软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(5):2063−2084 [doi: 10.13328/j.cnki.jos.007453] [CSTR: 32375.14.jos.007453]

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

基于动态分析和引力公式的关键类识别*

潘伟丰[1], 杨燕微[1], 杨子江[2], 姜 波[1], 王家乐[1], 杨柏林[1]

[1](浙江工商大学 计算机科学与技术学院, 浙江 杭州 310018)

[2](中国科学技术大学 计算机科学与技术学院, 安徽 合肥 230026)

通信作者: 潘伟丰, E-mail: wfpan@zjgsu.edu.cn; 杨柏林, E-mail: ybl@zjgsu.edu.cn

摘 要: 关键类是理解复杂软件的极佳起点, 不仅有助于优化软件的文档化过程, 也有助于压缩逆向工程得到的类图. 尽管目前已经提出了很多有效的关键类识别方法, 但普遍存在 3 点不足: 1) 现有工作所依赖的软件网络 (由软件元素及其依赖构建的图) 不可避免地会包含软件运行时永远不会 (或极少) 被执行到的软件元素; 2) 基于动态分析构建的软件网络往往不够完整, 会遗漏真正的关键类; 3) 现有工作通常只考虑类之间的直接耦合对类重要性的影响, 而忽视类之间的间接 (非接触) 耦合及邻居节点度分布的多样性对类重要性的影响. 鉴于此, 提出一种融合动态分析和引力公式的关键类识别方法. 首先, 使用静态分析技术构建面向对象软件的类依赖网络 CCN (class coupling network), 用以抽象类及类之间的耦合关系. 其次, 综合考虑 CCN 中类之间"直接和间接的耦合""邻居节点度分布的多样性"等对类重要性的影响, 构建引力熵 *GEN* (gravitational entropy) 度量指标以量化类的重要性. 然后, 按照类的 *GEN* 值对所有类进行降序排列, 从而得到初步的排序结果. 最后, 通过动态分析技术收集运行时类之间真实的交互关系, 进而对初步排序的结果进行优化, 并通过设定阈值来过滤非关键类, 从而得到候选的关键类. 8 个开源 Java 软件上的实验结果表明: 1) 在检查不超过前 15% (或 top-25) 的节点时, 该方法从整体上而言均显著优于其他 11 种对比方法; 2) 使用动态分析对结果进行优化, 有助于显著提升该方法的性能; 3) 耦合类型的不同赋权方式对该方法的性能没有显著影响; 4) 该方法在运行效率上是可以接受的.

关键词: 关键类识别; 软件网络; 引力公式; 熵; 软件度量

中图法分类号: TP311

中文引用格式: 潘伟丰, 杨燕微, 杨子江, 姜波, 王家乐, 杨柏林. 基于动态分析和引力公式的关键类识别. 软件学报, 2026, 37(5): 2063–2084. http://www.jos.org.cn/1000-9825/7453.htm

英文引用格式: Pan WF, Yang YW, Yang ZJ, Jiang B, Wang JL, Yang BL. Key Class Identification Based on Dynamic Analysis and Gravitational Formula. Ruan Jian Xue Bao/Journal of Software, 2026, 37(5): 2063–2084 (in Chinese). http://www.jos.org.cn/1000-9825/7453.htm

Key Class Identification Based on Dynamic Analysis and Gravitational Formula

PAN Wei-Feng[1], YANG Yan-Wei[1], YANG Zi-Jiang[2], JIANG Bo[1], WANG Jia-Le[1], YANG Bai-Lin[1]

[1](School of Computer Science and Technology, Zhejiang Gongshang University, Hangzhou 310018, China)

[2](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China)

Abstract: Key classes are a crucial starting point for understanding complex software, contributing to the optimization of documentation and the compression of reverse-engineered class diagrams. Although many effective key class identification methods have been proposed, three major limitations remain: 1) software networks, which are graphs representing software elements and their dependencies, often include elements that are never or rarely executed at runtime; 2) networks constructed through dynamic analysis are frequently incomplete, potentially omitting truly key classes; and 3) most existing approaches consider only the effect of direct coupling between classes, while

* 基金项目: 国家自然科学基金 (62272412, 62232008, 62032010); 浙江省自然科学基金 (LY22F020007); 桐乡市通用人工智能研究院项目 (TAGI2-A-2024-0003); 浙江工商大学"数字+学科建设"项目 (SZJ2022B015)
收稿时间: 2024-12-14; 修改时间: 2025-02-17; 采用时间: 2025-04-23; jos 在线出版时间: 2025-09-17
CNKI 网络首发时间: 2025-09-18

ignoring the influence of indirect (non-contact) coupling and the diversity of degree distribution among neighboring nodes. To address these issues, a key class identification approach is proposed that integrates dynamic analysis with a gravitational formula. First, a class coupling network (CCN) is constructed using static analysis to represent classes and their coupling relationships. Second, a gravitational entropy (*GEN*) metric is introduced to quantify class importance by jointly considering direct and indirect couplings in the CCN and the degree-distribution diversity of neighboring nodes. Third, classes are ranked in descending order based on their *GEN* values to obtain a preliminary ranking. Finally, dynamic analysis is performed to capture actual runtime interactions between classes, which are used to refine the preliminary results. A threshold is applied to filter out non-key classes, producing a final set of candidate key classes. Experimental results on eight open-source Java projects demonstrate that the proposed method significantly outperforms eleven baseline approaches when considering no more than the top 15% (or top 25) of nodes. The integration of dynamic analysis notably improves the performance of the proposed method. Moreover, the choice of weighting schemes for coupling types has a minimal impact on performance, and the overall computational efficiency is acceptable.

