桩[6]: A 依赖于 B, 集成 A 时, 需要建立一个 B 中针对 A 调用的方法的特效测试桩, 同理为 B 建立 C 中相应的特效测试桩, 为 D 建立 C 中 D 需要的方法的特效测试桩. 故上图需要的特效测试桩有 B 为 A 建立的、C 为 B 建立的、C 为 D 建立的共 3 个.

测试桩并非一个系统中真正的类, 是服务于被测对象的一个组件模块或一段目标代码. 当两个类之间的依赖关系较强时, 测试桩需要模拟的功能就比较多, 构建起来较为复杂; 当关系较弱时, 构建测试桩的难度降低, 代价较小. 因此, 可以通过分析类间依赖关系得到测试代价.

(2) 类间耦合度量

构建测试桩的过程中需要考虑类间依赖关系的强弱, 而度量这种依赖关系就需要分析类间的耦合信息, 计算构建测试桩的代价. 经过分析, 耦合度的紧密程度与依赖关系的强弱呈正相关关系, 即耦合度与测试代价呈正相关关系.

根据 Briand 等人[9]的研究, 类间耦合信息主要可以分为属性耦合 (attribute coupling) 与方法耦合 (method coupling)[9]这两种, 其主要定义分别如下.

**定义 2**. 属性耦合[9]. 存在两个独立类, 类 X 与类 Y, 假设 X 是源类, Y 是目标类, X 依赖于 Y, 包含以下 3 种情况时可以称之为类间存在属性耦合: (1) X 调用 Y 的成员变量; (2) X 调用 Y 中的实例或属性作为自己的方法参数; (3) X 调用 Y 中的实例或属性作为自己的方法返回值.

**定义 3**. 方法耦合[9]: 若 X 调用 Y 中方法, 则称 X 和 Y 之间存在方法耦合. 这种调用可以是 X 调用 Y 中的构造函数, 也可以是 X 中存在调用 Y 中方法的语句.

根据以上定义, 结合 X 和 Y 的耦合信息, 进一步得到 X 依赖于 Y 这一步如果构建测试桩分别需要的属性耦合和方法耦合的数量, 具体为: 根据 X 访问 Y 的属性个数、X 访问 Y 的方法个数以及 X 访问 Y 的返回值个数统计属性耦合; 根据 Y 向 X 传递的不同参数的个数统计方法耦合. 进一步计算得到两个指标各自的复杂度, 进而得到集成 X 时建立 Y 测试桩需要的测试代价.

(3) 测试代价度量方法

针对构建测试桩的代价的度量标准有两个[8], 一是根据测试桩的数目, 包括通用测试桩和特效测试桩; 二是根据测试桩复杂度. 这两种度量策略是当下各研究者解决衡量类集成测试序列问题普遍认同的方法.

测试桩数目[8]: 早期学者主要通过构建的测试桩数目衡量测试代价的高低. 当几个待测类共同依赖于某一个类时, 构建特效测试桩的数目等同于待测类数目, 构建通用测试桩需要模拟整个类的方法, 两种方式均开销极大. 并且, 这种方法忽略了每个类之间的属性和方法因素, 认为破坏不同类间的依赖边代价等同, 因此单以数量高低衡量并不准确.

测试桩复杂度[10]: 基于以上原因, Briand 等人[10]提出结合类间属性和方法耦合度量测试代价的公式, 基于线性加权计算构建测试桩的代价.

首先, 通过属性复杂度计算类间的属性耦合, 使用 $A(i,j)$ 表示, $i$、$j$ 分别表示程序中的类, 属性复杂度数量上等于 $i$ 调用 $j$ 的成员变量、方法参数类型与方法的返回值数目三者之和. 然后, 通过方法复杂度计算类间的方法耦合, 使用 $M(i,j)$ 表示, 方法复杂度数目上等于 $i$ 调用 $j$ 中方法的数目. 为防止作用效果不均, 在公式中对二者进行归一化[11], 这样得到的结果均在 0 到 1 之间, 归一化公式如 (1)、公式 (2) 所示.

$$\overline{A(i,j)}=\frac{A(i,j)}{A_{\max}-A_{\min}} \tag{1}$$

$$\overline{M(i,j)}=\frac{M(i,j)}{M_{\max}-M_{\min}} \tag{2}$$

对上述二者再分别赋予权重 $W_A$ 和 $W_M$ 来计算测试桩复杂度 $SCplx(i,j)$[11], 计算过程如公式 (3) 所示.

$$SCplx(i,j)=\left[W_A\cdot\overline{A(i,j)}^2+W_M\cdot\overline{M(i,j)}^2\right]^{\frac{1}{2}} \tag{3}$$

函数 $SCplx(i,j)$ 的值越高, 表明构建为 $i$ 模拟 $j$ 所需的功能模块的测试桩的代价越大. 用 $o$ 表示集成的整个类集成测试序列, 用 $OCplx(o)$ 表示总体测试桩复杂度, 用 $N$ 表示需要构建的测试桩集合, 则生成类集成测试序列的总体测试桩复杂度[11]公式为:

$$OCplx(o)=\sum_{(i,j\in N)}SCplx(i,j) \tag{4}$$

这种归一化处理计算属性和方法复杂度的方法可能会使计算的测试桩复杂度丧失一部分原有的信息, 但易于实现, 处理方便, 是目前公认的计算测试桩复杂度的策略, 通过线性加权生成类集成测试序列总体测试桩复杂度的方法也普遍出现在各种算法中, 例如遗传算法、蚁群算法等.

## 2 面向类集成测试序列生成的强化学习研究

类集成测试序列问题本质上是一个 NP 完全性问题, 近年来, 由于人工智能的崛起, 人们发现其在解决 NP 完全性问题方面具有一定效用. 目前, 解决该问题普遍采用基于图论与基于搜索的方法, 极少涉及到强化学习方面的知识, 然而又由于强化学习适用于序列决策问题, 而类集成测试序列问题本身属于序列决策问题, 因此本文利用强

化学习来解决类集成测试序列的生成问题.

### 2.1 强化学习结构

强化学习是机器学习领域的一个重要分支, 其结构如图 3 所示. 首先, 智能体与周围环境交互, 在 $t$ 时刻得到外部环境的当前状态 $s_t$, 采取试探行为 $a_t$; 然后, 通过状态到动作的映射, 得到环境对这一步动作的反馈 $r$, 如果得到正反馈, 则下一步加强采取这个动作的趋势, 反之则减弱趋势; 之后, 得到下一状态 $s_{t+1}$, 智能体继续进行学习; 最终, 达到使智能体获得的收益最大化的追求目标.

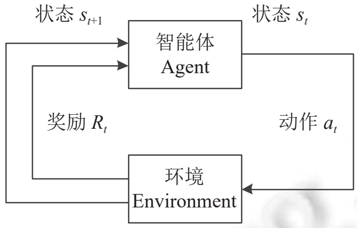



图 3　强化学习结构

强化学习主要由两个交互对象以及 3 项基本要素[2]构成, 其中, 交互对象包括智能体 (agent) 和环境 (environment), 基本要素包括状态 ($s$, state)、动作 ($a$, action) 以及奖励 ($r$, reward).

智能体 (agent): 执行动作的载体, 可以与环境交互, 得到环境的状态, 接收反馈的奖励, 进而调整学习策略, 决策出下一步动作的选择.

环境 (environment): 系统中不属于智能体的部分, 智能体在这里执行动作, 使自己处于多个状态, 进行探索并获得奖励.

状态 ($s$, state): 不断变化的量, 状态空间 $S$ 中包含智能体所能处于的各种状态.

动作 ($a$, action): 智能体采取的动作, 动作集合 $A$ 中包含智能体所能采取的各种动作.

奖励 ($r$, reward): 一个函数, 当智能体在状态 $s$ 下采取动作 $a$ 时, 环境会综合给予智能体一个奖励 $r$, 当智能体获得的 $r$ 为正向时, 智能体会加强相关方向动作的选择, 同时也会影响下一状态.

强化学习是以马尔科夫决策过程 (Markov decision process, MDP)[2]为理论基础的, 即智能体选择动作发生的条件概率只和当前事务所处状态相关, 但同时, 当前的动作会以一定概率影响到下一状态的转变, 智能体根据状态和动作得到奖励值 $R_i=r(s_i, a_i)$. 假设路径动作历史共有 $n$ 个动作, 则马尔科夫决策过程具体可表示如图 4 所示. 在既定的策略条件下, 马尔科夫决策过程的目标就是找到一组最优的序列, 使所获得的总体奖励最大. 这个过程还需要一个折扣因子 $\gamma$ ($\gamma\in(0, 1)$), 因此总的来说, 总体奖励 $R$ 可用公式 (5) 表示.

$$R=r(s_1,a_1)+r(s_2,a_2)\gamma+\ldots+r(s_i,a_i)\gamma^{i-1}+\ldots+r(s_n,a_n)\gamma^{n-1} \tag{5}$$

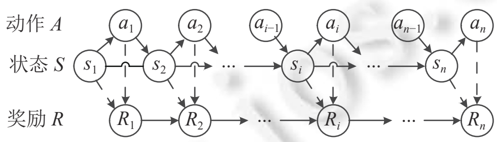



图 4　马尔可夫决策过程

### 2.2 基于熵权法的测试桩复杂度度量

度量类集成测试序列的评价指标有两种, 分别是测试桩数量和测试桩复杂度. 但是, 现有的以测试桩数量作为评价指标的方式将每个测试桩视作等价, 忽略了类间复杂的依赖关系, 因此, 当使用两种方法分别用于度量构建测试桩的测试代价时, 测试桩复杂度更为准确.

基于上述原因, 本文研究了一种从测试桩复杂度的评价角度进一步降低测试代价的类集成测试序列生成方法. 并且, 由于现有的方法中忽略了类间属性复杂度和方法复杂度的权值的区别, 为了使结果更精确, 本方法考虑

采用熵权法度量二者权值, 进而计算测试桩复杂度.

评估测试桩复杂度的代价函数 *Cplx*() 表示类 $i$ 和 $j$ 之间的边复杂度, 首先对其标准化, 即用现有的边值除以边最大值和最小值之差, 标准化之后的结果在 0 和 1 之间, 这种方法可以避免不同级别的值对准确率造成的影响, 下文中的标准化同理. 对 *Cplx*() 标准化的过程如公式 (6) 所示:

$$\overline{Cplx(i,j)}=\frac{Cplx(i,j)}{Cplx_{\max}-Cplx_{\min}} \tag{6}$$

然后, 依据前文中介绍到的公式 (1)、公式 (2) 分别为得到的属性复杂度和方法复杂度的值标准化.

$$SCplx(i,j)=\left[W_A\cdot\overline{A(i,j)}^2+W_M\cdot\overline{M(i,j)}^2\right]^{\frac{1}{2}} \tag{7}$$

最后, 再对二者进行加权耦合, 计算过程如公式 (7) 所示. 其中, $\overline{A(i,j)}$ 代表标准化后的类 $i$ 和 $j$ 之间的属性复杂度值, 包含 $i$ 调用 $j$ 中的成员变量、方法参数和方法返回值的数量; $\overline{M(i,j)}$ 代表 $i$ 调用 $j$ 中的方法数进行标准化后得到的方法复杂度值, 这里也包含构造函数. 然后对其分别赋予权值 $W_A$ 和 $W_M$, 权值越大, 该指标在计算测试桩复杂度时占比越大.

通常来说, 为指标赋权的方法可以分为主观与客观两种, 主观方法具有较大的灵活空间, 但这种方法依赖主观因素, 忽略了其他因素 (例如随机性等) 对评估结果的影响. 信息论的原理中用信息熵表示系统的无序程度, 定量度量信息[12]. 系统中的某个指标越混乱, 不确定性越高, 熵值就越高, 表示该指标携带的信息越少. 反之, 当指标越有序, 熵值越低, 表示携带的信息越多, 也更有参考价值. 熵的计算只依赖样本数据本身的离散程度, 因此经常被应用到研究中, 通过用样本数据来计算指标的权重, 这种方法也被称为熵权法[12]. 本文中, 为了更准确的得到测试桩复杂度, 采用熵权法[12]来计算属性和方法复杂度的权值.

熵权法[12]计算测试桩复杂度的步骤如下.

(1) 标准化指标

就本文而言, 计算测试桩复杂度有两个指标, 分别是属性复杂度与方法复杂度, $A(i,j)$ 代表类 $i$ 和 $j$ 之间的属性复杂度, $M(i,j)$ 代表类 $i$ 和 $j$ 之间的方法复杂度, 熵权法最先进行的就是对两个指标进行标准化, 这样得到的结果均在 0 到 1 之间. 标准化过程同样如公式 (1)、公式 (2) 所示, 得到类间属性复杂度 $\overline{A(i,j)}$, 类间方法复杂度 $\overline{M(i,j)}$.

(2) 建立评价矩阵

假设待测系统包含 $m$ 个类, 则可以构造一个 $m\times 2$ 的矩阵表示类间的两种关系, 其中以第 1 列表示在属性复杂度的指标下的类间关系的评价值, 第 2 列表示在方法复杂度的指标下类间关系的评价值. 例如:$r_{i1}$ 代表第 $i$ 个属性复杂度值, $r_{i2}$ 代表第 $i$ 个方法复杂度值. 两列共同构成一个评价矩阵 $R$, 计算过程如公式 (8) 所示.

$$R=(r_{ij})_{m\times 2}(i=1,2,\ldots,m;j=1,2) \tag{8}$$

(3) 计算信息熵

计算信息熵之前需要首先计算每个类对应每个指标的评价值占比, 以 $P_{ij}$ 表示第 $j$ 个指标的比重, 如公式 (9), 然后根据得到的比重计算第 $j$ 个指标的信息熵 $e_j$, 如公式 (10) 所示, 其中 $K$ 为一个常数, 如公式 (11) 所示.

$$p_{ij}(i=1,2,\ldots,m;j=1,2)=\frac{r_{ij}}{\sum_{i=1}^{m}r_{ij}}(i=1,2,\ldots,m;j=1,2) \tag{9}$$

$$e_j=-K\sum_{i=1}^{m}p_{ij}\ln p_{ij}(i=1,2,\ldots,m;j=1,2) \tag{10}$$

$$K=\frac{1}{\ln m} \tag{11}$$

(4) 计算权重

将第 $j$ 个指标的权重表示为 $W_j$, 其中 $j$ 为 1 时为属性复杂度权重, $j$ 为 2 时为方法复杂度权重, 计算见公式 (12):

$$W_j(j=1,2)=\frac{1-e_j}{\sum_{j=1}^{2}(1-e_j)}(j=1,2) \tag{12}$$

得到属性复杂度和方法复杂度的权重后, 再通过公式 (7) 得到测试桩复杂度 $SCplx(i,j)$, 通过线性加权得到序列的总体测试桩复杂度 $OCplx(o)$, 即得到类集成测试序列的总体测试代价.

### 2.3 面向类集成测试序列生成的强化学习策略

本文通过强化学习的方法生成类集成测试序列, 目标是进一步降低总体测试桩复杂度. 首先, 将要分析的软件系统视为一组需要进行集成的类的集合; 其次, 保留智能体在路径中执行的动作序列, 作为本文研究问题的候选解决方案; 最后, 从候选方案中选择总体测试桩复杂度最小的动作序列作为类集成测试序列.

如前文第 2.1 节中所说, 强化学习中通过状态空间 $S$ 存储智能体所处的状态, 动作空间 $A$ 存储智能体选择的动作. 假设当前有 $n$ 个类, 那么智能体从初始到选择第 1 个动作有 $n$ 个可能的类可以作为选择项, 即有 $n$ 种可能的状态, 而第 1 个动作到第 2 个动作同样有 $n$ 个可能的选择项. 这样计算至 $n$ 个动作完成, 状态空间所存储的状态最大有 $(n^n+1-1)/(n-1)$ 个, 动作空间存储 $n$ 个动作, 这些动作组成的路径构成一种可能的动作序列, 即本文所求的类集成测试序列.

观察图 5, 假设智能体在第 1 个状态到第 $f$ 个状态中, 共经历了 $n+1$ 个状态, 选择了 $n$ 个动作, 则认为第 $f$ 个状态 $s_f$ 即为最终态, 从 $s_1$ 到 $s_f$ 之间的状态变化函数则如公式 (13) 所示[4], $s'$为状态 $s$ 的下一状态.

$$s'=s_{n\cdot i-n+1+k}\left(i\in\left\{1,2,\ldots,\frac{n^n-1}{n-1}\right\},k\in\{1,2,\ldots,n\}\right) \tag{13}$$

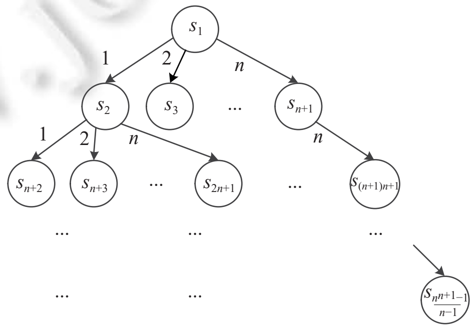



图 5　状态变化图

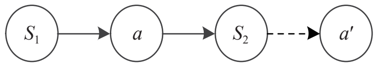
用 $\sigma=(\sigma_0,\sigma_1,\ldots,\sigma_f)$ 表示从最初状态到最终状态 $s_f$ 的状态路径, 其中 $\sigma_0$ 表示初始状态 $s_1$, $\sigma_i=s_1\cup s_2\cup\ldots\cup s_{i-1}$. 智能体按照该路径执行的 $n$ 个动作组成的序列即可以表示为 $a_\sigma$, 即与状态路径对应的动作历史. 如果该路径不包含重复动作, 则可以认为 $a_\sigma$ 即为一条可供选择的类集成测试序列.

强化学习中的奖惩机制是控制智能体探索最佳路径的核心[2], 当智能体探索并集成到的类越优, 其得到奖励值越高. 为了进一步降低总体测试代价, 本文方法结合测试桩复杂度设计奖励函数, 如公式 (14) 所示, 其中 $c$ 为正整数, $Max$ 为提前设定的固定值, 表示最大值:

$$r(\sigma_i)=\left\{\begin{array}{c}-\infty\\ Max-c\times SCplx(a_{\sigma_0},a_{\sigma_1},\ldots,a_{\sigma_{i-1}})\end{array}\right\} \tag{14}$$

智能体经过 $i-1$ 次状态变换到达 $\sigma_i$, 当过程中出现任何不符合要求的情况, 环境会赋予智能体一个最小奖励值, 即惩罚值例如. 过程中任意一个动作类重复出现两次, 即给予这条路径一个最小奖励值$-\infty$, 后面继续探索时可以对这种情况进行避免. 当该路径中没有出现重复类时则可认为此条路径可行, 环境会赋予智能体一个正奖励值. 当到达终态仍没有重复类出现时, 即可以认为已经找到一条备选动作路径并计算其整体测试代价. 假如当前得到的类集成测试序列的测试代价小于前面所得到的所有序列的测试代价, 则可以给予其一个较高的奖励值. 按照这种路径动作历史的顺序进行集成, 可以使生成的类集成测试序列总体测试桩复杂度最小化, 最大程度的节约测试成本.

本文采用的是强化学习中的 Q-学习算法[2], 这种算法融合了蒙特卡洛和动态规划的思想, 无需建立模型, 根据智能体在与环境的交互中产生的状态和选择的动作, 得到一个即时回报的 $Q$ 值, 用 $Q(s, a)$ 表示, 然后根据下一个状态 $s'$选择最大的 $Q(s', a')$ 并将其乘以一个折扣因子, 再加上刚才即时回报的 $Q$ 值得到现在的 $Q$ 值, $Q$ 值更新的过程如图 6 所示.




现实 $Q(s_1, a)=r+\gamma\max Q(s_2, a')$

估计 $Q(s_1, a)$

差值=现实 $Q(s_1, a)$−估计 $Q(s_1, a)$

更新 $Q(s_1, a)=Q(s_1, a)+\alpha\times$差值

图 6 $Q$ 值更新图

图 6 中, $\alpha$ 表示学习率, $r$ 表示智能体在状态 $s$ 下执行动作 $a$ 得到的奖励值, $\gamma$ 表示折扣因子. 计算过程如公式 (15)[4]所示:

$$Q(s,a)=Q(s,a)+\alpha(r+\gamma\max Q(s',a')-Q(s,a)) \tag{15}$$

强化学习是通过探索和利用选择动作的过程, 为了使智能体避免在学习过程中陷入局部最优, 本文进一步增加探索的比例, 在 ε-贪婪方法[2]的基础上, 采取两种选择机制.

(1) 传统的ε-贪婪算法[2]: 以 1−ε 的概率选择当前状态下 Q 值最大对应的动作, 以ε 的概率随机选择动作.

(2) 动态调整ε 算法: 首先以公式 (16) 动态调整ε 的值[13], 其中 *times* 表示训练次数, 然后按照方法 (1) 的步骤继续进行动作选择.

$$\varepsilon=0.15+(0.85-0.15)\times e^{\frac{-1}{(times/5000)}} \tag{16}$$

通过上述 Q-学习算法, 得到智能体从初始到最终状态的路径 $\sigma$, 若所有状态下的所有动作均被访问到, 则智能体得到的 $Q$ 值达到最佳值, 与之对应的状态路径相关联的动作历史, 即为本方法得到的最佳的类集成测试序列.

综上, 本文改进的 Q-学习算法[2]训练智能体的过程如下.

**算法**. 改进的 Q-学习算法.

输入: 状态 $s$, 动作 $a$, 学习率 $\alpha$, 折扣因子 $\gamma$;

输出: 一个类集成测试命令.

```
重复 (对每次训练):
  初始化 s;
  重复 (针对每次训练的每个步骤):
      使用 ε-贪婪策略选择动作 a;
      执行动作 a, 获得奖励 r, 获得状态 s';
      获取更新的 Q 值:
        Q(s,a) = Q(s,a) + α(r + γ max Q(s',a') − Q(s,a))
      s ← s';
   直到 s 为终端为止;
结束
```

### 2.4 实例分析

本文选择 SIR (software-artifact infrastructure repository, 软件工件基础结构存储库)[14]中的 elevator——电梯调度算法进行实例分析, 该程序共包含 12 个类, 934 行代码, 其系统信息如表 1 所示.

表 1 elevator 系统信息

| 编号 | 类名 | 编号 | 类名 |
|---|---|---|---|
| 0 | Building | 6 | ElevatorState |
| 1 | DoorClosedException | 7 | Floor |
| 2 | Elevator | 8 | Logger |
| 3 | ElevatorController | 9 | Person |
| 4 | ElevatorFullException | 10 | PersonState |
| 5 | ElevatorMovingException | 11 | Simulator |

首先, 通过 Soot 对待测系统进行静态分析, 对应表 1 的类名与编号, 根据前文第 1.3 节得到类间依赖关系中的类间属性依赖值和类间方法依赖值, 如表 2 所示.