Key words: key class identification; software network; gravitational formula; entropy; software measurement

软件维护是软件生命周期中不可或缺的环节, 也是确保软件持续高效运行的关键[1−3]. 在软件的维护过程中, 软件理解至关重要. 只有深入理解软件的架构、代码逻辑及历史变更等方面的内容, 才能有效地开展维护和改进工作[4]. 然而, 随着软件复杂性的激增, 一个成功的软件通常由数百、数千, 甚至数万个软件元素 (如方法、类、包、模块等) 通过复杂的交互构成, 这给软件的维护工作带来了巨大的挑战. 尤其当系统文档陈旧或不完整, 且开发人员对系统的了解有限时, 这种维护上的困难会更加突出[4,5].

理解软件要解决的首要问题是从软件的何处开始理解, 即软件理解的起点问题[6−9]. 关键类在面向对象软件 (如 Java 软件) 中扮演着重要角色[6,7]: 它们实现了软件的核心功能, 又与软件其他部分紧密耦合, 因而对软件的结构和功能具有重要影响. 以关键类为起点理解软件, 已成为理解复杂软件系统的有效途径之一[6−10]. 此外, 关键类还可用于软件文档化过程的优化[5]以及逆向工程类图的压缩[11]. 因此, 提供有效的技术来支持关键类的识别具有重要意义[12,13].

为了识别软件中的关键类, 研究者们提出了很多方法[5−10,13−28], 其中大部分是无监督学习方法[5−10,13−24]. 这些方法通常将软件的拓扑结构抽象为 (有向/无向或加权/无权) 软件网络——类抽象为网络中的节点, 类之间的耦合关系抽象为网络中的边, 并使用不同的网络指标 (如 *a*-index[14]、*h*-index[14]、coreness 及其变体[16,19]、PageRank 及其变体[5−10,13,15,17,20,23]等) 量化类的重要性, 进而将排名靠前 (按重要性降序排序) 的类作为候选的关键类. 现有工作取得了不错的效果: 在仅检查前 15% 的类时, 现有方法在大部分实验系统上的 *Recall* 值就已经达到了 50%–90%, 部分实验系统上甚至达到了 100%. 但是, 仍存在 3 点不足: 1) 现有方法使用的软件网络通常是通过对代码的静态分析构建的[5−10,13−28], 包含了软件元素之间所有可能的耦合关系. 然而, 随着软件的演化, 源代码中不可避免地会存在一些运行时永远不会 (或极少) 被执行到的“死代码”. 静态分析技术通常无法识别出这些“死代码”, 从而影响了基于静态分析技术的关键类识别方法的性能. 2) 目前也有部分工作通过动态分析构建软件网络. 但是动态分析比较依赖测试用例, 测试用例若构建得不够全面, 会导致所构建的软件网络存在遗漏真正关键类的风险. 3) 现有方法使用的度量指标通常只考虑了类之间的直接耦合对类重要性的影响, 忽视了类之间的间接 (非接触) 耦合的影响; 只考虑了邻居节点的度的影响, 忽视了邻居节点度分布多样性 (不同邻居节点的度不同) 带来的影响. 这些不足会导致所构建的度量指标无法充分、全面地刻画类的结构特征, 进而影响类重要性的准确量化.

为了解决上述问题, 本文提出了一种融合动态分析和引力公式的关键类识别方法 CDAG (identifying key classes using dynamic analysis and gravitational formula). 首先, CDAG 使用静态分析技术构建面向对象软件的类依赖网络 CCN (class coupling network), 以抽象类以及类之间的耦合关系. 其次, CDAG 综合考虑 CCN 中类之间“直接和间接的耦合”“邻居节点度分布的多样性”等对类重要性的影响, 构建了一个新的度量指标——引力熵 (gravitational entropy, *GEN*), 用以量化类的重要性. 然后, CDAG 按照类的 *GEN* 值对所有类进行降序排列, 从而得到初步的排序结果. 最后, CDAG 通过动态分析技术收集运行时类之间真实的交互关系, 进而对初步排序的结果进行优化, 并通过设定阈值来过滤非关键类, 从而得到候选的关键类. 本文以 8 个开源的 Java 软件作为实验对象, 并与文献中的 11 种方法进行对比. 研究结果表明, 本文提出的 CDAG 方法整体上均优于所有对比方法.

本文的主要贡献如下.

1) 提出了一个度量类重要性的新指标 *GEN*. 该指标基于引力公式和熵, 综合考虑了类之间的间接 (非接触) 耦

合及邻居节点度分布的多样性对类重要性的影响. 相比于其他指标, *GEN* 可以更加全面地刻画类的结构特征.