表 2 elevator 系统的类间依赖关系

类间属性依赖值表

| 类 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | | | 1 | | | 1 | 2 | | | | |
| 1 | | | | | | | | | | | | |
| 2 | | | | 2 | | | 1 | | 1 | 2 | | |
| 3 | | | 4 | | | | 1 | 1 | 1 | 2 | | |
| 4 | | | | | | | | | | | | |
| 5 | | | | | | | | | | | | |
| 6 | | | | | | | | | | | | |
| 7 | | | 2 | 2 | | | | | 1 | 3 | | |
| 8 | | | | | | | | | | | | |
| 9 | 2 | | 2 | | | | | 1 | 1 | | 1 | |
| 10 | | | | | | | | | | | | |
| 11 | | | | | | | | | | | | |

类间方法依赖值表

| 类 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | | 2 | 7 | | | | 2 | | | | |
| 1 | | | | | | | | | | | | |
| 2 | | 2 | | 1 | 1 | 2 | 2 | | 24 | 1 | | |
| 3 | | | 25 | | | | | 15 | 19 | 2 | | |
| 4 | | | | | | | | | | | | |
| 5 | | | | | | | | | | | | |
| 6 | | | | | | | | | | | | |
| 7 | | | 2 | 2 | | | | | 9 | 9 | | |
| 8 | | | | | | | | | | | | |
| 9 | 3 | | 6 | | | | | 10 | 4 | | 2 | |
| 10 | | | | | | | | | | | | |
| 11 | 2 | | | | | | | | | 4 | | |

表 2 左侧表示类间的属性依赖关系, 表 2 右侧表示类间的方法依赖关系, 表中的编号即类编号. 表中纵向编号代表源类, 横向编号代表目标类, 二者之间的交叉值代表它们之间的依赖值, 例如左侧表中 (0, 3)=1, 代表 0 号类 Building 类对 3 号类 ElevatorController 类的属性依赖值为 1; 右侧表中 (0, 2)=2, 则代表 0 号类 Building 类对 2 号类 Elevator 类的方法依赖值为 2, 以此类推. 其中空格表示纵横二类间不存在类间的属性或方法依赖关系.

接下来, 按第 2.3 节改进的 Q-学习算法输入参数值, 对于 elevator 系统, 状态值约有 10 的 13 次方个, 每一轮选择前都要先初始化 $s$, 然后根据随机生成的动作选择概率选择动作, 智能体得到奖励值后进入下一状态, 继而更新 $Q$ 值.

当终止标志为 TRUE 且每个类均被选择到时, 如表 3 中编号为 12 的序列中 0 至 11 这 12 个类均被包含, 环境会给予智能体一个较大的奖励值, 该次选择完成.

智能体探索失败的某一次选择过程如表 4 所示, 当过程中出现对同一个类的二次选择时, 如表 4 中序号为 5 的序列中包含两个 6 号类, 环境会给予智能体一个较低的奖励值, 同样将终止状态设为 TRUE, 结束本轮选择.

假设本文设置的实验次数为两次, 当两次实验完成, 系统从上述两次结果中选择出最大奖励值的动作序列 [8, 6, 10, 9, 2, 3, 1, 4, 5, 7, 11, 0], 即为本次训练所求得的最优类集成测试序列.

## 3 实 验

为验证本文提出的基于强化学习的类集成测试序列生成方法 CITO_RL 的效果, 本节将对从 SIR 网站[14]选取的部分有代表性的程序展开实验. 本文实验运行于 Eclipse 平台, 使用 Java 语言实现.

表 3　elevator 执行一次过程 (成功)

| 序号 | 动作选择概率 | 执行动作 | 奖励值 | 下一状态 | 备选动作 | 终态标志 | 序列 |
|---|---|---|---|---|---|---|---|
| 1 | 0.115170 | 8 | 1 000 | 10 | 6 | FALSE | [8] |
| 2 | 0.211956 | 6 | 1 000 | 116 | 10 | FALSE | [8, 6] |
| 3 | 0.538042 | 10 | 1 000 | 1 392 | 9 | FALSE | [8, 6, 10] |
| 4 | 0.860361 | 9 | 990.36 | 16703 | 2 | FALSE | [8, 6, 10, 9] |
| 5 | 0.304143 | 2 | 995.78 | 200428 | 3 | FALSE | [8, 6, 10, 9, 2] |
| 6 | 0.207753 | 3 | 996.06 | 2405129 | 1 | FALSE | [8, 6, 10, 9, 2, 3] |
| 7 | 0.459753 | 1 | 1 000 | 28861539 | 4 | FALSE | [8, 6, 10, 9, 2, 3, 1] |
| 8 | 0.224424 | 4 | 1 000 | 346338462 | 5 | FALSE | [8, 6, 10, 9, 2, 3, 1, 4] |
| 9 | 0.751222 | 5 | 1 000 | 4156061539 | 7 | FALSE | [8, 6, 10, 9, 2, 3, 1, 4, 5] |
| 10 | 0.160097 | 7 | 1 000 | 49872738465 | 11 | FALSE | [8, 6, 10, 9, 2, 3, 1, 4, 5, 7] |
| 11 | 0.023287 | 11 | 999.51 | 598472861581 | 0 | FALSE | [8, 6, 10, 9, 2, 3, 1, 4, 5, 7, 11] |
| 12 | 0.388568 | 0 | 10000 | 7181674338962 | | TRUE | [8, 6, 10, 9, 2, 3, 1, 4, 5, 7, 11, 0] |

表 4　elevator 执行一次过程 (失败)

| 序号 | 动作选择概率 | 执行动作 | 奖励值 | 下一状态 | 备选动作 | 终态标志 | 序列 |
|---|---|---|---|---|---|---|---|
| 1 | 0.770786 | 6 | 1 000 | 8 | 8 | FALSE | [6] |
| 2 | 0.302418 | 8 | 1 000 | 94 | 1 | FALSE | [6, 8] |
| 3 | 0.649659 | 1 | 1 000 | 1 119 | 4 | FALSE | [6, 8, 1] |
| 4 | 0.687569 | 4 | 1 000 | 13422 | 6 | FALSE | [6, 8, 1, 4] |
| 5 | 0.972340 | 6 | −2147483648 | 161060 | | TRUE | [6, 8, 1, 4, 6] |

### 3.1　参数设计

实验之前需要为 Q-学习算法设置相关参数, 本文的参数设置如表 5 所示. 对于 Q-学习参数, 初始学习率 $\alpha$ 设为 0.9, 在训练过程中线性降低, 直至变为 0.01, 这个过程有助于智能体更注重未来的奖励. 折扣率 $\gamma$ 表示未来奖励对目前状态的影响程度, 设置为较大的值 0.9 也助于智能体向未来奖励更大化靠拢. 考虑到本文选择的实验对象中最大的程序类数不超过 70, 以及本文方法的实验环境, 选择 20 万次的训练次数既可以最大程度的对实验对象进行训练获取最优结果, 也不至于使实验设备负载过高. 探索率ε的取值是实验得到的结果, 同时为了进一步平衡强化学习中的探索和利用, 本文选择了多种动作选择方式进行对比实验 (详见第 3.3 节问题 2). 经验证, 对于本文选择的待测程序, elevator、SPM、ATM、daisy、ANT 与 email-spl 系统取 ε 为 0.8 时效果较好, 其他 3 个系统 BCEL、DNS、notepad_spl 使用动态 ε 取值效果更佳.

表 5　参数设置

| Q-学习参数 | 值 |
|---|---|
| 初始学习率$\alpha$ | 0.9 |
| 折扣率$\gamma$ | 0.9 |
| 训练次数 | $2\times10^5$ |
| 探索率ε | 0.8& 动态ε |
| $c$ | 100 |
| Max | 1 000 |

将强化函数的参数设置 $c$ 设置为 100, 是根据本文重点考虑的测试桩复杂度的取值范围设置的较为平衡的取值, 可以更好地区分得到的动作序列的奖励值. 当智能体当前探索得到的类集成测试序列整体测试代价最小, 智能体的奖励值会被赋予一个 Max 值, 本文中这个值取 1 000. 当完成所有训练次数, 我们找出其中拥有最大总体奖励值的动作序列, 即为运行本方法最终得到的最优类集成测试序列.

### 3.2 实验对象和评价指标

实验对象信息如表 6 所示, 其中 SPM、ATM、ANT、BCEL 和 DNS 这 5 个程序来自 Briand 等人的文献[10], 通常作为研究生成类集成测试序列方法的基准系统, 其他 4 个均为 SIR 网站[14]上的开源程序. 表 6 中的列分别表示选择的程序、对程序的描述、程序中的类数、依赖关系数、环路数和代码的总行数. 选择的程序规模从 12 到 65 不等, 类间依赖关系从 36 到 294 不等, 类间环路从 4 到 416091 不等, 代码行数从 934 到 6 710 不等. 由于实验版本不一, 表 3–表 6 中信息均来源于 Soot 程序分析, 个别系统与 SIR 网站或其他文献有细微差别.

表 6 实验程序信息

| 程序 | 描述 | 类数 | 依赖关系 | 环路数 | 代码行数 |
|---|---|---|---|---|---|
| elevator | 经典电梯调度算法 | 12 | 27 | 23 | 934 |
| SPM | 保安巡逻监控系统 | 19 | 72 | 1 178 | 1 198 |
| ATM | 自动取款机模拟系统 | 21 | 67 | 30 | 1 390 |
| daisy | Unix下的网络文件系统 | 23 | 36 | 4 | 1 148 |
| ANT | 基于Java平台的集成包管理系统 | 25 | 83 | 654 | 4 093 |
| email_spl | 电子邮件工具 | 39 | 63 | 38 | 2 276 |
| BCEL | 创建、分析、操作Java类文件的插桩工具 | 45 | 294 | 416091 | 3 033 |
| DNS | 网络域名服务系统 | 61 | 276 | 16 | 6 710 |
| notepad_spl | 源代码编辑器 | 65 | 142 | 227 | 2 419 |

为了更精确地评价本文方法的实验效果, 本文采用改进的权值计算法, 即, 使用熵权法[12]计算权值, 进而得到测试桩复杂度, 并用测试桩复杂度来衡量生成的类集成测试序列的优劣, 总体测试桩复杂度越小则类集成测试序列越优, 计算方法如第 2.2 节.

### 3.3 实验结果及分析

为了便于更加直观地表现本文提出的面向类集成测试序列生成的强化学习方法 CITO_RL 的效果, 本节将进行实验, 并和经典的图论算法与搜索算法的实验结果做对比. 为了便于深入地以及有针对性地评估本文方法的效果, 本节针对实验提出了以下几个问题.

问题 (1): 使用熵权法计算权值表现如何?

问题 (2): 动作选择参数的设计是否合理?

问题 (3): 使用强化学习的方法生成类集成测试序列的效果如何?

为了回答以上 3 个问题, 本文对表 6 中的程序展开实验.

本文实验基于强化学习方法生成类集成测试序列, 其参数设置如第 3.1 节所示. 针对每种算法的每次实验都有一定的随机性的问题, 本文对每次实验重复运行多次, 并对结果取平均值, 来与其他算法的结果进行比较.

回答问题 (1) 使用熵权法计算权值表现如何.

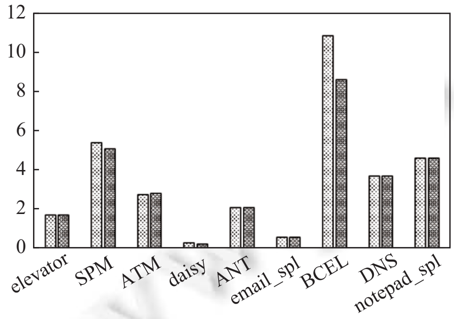
针对第一个问题, 我们进行了两组对比实验. 首先, 使用本领域最常使用的方法, 即 Briand 等人[9]提出的取 0.5 作为属性和方法复杂度的权值的方法; 然后, 再使用熵权法计算权值对实验对象进行对比实验. 实验中除属性和方法复杂度的权值之外, 其余参数均与表 5 一致. 两实验均运行 10 次对得到的数据取平均值, 其结果如表 7 所示, 其中, OCplx 表示生成的类集成测试序列的总体测试桩复杂度, StubsG 表示过程中构建的通用测试桩数量, SubsS 表示构建的特效测试桩数量. 表中较小值的结果均以粗体标注, 相同值未作标记.

观察表 7, 9 个程序中, 在总体测试桩复杂度方面, 熵权法得到的结果有 7 个总体测试桩复杂度较小, 另外 2 个相差也不足 0.1. 在构建的测试桩数目方面, 权值取 0.5 方法的结果中仅 email_spl 和 BCEL 系统通用测试桩构建最少, 仅 email_spl 系统特效测试桩构建最少, 且与熵权法得到的结果均只相差 1 个测试桩; 而熵权法得到的结果中, SPM 程序构建的特效测试桩最少, 与权值取 0.5 得到的结果对比, 可少构建 5 个特效测试桩, 其余二者持平.

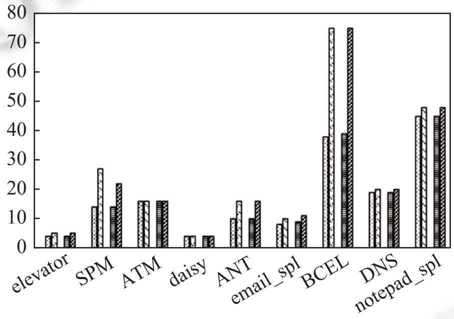
表 7 权值取 0.5 与熵权法计算权值

| 程序 | 权值均为0.5 | | | 熵权法计算权值 | | |
|---|---|---|---|---|---|---|
| | OCplx | StubsG | StubsS | OCplx | StubsG | StubsS |
| elevator | 1.713 | 4 | 5 | **1.711** | 4 | 5 |
| SPM | 5.382 | 14 | 27 | **5.075** | 14 | **22** |
| ATM | **2.746** | 16 | 16 | 2.810 | 16 | 16 |
| daisy | 0.292 | 4 | 4 | **0.219** | 4 | 4 |
| ANT | 2.079 | 10 | 16 | **2.063** | 10 | 16 |
| email_spl | 0.566 | **8** | 10 | **0.562** | 9 | 11 |
| BCEL | 10.862 | **38** | 75 | **8.612** | 39 | 75 |
| DNS | 3.692 | 19 | 20 | **3.689** | 19 | 20 |
| notepad_spl | **4.610** | 45 | 48 | 4.618 | 45 | 48 |

图 7 表示属性和方法复杂度的权值均取 0.5 和使用熵权法计算权值得到的总体测试桩复杂度 OCplx 与构建的测试桩数量 Stubs 的情况, 通过柱形图可以直观地看出无论是总体测试桩复杂度还是所需构建的测试桩数量, 熵权法得到的结果都更有优势.



(a) 测试桩复杂度 OCplx



(b) 测试桩数目 Stubs

图 7 权值取 0.5 与熵权法计算权值

综上所述, 本文提出的基于熵权法计算属性和方法复杂度权重的策略, 总体上优于 Briand 等人[9]权值取 0.5 的策略.

回答问题 (2) 动作选择参数的设计是否合理.

强化学习算法着重于两部分, 分别是奖励函数的设计和动作选择策略的设置. 因此, 动作选择参数的取值在强化学习算法中处于重要地位. 为了进一步降低选择参数对本实验的影响, 本节就该参数的取值做了多次对比实验, 使参数分别取 0、0.3、0.5、0.7、0.9、1 以及动态变化值, 见公式 (16). 分析得到的实验结果, 观察参数取值为多少时效果更优. 其中 OCplx 表示生成类集成测试序列的总体测试桩复杂度, StubsG 表示生成通用测试桩的数量, StubsS 表示生成特效测试桩的数量.

由于实验中 ε 取值 0.7 和 0.9 时得到的前 3 个指标基本没有差别, 因此为了进一步探究, 寻找最优的 ε 取值, 本实验增加了取值 0.8 和 0.85 的情况. 以 ANT 系统为例, 其结果如表 8 所示.

根据表 8 可以得到, 当 ε 取 0.8 时, 测试 ANT 系统得到的 3 个指标结果值均为最低, 即得到最优结果. 图 8 可以更直观的表现 ε 取值对 ANT 系统生成的类集成测试序列中各指标的影响. 观察发现, 当 ε 取 0.8 时总体测试桩复杂度、所需建立的通用测试桩数目以及特效测试桩数目均为最低值, 即测试 ANT 系统时 ε 取值为 0.8 效果最好. 经实验, 其他 8 个程序中 elevator、SPM、ATM、daisy、email-spl 同 ANT 系统, ε 取值为 0.8 时效果更佳, 其他 3 个系统 BCEL、DNS、notepad_spl 使用动态 ε 取值效果更优.

表 8 ε 取值对测试 ANT 系统的影响

| ε | 0.0 | 0.3 | 0.5 | 0.7 | 0.8 | 0.85 | 0.9 | 1.0 | 动态 |
|---|---|---|---|---|---|---|---|---|---|
| OCplx | 7.91 | 3.83 | 2.40 | 2.11 | **2.00** | 2.04 | 2.01 | 2.72 | 2.72 |
| StubsG | 14 | 15 | 12 | 11 | **10** | **10** | 11 | 11 | 11 |
| StubsS | 44 | 26 | 17 | 17 | **16** | **16** | 17 | 19 | 19 |

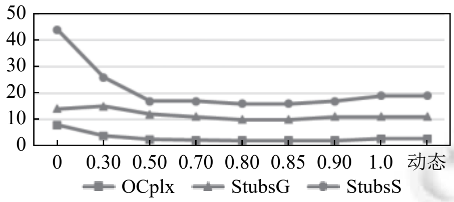



图 8 ε 取值对测试 ANT 系统的影响

综上, 本文动作选择参数的设计方式更可以适应不同系统需求, 更为合理.

回答问题 (3) 使用强化学习的方法生成类集成测试序的效果如何.

为了比较实验效果, 本节针对本文方法对 9 个程序进行了 30 次实验并取平均值, 并分别选择了基于图论与基于搜索的各 3 种算法进行对比, 其中图论的三种算法分别是 Le Traon 等人[15]的方法、Tai 和 Daniels[16]的方法以及 Briand 等人[9]的方法, 基于搜索的算法分别是 Briand 等人[10]提出的遗传算法 GA、王正山等人[17]的随机交互算法 RIA 以及张艳梅等人[18]的粒子群算法 PSO, 其中图论部分的数据取自文献 [19], 其他对比方法参数设置与各文献中一致.

表 9 展示了实验得到的总体测试桩复杂度 OCplx 的数据对比情况, 其中每个程序使用不同方法对比得到的最小值均用粗体表示出. 可以看到 9 个程序中有 4 个程序由 CITO_RL 方法所生成的总体测试桩复杂度最低; 由 Briand 等人的图论方法[9]生成的程序中有 2 个程序总体测试桩复杂度最低, 分别是 BCEL 和 DNS; 由王正山等人的 RIA 算法[17]生成的程序中有 2 个程序总体测试桩复杂度最低; 由 PSO 算法[18]生成的程序中只有 SPM 的总体测试桩复杂度最低.

表 9 总体测试桩复杂度对比

| 程序 | 图论 | | | 搜索 | | | 强化学习 |
|---|---|---|---|---|---|---|---|
| | Le Traon文献[15] | Tai&Daniels文献[16] | Briand文献[9] | GA文献[10] | RIA文献[17] | PSO文献[18] | CITO_RL |
| elevator | — | — | — | 2.03 | 2.04 | 2.87 | **1.72** |
| SPM | 8.4 | 8.08 | 5.82 | 3.5 | 3.48 | **3.02** | 5.26 |
| ATM | 3.37 | 2.99 | 2.7 | 3.09 | **2.43** | 2.59 | 2.81 |
| daisy | — | — | — | 0.58 | 0.32 | 0.92 | **0.22** |
| ANT | 3.72 | 3.87 | 3.31 | 2.13 | 2.23 | 2.32 | **2.08** |
| email_spl | — | — | — | 0.74 | 0.66 | 1.02 | **0.59** |
| BCEL | 8.23 | 8.68 | **5.81** | 9.7 | 8.71 | 8.61 | 8.58 |
| DNS | 5.02 | 4.63 | **1.51** | 5.51 | 4.33 | 5.81 | 3.7 |
| notepad_spl | — | — | — | 1.96 | **1.92** | 4.68 | 4.67 |

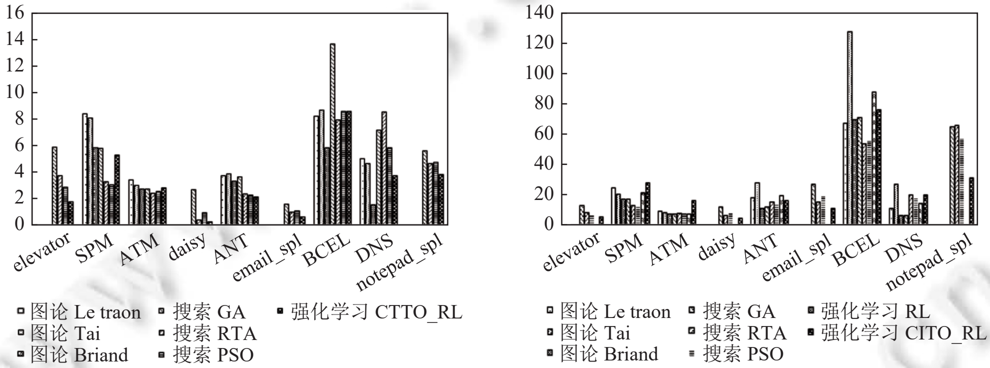
表 10 中展示了实验得到的各方法生成类集成测试序列过程中所需构建的测试桩数目 StubsS 的数据对比情况. 同样, 其中对比得到的每个程序需构建的最小测试桩数目用粗体标出.

观察图 9 可得, 在 9 个程序中, 使用强化学习方法生成类集成测试序列能构建最低测试桩数目的占据 4 个, 使用 Briand 等人图论方法[9]生成类集成测试序列能构建最低测试桩数目的有 3 个, 使用 RIA 随机交互算法[17]生成类集成测试序列能构建最低测试桩数目的有 1 个, 使用 PSO 粒子群算法[18]生成类集成测试序列能构建最低测试桩数目的有 1 个. 表 10 中增加了与只考虑测试桩数目的基于强化学习生成类集成测试序列的方法[4]RL 的对比,

其中, 有数据的 5 个程序中, 本文方法所需构建的测试桩数目有 2 个少于 RL 方法得到的结果. 虽然 SPM 和 DNS 通过本文方法得到的效果没有文献 [4] 的效果好, 但是, 本文方法着重于以测试桩复杂度为评价指标, 因此, 对比 RL 方法, CITO_RL 方法不一定会增加测试代价.

表 10 测试桩数目对比

| 程序 | 图论 | | | 搜索 | | | 强化学习 | |
|---|---|---|---|---|---|---|---|---|
| | Le Traon 文献[15] | Tai&Daniels 文献[16] | Briand文献[9] | GA文献[10] | RIA文献[17] | PSO文献[18] | RL文献[4] | CITO_RL |
| elevator | — | — | — | 8 | 8 | 6 | — | **5** |
| SPM | 25 | 20 | 17 | 13 | 14 | **12** | 21 | 28 |
| ATM | 9 | 8 | **7** | 11 | 9 | 8 | 7 | 16 |
| daisy | — | — | — | 7 | 5 | 8 | — | **4** |
| ANT | 18 | 28 | **11** | 14 | 13 | 14 | 19 | 16 |
| email_spl | — | — | — | 15 | 14 | 19 | — | **11** |
| BCEL | 68 | 128 | 70 | 63 | **51** | 55 | 88 | 77 |
| DNS | 11 | 27 | **6** | 33 | 27 | 17 | 14 | 20 |
| notepad_spl | — | — | — | 55 | 49 | 57 | — | **31** |




(a) 测试桩复杂度 OCplx (b) 测试桩数目 StubsS

图 9 实验结果对比

仍以 elevator 系统为例, 该系统在 3 种搜索方法与本文方法得到的结果作比较时, 本文方法得到的总体测试桩复杂度最低, 所需测试桩数目也最少, GA 方法[10]和 RIA 方法[17]的总体测试桩复杂度稍次之, 但是所需测试桩数目最多. 取某一次使用四种方法分别运行 elevator 系统得到的实验结果进行分析, 如表 11 所示, 其中, Order 表示得到的类集成测试序列, OCplx 表示总体测试桩复杂度, StubsS 表示所需构建的特效测试桩数目.

表 11 elevator 系统的结果对比

| 方法 | Order | OCplx | StubsS |
|---|---|---|---|
| GA文献[10] | [4, 10, 1, 5, 8, 6, 7, 2, 3, 11, 0, 9] | 2.18 | 8 |
| RIA文献[17] | [4, 8, 10, 5, 7, 6, 1, 2, 9, 3, 11, 0] | 2.16 | 8 |
| PSO文献[18] | [8, 6, 10, 4, 1, 5, 2, 7, 3, 0, 9, 11] | 3.02 | 6 |
| CITO_RL | [8, 6, 10, 4, 1, 5, 4, 2, 7, 3, 0, 11] | 1.57 | 5 |

对于 elevator 系统, 进一步分析得到每种方法所建立的测试桩如表 12 所示, 其中 StubsS 表示方法运行过程中构建的测试桩的具体信息, ACplx 表示生成类集成测试序列的属性复杂度, MCplx 表示方法复杂度. 例如“7<--9(3, 6)”表示为类 7 构建模拟类 9 的测试桩, 该步骤涉及到的属性复杂度是 3、方法复杂度是 6.

表 12 elevator 系统的测试桩对比

| 指标 | GA 文献[10] | RIA 文献[17] | PSO 文献[18] | CITO_RL |
|---|---|---|---|---|
| StubsS | 7<--9(3, 9)<br>7<--2(2, 2)<br>7<--3(2, 2)<br>2<--9(2, 1)<br>2<--3(2, 1)<br>3<--9(2, 2)<br>11<--0(0, 2)<br>11<--9(0, 4) | 7<--9(3, 9)<br>7<--2(2, 2)<br>7<--3(2, 2)<br>2<--9(2, 1)<br>2<--3(2, 1)<br>9<--0(2, 3)<br>9<--3(0, 0)<br>11<--0(0, 2) | 2<--3(2, 1)<br>2<--6(1, 2)<br>2<--9(2, 1)<br>7<--3(2, 2)<br>7<--9(3, 9)<br>3<--9(2, 2) | 2<--3(2, 1)<br>7<--3(2, 2)<br>9<--0(2, 3)<br>9<--2(2, 6)<br>9<--7(1, 10) |
| Acplx | 13 | 13 | 12 | 9 |
| MCplx | 23 | 22 | 17 | 22 |

观察表 12 发现, GA 方法[10]和 RIA 方法[17]为 elevator 程序构建的测试桩数目涉及到的类基本一致, 涉及到的属性依赖关系数相同; PSO 方法[18]构建的测试桩虽少, 但其为 2 号类模拟构建了 6 号类, 而 6 号类和 0 号类、2 号类、3 号类均存在属性依赖关系, 导致总体测试桩复杂度高于 GA[10]和 RIA 方法[17], 所以测试桩数目更少并不能保证总体测试桩复杂度也越小. 本文方法为 elevator 系统建立的测试桩数目最少, 所涉及到的属性复杂度最低, 进一步证实了在该系统上的优势.

对于 SPM 系统, 搜索方法无论是最终生成序列的总体测试桩复杂度还是所需构建的测试桩数目均为最优. 因为 SPM 系统相比其他程序而言环路更多, 仅次于类数最多的 notepad_spl, 有 1 178 个, 而强化学习并不着重于破环, 更重视在不断的学习中让智能体自己找到一条更合适的类集成测试序列. 本实验选择的训练次数是 20 万次, 当训练次数达到 50 万次时, 本文方法得到的总体测试桩复杂度为 4.9334. 因此针对 SPM 系统, CITO_RL 算法得到的总体测试桩复杂度仍有降低的空间, 所需构建的测试桩数目也同样如此.

对于 ATM 系统, RIA 方法[17]生成的类集成测试序列的总体测试桩复杂度最低, Briand 等人的图论算法[9]所需构建的测试桩数目最少. 结合两个指标观察, 本文方法 CITO_RL 为 ATM 构建的测试桩数量是最低数目的两倍之多, 但是总体测试桩复杂度却只比最低数目多出约 0.4. 可以看到, CITO_RL 方法容易忽略对于测试桩数量这一因素的考虑, 倾向于选择能使总体测试桩复杂度更低的类进行集成, 这与本文方法以总体测试桩复杂度为评价指标相吻合.

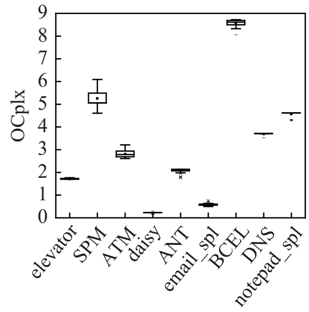
对于 BCEL 系统, CITO_RL 得到的总体测试桩复杂度在 7 个方法中排名第三, 构建测试桩数目方面排名第六, 两个指标上表现均不理想. 针对该系统, 总体表现最好的是 Briand 等人的图论算法[9]和 RIA 随机交互算法[17], 图论算法倾向于根据对象关系图破环, 先去除复杂的依赖关系, 搜索算法也倾向于为更复杂的依赖关系构建更少的测试桩, 而 BCEL 系统拥有 45 个类, 294 个依赖关系以及 40 多万个环路, 并且较多环路都属于同一个 SCC 内, 对于这样复杂的系统, 智能体训练时更容易陷入局部最优, 从而导致效果没有其他方法好.

对于 DNS 系统, 虽然类的数目比 BCEL 系统还要多, 但是类间依赖关系与环路数量均少于 BCEL 系统, 因此 CITO_RL 的总体测试桩复杂度为第二. 表现最好的是 Briand 等人的图论算法[9], 这种算法可以凭借更少的数量构建出复杂度更低的测试桩, 适合于类数目多但相对来说没那么复杂的系统. 因此本文算法在 DNS 系统上仅略逊于 Briand 等人的算法[9], 但优于其他算法, 整体表现尚可.

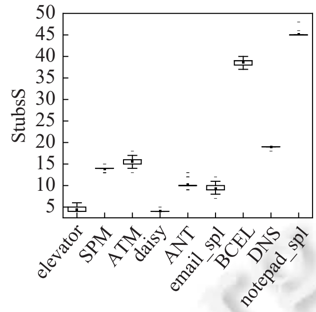
对于 notepad_spl 系统, 总体测试桩复杂度方面 RIA 随机交互算法[17]结果较优, 所需构建的测试桩数目方面本文方法结果更好, 可见本文方法更倾向于构建复杂度更高的测试桩来应对此类系统. 同 DNS 系统一样, notepad_spl 系统类数较多, 依赖关系却比 BCEL 和 DNS 还要少, 但是由于缺少图论部分的数据, 对于该系统, 无法得知图论与搜索哪种方法效果更好.

图 10(a) 展示了本文方法运行实验程序 30 次得到的总体测试桩复杂度 OCplx 结果的箱型图. 观察可得, 30 次运行实验得到的结果中, SPM、ATM、BCEL 这 3 个系统的波动范围较大, 其他实验程序波动范围较小. 其中, daisy 系统由于其类间关系较为简单: 依赖关系 36 个, 仅高于拥有最小类数的 elevator 系统; 环路数仅 4 个, 为实验

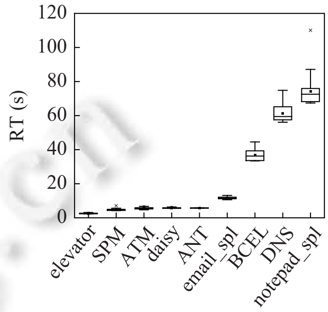
程序中环路数最小的系统. 因此经过 20 万次训练得到的类集成测试序列 30 次结果差别不大. DNS 系统虽然载目前待测系统中拥有第二多类数, 但其环路数为 16, 仅高于 daisy 系统, 结果波动不大; 而 notepad_spl 系统拥有最大类数 65, 类间依赖关系有 142 个, 环路数有 227 个, 均处于中等水平, 但其属性依赖主要集中在 4 号类 featuremodel. BinaryProposition 和 26 号类 smashed. Fonts 之间, 其余类中仅有 57 号类 smashed. Notepad 涉及到两个属性依赖关系, 因此训练得到的实验结果范围波动不大.



(a) 测试桩复杂度 OCplx



(b) 测试桩数目 Stubss



(c) 运行时间 RT

图 10 箱型图

图 10(b) 展示了本文方法经过 30 次实验得到的所需建立的测试桩数量 StubsS 的箱形图, 观察得到其与总体测试桩复杂度得到的箱型图大致相同. 图 10(c) 表示 30 次运行时间 RT 的箱型图, 以 s 为单位, 观察可得, 程序的运行时间会随着类数的增加而增加.

图论方法首先进行破环, 继而生成类集成测试序列, 运行时间大多不过百毫秒, 而本文方法的关键在于智能体依据奖励函数进行的探索, 在训练过程中并不存在破环的行为, 因此, 在时间上, 两类方法并不属于同一量级, 且由于文献 [19] 未列出各图论方法的运行时间, 在此不做比较. 如表 13 所示, 本文在选择与 3 种基于搜索的方法进行运行时间的比较时, RIA 方法[17]在时间上较占优势, 本文的强化学习方法运行时间较久, 其原因是为了均衡实验对象, 方便智能体进行学习, CITO_RL 获得每个结果都需要经过 20 万次训练, 而搜索方法的初始种群均为 100, 经过数百次迭代得到类集成测试序列, 因此相对来说, 本文方法在运行时间这一指标上并不具备明显优势.

表 13 运行时间对比 (ms)

| 程序 | 搜索 | | | 强化学习 |
|---|---|---|---|---|
| | GA文献[10] | RIA文献[17] | PSO文献[18] | CITO_RL |
| elevator | 1 390 | **374** | 1 137 | 2 623 |
| SPM | 2 210 | **206** | 2 994 | 4 739 |
| ATM | 3 546 | **387** | 2 757 | 5 634 |
| daisy | 3 147 | **522** | 1 803 | 5 856 |
| ANT | 7 165 | **377** | 3 655 | 5 796 |
| email_spl | 10 260 | **1 192** | 3 268 | 11 675 |
| BCEL | 22 412 | **1 809** | 10 645 | 36 804 |
| DNS | 20 495 | **3 509** | 8 644 | 61 382 |
| notepad_spl | 58 518 | **4 815** | 8 330 | 74 094 |

为验证方法的显著性, 我们将基于图论和基于搜索的方法与本文方法得到的各项指标进行统计分析. 每种方法运行 30 次, 采用非参数检验中的曼惠特尼 U 检验, 置信度为 95%, 统计过程可见 Github 链接 (https://github.com/Dia-cat/Data/blob/dccda06b00c2ad4c21b383939460c0c864b9098b/README.md). 若 $P$<0.05, 认为两种算法在相关指标上有显著差异.

本文方法与基于图论方法进行统计分析得到的对比结果如表 14 所示, 其中, "有显著性差异 (好)"表示对于该

系统, 与该方法相比, 本文提出的方法具有显著性优势, “有显著性差异 (差)”表示本文方法有显著性劣势, 对于 ATM 系统, 本文方法与 Briand 等人的图论方法[9]渐进显著性 (双尾) 为 0.633, 表明对于该系统, 两种方法得到的结果并无显著差异.

表 14 与图论方法显著性效果对比

| 程序 | 方法 | OCplx(*P*) | StubsS(*P*) |
|---|---|---|---|
| SPM | Le Traon文献[15] | 有显著差异(好) | 有显著差异(差) |
| | Tai&Daniels文献[16] | 有显著差异(好) | 有显著差异(差) |
| | Briand文献[9] | 有显著差异(好) | 有显著差异(差) |
| ATM | Le Traon文献[15] | 有显著差异(好) | 有显著差异(差) |
| | Tai&Daniels文献[16] | 有显著差异(好) | 有显著差异(差) |
| | Briand文献[9] | 无显著差异(0.633) | 有显著差异(差) |
| ANT | Le Traon文献[15] | 有显著差异(好) | 有显著差异(好) |
| | Tai&Daniels文献[16] | 有显著差异(好) | 有显著差异(好) |
| | Briand文献[9] | 有显著差异(好) | 有显著差异(差) |
| BCEL | Le Traon文献[15] | 有显著差异(差) | 有显著差异(差) |
| | Tai&Daniels文献[16] | 有显著差异(好) | 有显著差异(好) |
| | Briand文献[9] | 有显著差异(差) | 有显著差异(差) |
| DNS | Le Traon文献[15] | 有显著差异(好) | 有显著差异(差) |
| | Tai&Daniels文献[16] | 有显著差异(好) | 有显著差异(好) |
| | Briand文献[9] | 有显著差异(差) | 有显著差异(差) |

在总体测试桩复杂度 OCplx 方面, 15 个结果中有 11 个显示本文方法具有显著性优势, 对于 BCEL 系统, 对比 Le Traon 等人的方法[15]与 Briand 等人的图论方法[9]本文方法结果显著性较差, 这是由于 BCEL 系统环路众多, 且较多环路同处于一个强连通分量内, 依赖关系较为复杂, 而 3 种图论方法中, Tai 等人[16]基于根据聚合和继承关系获得主要顺序, 根据关联关系确定次要顺序, Le Traon 等人[15]则基于识别依赖图中的强连通分量进行排序, Briand 等人[9]使用权重并利用强连通分量的最小环路数对关联关系进行标识, 后两种图论方法通过删除系统中的弱依赖关系优先打破强连通分量, 相比于在训练过程找最优序列的强化学习算法, 在 BCEL 这类系统上更占优势. DNS 和 BCEL 均属于弱依赖关系较多的系统, 其中 DNS 的 276 个依赖关系中有 234 个弱依赖关系, BCEL 的 294 个依赖关系中有 244 个弱依赖关系, 对比 Briand 等人的图论方法[9]本文方法效果显著性较差, 这是由于 Briand 等人的图论方法[9]融合了 Tai 等人[16]和 Le Traon 等人[15]的方法, 旨在打破具有关联关系最多的环路, 而本文方法追求当下的测试桩复杂度更低, 很可能在训练过程中先打破聚合或继承等强依赖关系, 导致与 Briand 等人的图论方法[9]相比本文方法显著性效果更差.

在所需构建测试桩数目 StubsS 方面, 对于所有待测系统, 本文方法仅有 4 个结果较好, 不具备优势, 这是由于这 3 种图论算法的目标都是尽量减小测试桩数目, 而本文方法是以总体测试桩复杂度为评价指标, 倾向于构建复杂度更低的测试桩, 造成测试桩数目没有最小化, 并且由于打破不同的依赖关系需要构建不同的特效测试桩, 而构建不同的测试桩所需的复杂度往往也不同, 因此构建更多测试桩的类集成测试序列可能比构建测试桩数目少的序列所花费的总体复杂度更低. 即, 相比于测试桩数目来说, 总体测试桩复杂度的衡量结果准确性更高.

表 15 为本文方法与基于搜索的方法进行对比得到的显著性效果. 其中, 在 OCplx 方面, 有 3 个结果无显著性差异, 7 个结果本文方法显著性较差, 其余 17 个本文方法均具有明显优势. 对于 SPM、ATM 和 notepad_spl 系统, 3 种基于搜索的方法均比本文方法显著性效果好或与本文方法无显著差异. 分析发现, 这是由于 SPM、ATM 和 notepad_spl 系统均有较多关系集中依赖到一个或两个类上, 如 SPM 中的 2 号类 CheckpointController 有 16 个入度和 2 个出度, 8 号类 CheckpointReferences 有 15 个入度和 3 个出度, 并且这些入度较多的类在 3 种搜索算法生成的最终序列中排序靠前, 能够被优先测试, 在一定程度上降低了总体测试桩复杂度. ATM 中的 7 号类 ATM、 notepad_spl 的 21 号类 Smashed. Actions 以及 57 号类 Smashed. Notepad 也是如此. 在 StubsS 方面, 本文方法有 13 个结果具有显著性优势, 同样, 由于本文方法并不着重考虑测试桩数目, 在此不做进一步分析. 在运行时间 RT 方面, 本文提出的面向类集成测试序列生成的强化学习方法劣势明显. 软件系统类的数量越多, 就越需要大量的训练集才能获得较好的结果, 但同时这也导致本文方法收敛速度较慢.

表 15　与搜索方法显著性效果对比

| 程序 | 方法 | OCplx(*P*) | StubsS(*P*) | RT(*P*) |
|---|---|---|---|---|
| elevator | GA文献[10] | 有显著差异(好) | 有显著差异(好) | 有显著差异(差) |
| | RIA文献[17] | 有显著差异(好) | 有显著差异(好) | 有显著差异(差) |
| | PSO文献[18] | 有显著差异(好) | 有显著差异(好) | 有显著差异(差) |
| SPM | GA文献[10] | 有显著差异(差) | 有显著差异(差) | 有显著差异(差) |
| | RIA文献[17] | 有显著差异(差) | 有显著差异(差) | 有显著差异(差) |
| | PSO文献[18] | 有显著差异(差) | 有显著差异(差) | 有显著差异(差) |
| ATM | GA文献[10] | 无显著差异(0.196) | 有显著差异(差) | 有显著差异(差) |
| | RIA文献[17] | 有显著差异(差) | 有显著差异(差) | 有显著差异(差) |
| | PSO文献[18] | 有显著差异(差) | 有显著差异(差) | 有显著差异(差) |
| daisy | GA文献[10] | 有显著差异(好) | 有显著差异(好) | 有显著差异(差) |
| | RIA文献[17] | 有显著差异(好) | 有显著差异(好) | 有显著差异(差) |
| | PSO文献[18] | 有显著差异(好) | 有显著差异(好) | 有显著差异(差) |
| ANT | GA文献[10] | 无显著差异(0.221) | 有显著差异(差) | 有显著差异(好) |
| | RIA文献[17] | 有显著差异(好) | 有显著差异(差) | 有显著差异(差) |
| | PSO文献[18] | 有显著差异(好) | 有显著差异(差) | 有显著差异(差) |
| email_spl | GA文献[10] | 有显著差异(好) | 有显著差异(好) | 有显著差异(差) |
| | RIA文献[17] | 有显著差异(好) | 有显著差异(好) | 有显著差异(差) |
| | PSO文献[18] | 有显著差异(好) | 有显著差异(好) | 有显著差异(差) |
| BCEL | GA文献[10] | 有显著差异(好) | 有显著差异(差) | 有显著差异(差) |
| | RIA文献[17] | 有显著差异(好) | 有显著差异(差) | 有显著差异(差) |
| | PSO文献[18] | 有显著差异(好) | 有显著差异(差) | 有显著差异(差) |
| DNS | GA文献[10] | 有显著差异(好) | 有显著差异(好) | 有显著差异(差) |
| | RIA文献[17] | 有显著差异(好) | 无显著差异(0.630) | 有显著差异(差) |
| | PSO文献[18] | 有显著差异(好) | 有显著差异(差) | 有显著差异(差) |
| notepad_spl | GA文献[10] | 有显著差异(差) | 有显著差异(好) | 有显著差异(差) |
| | RIA文献[17] | 有显著差异(差) | 有显著差异(好) | 有显著差异(差) |
| | PSO文献[18] | 无显著差异(0.139) | 有显著差异(好) | 有显著差异(差) |

总体来说, 本文提出的面向类集成测试序列生成的强化学习方法在总体测试桩复杂度方面优势显著; 在构建测试桩的数目方面, 与基于图论方法相比较差, 与基于搜索方法相比略有优势; 在运行时间方面, 与基于搜索方法相比劣势较为明显, 需要在未来进一步优化.

### 3.4　有效性分析

本节将分析影响本文实验结果的潜在因素, 进一步从内外两个方面验证实验结果的有效性.

(1) 外部有效性分析

受各种外部因素限制, 无论时哪种方法都不可能完全的评测所有程序. 为了进一步说明 CITO_RL 方法的可行性, 本文从 SIR——权威的软件工件基础结构存储库中[14]选择了 9 个实验对象, 选择的程序中类的个数范围覆盖 12-65, 类间依赖关系的个数范围覆盖 36–294, 类间环路个数范围覆盖 4–416091, 代码行数范围覆盖 934–6710, 其中更是包含 5 个行业内认可的基准程序, 其各个方法得到的数据均已发表, 更具有代表性, 结果也更有说服力.

(2) 内部有效性分析

本文方法 CITO_RL 的实验正常运行的内部有效性威胁主要在如何确定智能体完成了一次训练. 为了明确这一点, 本方法给出一个终态标志, 初始时终态标志为 FALSE, 当程序中出现重复的类时, 环境会给予智能体一个很小的负值, 告诉它这条序列不能通过, 终止标志置为 TRUE, 此次训练完成; 当程序无重复地走完所有类时, 环境会给予智能体一个较大的正值, 告诉它这可以作为一个备选序列, 终止标志置为 TRUE, 此次训练完成.

为了尽可能避免随机性数据带来的影响, 本实验选择 30 次运行对结果取平均值, 并分别从总体测试桩复杂度和构建的测试桩数目两个方面评测本方法的效果, 也进一步证明了本方法的有效性.

## 4 相关工作

目前国内外研究人员针对类集成测试序列生成这一课题已经进行了大量研究, 提出了很多解决该问题的技术, 主要可分为基于图论与基于搜索这两种类别, 随着人工智能、机器学习等一些新兴技术的崛起, 近年来还增加了基于强化学习生成类集成测试序列的方法.

(1) 基于图论的方法

图可以分为无环图和有环图两类. 针对有环图, 必须先删除关联边以断开环路, 通过这种方式得到无环图后, 再对其进行逆向拓扑排序; 针对无环图, Kung 等人[8,20]最早提出根据对象关系图 (object relation diagram, ORD)[21]生成类集成测试序列的解决方法. ORD 是一个有向图, 他们用其中的顶点表示对象类别, 用边缘表示对象类别之间的关系. 基于 ORD, 他们描述了一种基于类间的强弱关系生成类集成测试序列的策略, 通过优先删除代表弱依赖关系的边, 即关联关系的边, 减小构造测试桩的测试代价. 他们将这种策略应用于面向对象程序的单元测试和集成测试.

后来的研究者提出了多项为有环图的边赋值、依据权值进行破环的方案. Tai 和 Daniels[16]将各关联边的起点入度与终点出度的和作为权值; Tarjan 等人[22]曾提出一种基于深度优先搜索划分强连通组件的方法, Le Traon 等人[15]在此基础上做了改进, 在 ORD 中进行深度优先搜索, 为每个节点赋权值, 从具有最大权重的节点入边开始删除, 直至图中不存在环路. Briand 等人[6,7]又在 Le Traon 等人[15]的算法基础上做了进一步改进, 明确区分特效测试桩与通用测试桩, 重新设计各边权值, 将关联边作为对象且只删除关联边. 国内的高海昌教授[23]分析了 ORD 图中的依赖关系, 改进了类集成测试序列相关算法. 李远杰等人[24]也通过分析类图中存在的依赖关系, 设计出类集成测试序列的计算准则, 有效地减少了测试桩的数目. 高建华等人[25]考虑了程序中可能存在间接依赖关系的模块, 提出了基于对象模式关系图的类集成测试序列生成方法, 有效降低了测试花费的代价.

上述方法都在尽量减少测试桩的数目, 然而衡量测试代价高低的因素并非只有这一个. Briand 等人[9]从类间方法调用以及属性依赖的方向衡量测试代价, 提出了类间的属性复杂度与方法复杂度这两个概念. Hashim 等人[26]通过计算类间交互的方法数目与属性数目测量类间依赖的强度. Abdurazik 和 Offutt[27,28]在 Briand 等人[9]的基础上, 结合方法参数调用和返回值传递, 总结出一套完整的测试桩度量公式. Bansal 等人[29]将测试桩的复杂度与边涉及的环路数相结合, 当破除环路时, 综合考虑到所需构造测试桩的数目以及测试代价两个因素, 保证在较低的测试代价下得到所需的最低的测试桩数目. Steindl 等人[30]在上述基础上, 进一步增加了对测试工作量的评测. 卢炎生、毛澄映等人[31,32]将关联强度、成环权重以及边向因子共同结合起来, 衡量为某个类构建测试桩的复杂度. 姜淑娟等人[11]也对测试桩的复杂度做了新的耦合度量, 并提出了一种度量类间控制耦合度[1]的方法, 可以进一步降低总体测试桩复杂度.

除此之外, Hewett 等人[33]提出一种自底向上的增量式图论算法; 赵玉丽、王莹等人[34,35]相继提出了基于复杂网络解决类集成测试序列问题的方法, 并就类的节点重要性问题做了探讨; 张妙等人[19,36]提出的基于奖惩机制的多级反馈策略以及考虑类的相似性提出的破环方法均进一步拓展了基于图论方法解决类集成测试序列生成问题的广度与深度.

(2) 基于搜索的方法

搜索可分为无信息搜索和有信息搜索, 其中有信息搜索又被称为启发式搜索. 我们涉及的主要是启发式搜索,

这类方法通常将类集成测试序列问题看作多目标优化问题[37], 通过构建不同的模型, 可以把算法分为两种: (1) 线性加权方法; (2) 帕累托模型近似法.

第 (1) 类使用线性加权模型的启发式搜索算法在类集成测试序列确定问题中应用最为广泛. Briand 等人[10,38]和 Hanh 等人[39]均使用了遗传算法得到类集成测试序列的设计, 通过总体测试桩复杂度获取新的适应度函数, 经过数次迭代最终得到合适的类集成测试序列. Borner 等人[40]将属性复杂度、方法复杂度和测试焦点作为优化目标, 利用模拟退火算法生成类集成测试序列.

国内, 王正山等人[41,42]改进了遗传算法与模拟退火算法, 设计了新的类间耦合信息度量法, 利用测试桩代价最小化的原则, 得到新的类集成测试序列. 张艳梅等人[18]提出了使用粒子群算法, 把类集成测试序列映射为一维搜索空间中的粒子, 每个类集成测试序列作为粒子移动, 经过迭代, 找到最优粒子位置, 即过程中花费最小测试代价对应的类集成测试序列. 张悦宁等人[43]在上述基础上对遗传算法与粒子群算法做了进一步优化, 提出了基于梦境粒子群的优化算法, 进一步减小了测试代价.

第 (2) 类基于帕累托模型近似法的启发式搜索算法将属性复杂度与方法复杂度均视作目标函数, 使其满足帕累托最优, 这种方法也在获得类集成测试序列方面取得了不错效果. Cabral 等人[44]设计了基于帕累托模型的蚁群算法, 根据测试代价越低越能吸引蚁群这一选择机制获得类集成测试序列. Vergilio 等人[45]依据禁忌搜索的思想, 通过禁忌列表存储当前已生成过的类集成测试序列, 进而避免重复搜索, 然后依据藐视准则将列表中的较优解存储到候选列表, 再进行进一步筛选. Assunção 等人[46,47]比较了 NSGA-II[48]、SPEA2[49]以及 PAES[50]等 3 种常用的算法, 并将其用于生成类集成测试序列. 这些算法利用精英集策略, 保存每代进化出的优良个体, 通过控制个体间的拥挤距离与分布密度等来维持种群多样性. 张妙等人[51]在上述基础上提出了一种打破环路的启发式方法, 当给定一组已删除的依赖项时, 采用启发式方法搜索一组接近理想的类依赖项集, 这些依赖项与初始化依赖项具有相同或更多的环路数, 但构建的测试桩成本更低.

近十年来, 超启发式算法 (hyper-heuristic) 得到了越来越多的关注. 该算法主要是通过某种高层策略, 管理操纵一组低层的启发式算法, 进而得到一种新的启发式算法. Guizzo 等人[52,53]首先提出了利用超启发式算法获得类集成测试序列. 该算法为每次进化迭代筛选出最优的交叉及变异概率, 直至获得满意的类集成测试序列.

(3) 基于强化学习的方法

当前机器学习是计算机领域研究的热门方向. 研究者发现, 将机器学习中的强化学习应用于解决类集成测试序列问题能得到较好的结果. Czibula 等人[4]首次使用了强化学习方法, 分别以通用测试桩与特效测试桩作为评价方法优劣的指标, 通过 Q-学习算法训练智能体, 记录其行为路径, 并将其视为类集成测试序列. 该方法可以进一步减少生成最优类集成测试序列时所需构建的测试桩数量.

## 5 结束语

针对已有的面向类集成测试序列生成的强化学习研究方法测试代价的评价指标不够精确的问题, 本文以总体测试桩复杂度作为评价指标, 提出了一种面向类集成测试序列生成的强化学习研究方法 CITO_RL. 首先, 结合测试桩复杂度设计奖励函数, 并使用熵权法计算方法复杂度和属性复杂度的权值; 然后, 设计了强化学习中的动作选择策略, 采用了两种动作选择参数方法进行动作的选取, 进一步平衡强化学习中的探索和利用; 最后, 让智能体在环境中训练并最终学习到最优类集成测试序列. 本文选择了 9 个有代表性的实验对象, 分别从总体测试桩复杂度和测试桩数目两个方面与之前方法进行比较, 最终表明本文方法具有一定的有效性.

虽然我们在不同规模的基准程序上验证了本文方法的有效性, 但与真正应用在不同语言的真实大型程序上开展研究仍有一定距离. 同时本文尚未考虑面向对象程序自身的特性, 例如多态性等, 因此, 如何将本文方法和真正的待测程序结合起来并解决这些问题还有待进一步的探索. 此外, 本文提出的基于强化学习的类集成测试序列生成方法与以往的破环方法相比运行时间会随着类数成倍扩张, 未来可以考虑从软硬件两个方面提升运行效率, 软件方面可以采用多智能体进行任务调度进而优化算法; 硬件方面可以通过引入 GPU 并行操作来提高硬件利用率的效率和规模, 进一步降低运行时间.

**References:**

[1] Jiang SJ, Zhang M, Zhang YM, Wang RC, Yu Q, Keung JW. An integration test order strategy to consider control coupling. IEEE Trans. on Software Engineering, 2021, 47(7): 1350–1367. [doi: 10.1109/TSE.2019.2921965]

[2] Zhou ZH. Machine Learning. Beijing: Tsinghua University Press, 2016. 371–397 (in Chinese).

[3] 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]

[4] Czibula G, Czibula IG, Marian Z. An effective approach for determining the class integration test order using reinforcement learning. Applied Soft Computing, 2018, 65: 517–530. [doi: 10.1016/j.asoc.2018.01.042]

[5] Mathur AP. Foundations of Software Testing. 2nd ed., Boston: Addison-Wesley, 2014. 40–41.

[6] Zhang YM, Jiang SJ, Zhang M, Ju XL. Survey of class test order generation techniques for integration test. Chinese Journal of Computers, 2018, 41(3): 670–694 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2018.00670]

[7] Briand LC, Labiche Y, Wang YH. An investigation of graph-based class integration test order strategies. IEEE Trans. on Software Engineering, 2003, 29(7): 594–607. [doi: 10.1109/TSE.2003.1214324]

[8] Kung DC, Gao J, Hsia P, Lin J, Toyoshima Y. Class firewall, test order, and regression testing of object-oriented programs. Journal of Object Oriented Programming, 1995, 8(2): 51–65.

[9] Briand LC, Labiche Y, Wang YH. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. In: Proc. of the 12th Int'l Symp. on Software Reliability Engineering. Hong Kong: IEEE, 2001. 287–296. [doi: 10.1109/ISSRE.2001.989482]

[10] Briand LC, Feng J, Labiche Y. Using genetic algorithms and coupling measures to devise optimal integration test orders. In: Proc. of the 14th Int'l Conf. on Software Engineering and Knowledge Engineering. Ischia: ACM, 2002. 43–50. [doi: 10.1145/568760.568769]

[11] Jiang SJ, Zhang YM, Li HY, Wang QT. An approach for inter-class integration test order determination based on coupling measures. Chinese Journal of Computers, 2011, 34(6): 1062–1074 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2011.01062]

[12] He DY, Xu JQ, Chen XL. Information-theoretic-entropy based weight aggregation method in multiple-attribute group decision-making. Entropy, 2016, 18(6): 171. [doi: 10.3390/e18060171]

[13] Hu XH. Action choice mechanism of reinforcement learning based on adjusted dynamic parameters. Computer Engineering and Applications, 2008, 44(28): 29–31, 48 (in Chinese with English abstract). [doi: 10.3778/j.issn.1002-8331.2008.28.009]

[14] SIR. A repository of software-related artifacts meant to support rigorous controlled experimentation. https://sir.csc.ncsu.edu/portal/index.php

[15] Traon YL, Jéron T, Jezequel JM, Morel P. Efficient object-oriented integration and regression testing. IEEE Trans. on Reliability, 2000, 49(1): 12–25. [doi: 10.1109/24.855533]

[16] Tai KC, Daniels FJ. Test order for inter-class integration testing of object-oriented software. In: Proc. of the 21st Annual Int'l Computer Software and Applications Conf. Washington: UEEE, 1997. 602–607. [doi: 10.1109/CMPSAC.1997.625079]

[17] Wang ZS, Li BX, Wang LL, Meng W, Gong XF. Using coupling measure technique and random iterative algorithm for inter-class integration test order problem. In: Proc. of the 34th Annual Computer Software and Applications Conf. Workshops. Seoul: IEEE, 2010. 329–334. [doi: 10.1109/COMPSACW.2010.64]

[18] Zhang YM, Jiang SJ, Chen RY, Wang XY, Zhang M. Class integration testing order determination method based on particle swarm optimization algorithm. Chinese Journal of Computers, 2018, 41(4): 931–945 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2018.00931]

[19] Zhang M, Jiang SJ, Zhang YM, Wang XY, Yu Q. A multi-level feedback approach for the class integration and test order problem. The Journal of Systems and Software, 2017, 133: 54–67. [doi: 10.1016/j.jss.2017.08.026]

[20] Kung D, Gao J, Pei H, Toyoshima Y, Chen C. A test strategy for object-oriented programs. In: Proc. of the 19th Annual Int'l Computer Software and Applications Conf. Dallas: IEEE, 1995. 239–244. [doi: 10.1109/CMPSAC.1995.524786]

[21] Kung DC, Gao J, Hsia P, Toyoshima Y, Chen C. On regression testing of object-oriented programs. Journal of Systems and Software, 1996, 32(1): 21–40. [doi: 10.1016/0164-1212(95)00047-X]

[22] Tarjan R. Depth-first search and linear graph algorithms. SIMA Journal on Computing, 1971, 1(2): 146–160.

[23] Gao HC, Feng BQ, Li YJ, Zeng M. Improved inter-class integration testing sequence algorithm based on extended ORD. Journal of Chinese Computer Systems, 2007, 28(4): 725–728 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-1220.2007.04.032]

[24] Li YJ, Zhou GZ, Liu FC. Improved inter-class test sequence algorithm. Computer Engineering, 2010, 36(8): 74–75, 82 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-3428.2010.08.026]

[25] Gao JH, Qi LN. Research of the test order generation based on OMRD. Journal of Shanghai Normal University (Natural Sciences), 2008,

37(4): 369–376 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-5137.2008.04.006]

[26] Hashim NL, Schmidt HW, Ramakrishnan S. Test order for class-based integration testing of Java applications. In: Proc. of the 5th Int'l Conf. on Quality Software. Melbourne: IEEE, 2005. 11–18. [doi: 10.1109/QSIC.2005.64]

[27] Abdurazik A, Offutt J. Coupling-based class integration and test order. In: Proc. of the 2006 Int'l Workshop on Automation of Software Test. Shanghai: ACM, 2006. 50–56. [doi: 10.1145/1138929.1138940]

[28] Abdurazik A, Offutt J. Using coupling-based weights for the class integration and test order problem. Computer Journal, 2009, 52(5): 557–570. [doi: 10.1093/comjnl/bxm054]

[29] Bansal P, Sabharwal S, Sidhu P. An investigation of strategies for finding test order during integration testing of object oriented applications. In: Proc. of the Int'l Conf. on Methods and Models in Computer Science. New Delhi: IEEE, 2009. 1–8. [doi: 10.1109/ICM2CS.2009.5397936]

[30] Steindl M, Mottok J. Optimizing software integration by considering integration test complexity and test effort. In: Proc. of the 10th Int'l Workshop on Intelligent Solutions in Embedded Systems, 2012. 63–68.

[31] Lu YS, Mao CY. Method of inter-class test order determination for object-oriented cluster level testing. Mini-micro Systems, 2005, 26(6): 995–999 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-1220.2005.06.028]

[32] Mao CY, Lu YS. Aicto: An improved algorithm for planning inter-class test order. In: Proc. of the 5th Int'l Conf. on Computer and Information Technology. Shanghai: IEEE, 2005. 927–931. [doi: 10.1109/CIT.2005.59]

[33] Hewett R, Kijsanayothin P. Automated test order generation for software component integration testing. In: Proc. of the 2009 IEEE/ACM Int'l Conf. on Automated Software Engineering. Auckland: IEEE, 2009. 211–220.

[34] Zhao YL, Wang Y, Yu H, Zhu ZL. An inter-class integration test order generation method based on complex networks. Journal of Northeastern University (Natural Science), 2015, 36(12): 1696–1700 (in Chinese with English abstract). [doi: 10.3969/j.issn.1005-3026.2015.12.006]

[35] Wang Y, Yu H, Zhu ZL. A class integration test order method based on the node importance of software. Journal of Computer Research and Development, 2016, 53(3): 517–530 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2016.20148318]

[36] Zhang M, Keung JW, Xiao Y, Kabir MA. Evaluating the effects of similar-class combination on class integration test order generation. Information and Software Technology, 2021, 129: 106438. [doi: 10.1016/j.infsof.2020.106438]

[37] Zhang M, Jiang SJ, Zhang YM. Research on multi-objective optimization in class integration test order. Journal of Chinese Computer Systems, 2017, 38(8): 1772–1777 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-1220.2017.08.022]

[38] Briand LC, Feng J, Labiche Y. Experimenting with genetic algorithms to devise optimal integration test orders. In: Khoshgoftaar TM, ed. Software Engineering with Computational Intelligence. The Springer Int'l Series in Engineering and Computer Science. Boston: Springer, 2003. 204–234. [doi: 10.1007/978-1-4615-0429-0_8]

[39] Le Hanh V, Akif K, Le Traon Y, Jézéque JM. Selecting an efficient OO integration testing strategy: An experimental comparison of actual strategies. In: Proc. of the 15th European Conf. on Object-oriented Programming. Budapest: Springer, 2001. 381–401. [doi: 10.1007/3-540-45337-7_20]

[40] Borner L, Paech B. Integration test order strategies to consider test focus and simulation effort. In: Proc. of the 1st Int'l Conf. on Advances in System Testing and Validation Lifecycle. Porto: IEEE, 2009. 80–85. [doi: 10.1109/VALID.2009.30]

[41] Wang ZS, Li BX, Wang LL, and Qiao L. A brief survey on automatic integration test order generation. In: Proc. of the 23rd Int'l Conf. on Software Engineering and Knowledge Engineering. Eden Roc Renaissance, 2011. 254–257.

[42] Wang ZS, Li BX, Wang LL, Li Q. An effective approach for automatic generation of class integration test order. In: Proc. of the 35th Annual Computer Software and Applications Conf. . Munich: IEEE, 2011. 680–681. [doi: 10.1109/COMPSAC.2011.122]

[43] Zhang YN, Jiang SJ, Zhang YM. Approach for generating class integration test sequence based on dream particle swarm optimization algorithm. Computer Science, 2019, 46(2): 159–165 (in Chinese with English abstract). [doi: 10.11896/j.issn.1002-137X.2019.02.025]

[44] da Veiga Cabral R, Pozo A, Vergilio SR. A pareto ant colony algorithm applied to the class integration and test order problem. In: Proc. of the 22nd IFIP WG 6.1 Int'l Conf. on Testing Software and Systems. Natal: Springer, 2010. 16–29. [doi: 10.1007/978-3-642-16573-3_3]

[45] Vergilio SR, Pozo A, Árias JCG, Da Veiga Cabral R, Nobre T. Multi-objective optimization algorithms applied to the class integration and test order problem. Int'l Journal on Software Tools for Technology Transfer, 2012, 14(4): 461–475. [doi: 10.1007/s10009-012-0226-1]

[46] Assunção WKG, Colanzi TE, Pozo ATR, Vergilio SR. Establishing integration test orders of classes with several coupling measures. In: Proc. of the 13th Annual Conf. on Genetic and Evolutionary Computation. Dublin: ACM, 2011. 1867–1874. [doi: 10.1145/2001576.2001827]

[47] Assunção WKG, Colanzi TE, Vergilio SR, Pozo A. A multi-objective optimization approach for the integration and test order problem. Information Sciences, 2014, 267: 119–139. [doi: 10.1016/j.ins.2013.12.040]

[48] Deb K, Pratap A, Agarwal S, Meyarivan T. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. on Evolutionary Computation, 2002, 6(2): 182–197. [doi: 10.1109/4235.996017]

[49] Zitzler E, Laumanns M, Thiele L. SPEA2: Improving the strength pareto evolutionary algorithm. In: Proc. of the Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems, 2001. 95–100.

[50] Knowles JD, Corne DW. Approximating the nondominated front using the Pareto archived evolution strategy. Evolutionary Computation, 2000, 8(2): 149–172. [doi: 10.1162/106365600568167]

[51] Zhang M, Keung J, Xiao Y, Kabir MA, Feng S. A heuristic approach to break cycles for the class integration test order generation. In: Proc. of the 43rd Annual Computer Software and Applications Conf. Milwaukee: IEEE, 2019. 47–52. [doi: 10.1109/COMPSAC.2019.00016]

[52] Guizzo G, Fritsche GM, Vergilio SR, Pozo ATR. A hyper-heuristic for the multi-objective integration and test order problem. In: Proc. of the 2015 Annual Conf. on Genetic and Evolutionary Computation. Madrid: ACM, 2015. 1343–1350. [doi: 10.1145/2739480.2754725]

[53] Mariani T, Guizzo G, Vergilio SR, Pozo ATR. Grammatical evolution for the multi-objective integration and test order problem. In: Proc. of the Genetic and Evolutionary Computation Conf. Denver: ACM, 2016. 1069–1076. [doi: 10.1145/2908812.2908816]

附中文参考文献:

[2] 周志华. 机器学习. 北京: 清华大学出版社, 2016. 371–397.

[3] 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]

[6] 张艳梅, 姜淑娟, 张妙, 鞠小林. 集成测试中的类测试顺序生成技术述评. 计算机学报, 2018, 41(3): 670–694. [doi: 10.11897/SP.J.1016.2018.00670]

[11] 姜淑娟, 张艳梅, 李海洋, 王庆坛. 一种基于耦合度量的类间集成测试序的确定方法. 计算机学报, 2011, 34(6): 1062–1074. [doi: 10.3724/SP.J.1016.2011.01062]

[13] 胡晓辉. 一种基于动态参数调整的强化学习动作选择机制. 计算机工程与应用, 2008, 44(28): 29–31, 48. [doi: 10.3778/j.issn.1002-8331.2008.28.009]

[18] 张艳梅, 姜淑娟, 陈若玉, 王兴亚, 张妙. 基于粒子群优化算法的类集成测试序列确定方法. 计算机学报, 2018, 41(4): 931–945. [doi: 10.11897/SP.J.1016.2018.00931]

[23] 高海昌, 冯博琴, 李远杰, 曾明. 基于扩展ORD图的类间集成测试顺序改进算法. 小型微型计算机系统, 2007, 28(4): 725–728. [doi: 10.3969/j.issn.1000-1220.2007.04.032]

[24] 李远杰, 周国征, 刘凤晨. 一种类间测试顺序改进算法. 计算机工程, 2010, 36(8): 74–75, 82. [doi: 10.3969/j.issn.1000-3428.2010.08.026]

[25] 高建华, 齐丽娜. 基于对象模式关系图的测试顺序生成方法. 上海师范大学学报(自然科学版), 2008, 37(4): 369–376. [doi: 10.3969/j.issn.1000-5137.2008.04.006]

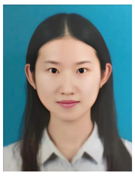
[31] 卢炎生, 毛澄映. 面向对象簇级测试中类间测试序确定方法研究. 小型微型计算机系统, 2005, 26(6): 995–999. [doi: 10.3969/j.issn.1000-1220.2005.06.028]

[34] 赵玉丽, 王莹, 于海, 朱志良. 基于复杂网络的类间集成测试序列生成方法. 东北大学学报(自然科学版), 2015, 36(12): 1696–1700. [doi: 10.3969/j.issn.1005-3026.2015.12.006]

[35] 王莹, 于海, 朱志良. 基于软件节点重要性的集成测试序列生成方法. 计算机研究与发展, 2016, 53(3): 517–530. [doi: 10.7544/issn1000-1239.2016.20148318]

[37] 张妙, 姜淑娟, 张艳梅. 多目标优化类集成测试序列确定问题研究进展. 小型微型计算机系统, 2017, 38(8): 1772–1777. [doi: 10.3969/j.issn.1000-1220.2017.08.022]

[43] 张悦宁, 姜淑娟, 张艳梅. 基于梦境粒子群优化的类集成测试序列生成方法. 计算机科学, 2019, 46(2): 159–165. [doi: 10.11896/j.issn.1002-137X.2019.02.025]



**丁艳茹**(1996－), 女, 博士生, CCF 学生会员, 主要研究领域为软件分析于测试, 强化学习.

**袁冠**(1982－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为数据挖掘, 软件工程.

**张艳梅**(1982－), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为软件分析与测试, 软件缺陷预测.

**王荣存**(1979－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为软件分析与测试, 故障定位.

**姜淑娟**(1966－), 女, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为软件分析与测试, 编译技术.

**钱俊彦**(1973－), 男, 博士, 教授, CCF 高级会员, 主要研究领域为形式化验证, 智能安全, VLSI 容错.