2) 提出了静态分析技术和动态分析技术相结合的关键类识别方法, 使用动态分析技术对静态分析的结果进行优化, 包括测试用例的自动生成、运行时类信息的收集以及类集的扩充等重要步骤.

3) 为了支持外部验证和后续研究, 我们公开了实验所用的全部数据及 CDAG 方法的源代码. 相关资料可从 https://github.com/wfpan/CDAG 下载.

本文第 1 节介绍所提出的基于动态分析和引力公式的关键类识别方法. 第 2 节通过充分的数据实验检验所提出方法的有效性. 第 3 节介绍关键类识别的相关方法和研究现状. 最后总结全文并展望下一步工作.

1 CDAG 方法

CDAG 方法主要包含 3 个部分 (如图 1 所示): ① CCN 的构建; ② *GEN* 指标的构建; ③ 类排序及关键类识别. 以下各节将以 Java 软件为例, 详细介绍各部分的实施过程.

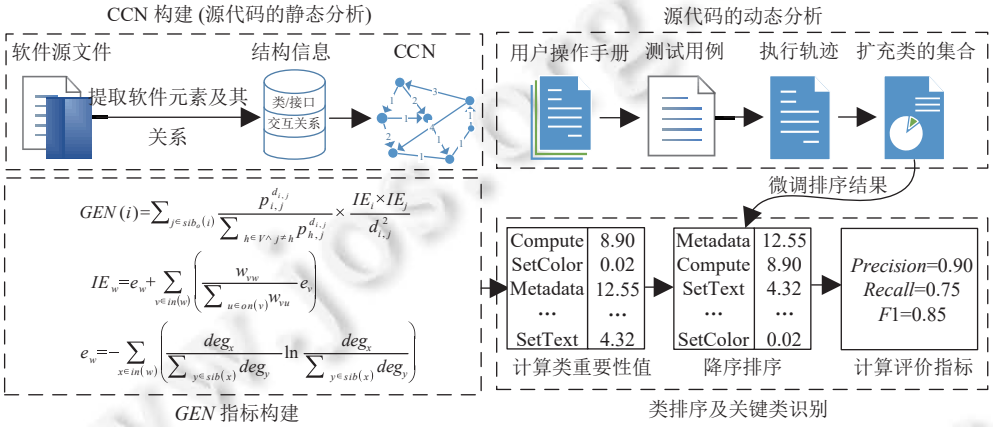

图 1 CDAG 方法的框架图

1.1 CCN 的构建

如前所述, 使用静态分析技术构建的软件网络可能会包含“死代码”, 从而导致所构建的软件网络“不准”; 使用动态分析技术构建的软件网络过分依赖于测试用例, 可能导致所构建的软件网络“不全” (遗漏真正的关键类). 为了兼顾软件网络的“准”和“全”, 我们使用静态分析技术和动态分析技术相结合的方式识别关键类, 即: 通过静态分析构建 CCN 和度量类的重要性, 并通过动态分析对静态分析的结果进行优化.

为了构建 CCN, CDAG 将使用静态分析技术对 Java 软件的源代码进行分析, 解析出源码中的各类软件元素 (类、接口、枚举、属性、方法、局部变量等) 以及元素之间的耦合关系 (类之间的继承关系、类对接口的实现关系、方法之间的调用关系等), 并将提取的信息抽象成 CCN 中的节点、有向边和边的权值. 我们使用自主研发的软件网络分析平台 SNAP (software network analysis platform)[5,9−11,24]提取上述软件的结构信息.

CCN 的定义如下[5,9−11,24]:

$$CCN=(V,L) \tag{1}$$

其中, V 是 CCN 的节点集, 用于抽象软件中的类; L 是 CCN 的有向边集, 用于抽象类之间的耦合关系. 在本文中, 类是 Java 中类、接口和枚举的统称. 此外, 我们忽略两节点之间在某个特定方向上的多条边, 即: 若节点 A、B 之间存在多条从节点 A 指向节点 B 的有向边, 则仅保留其中的一条.

在 CCN 的构建过程中, 我们将类之间共 9 种不同类型的耦合抽象为 CCN 中的有向边 $\langle A,B\rangle$.

1) 继承关系 INH (inheritance relation): 类 A 通过关键字“extends”继承类 B.

2) 实现关系 IMP (implements relation): 类 A 通过关键字“implements”实现接口 B.

3) 参数关系 PAR (parameter relation): 类 A 中的方法包含以类 B 为数据类型的参数.

4) 全局变量关系 GVR (global variable relation): 类 A 中存在以类 B 为数据类型的属性.

5) 局部变量关系 LVR (local variable relation): 类 A 中创建了以类 B 为数据类型的局部变量.

6) 返回类型关系 RET (return type relation): 类 A 中存在返回值类型是类 B 的方法.

7) 实例化关系 INS (instantiates relation): 类 A 中实例化了类 B 的对象.

8) 属性访问关系 ACC (field access relation): 类 A 中的方法访问了以类 B 为数据类型的属性.

9) 方法调用关系 MEC (method call relation): 类 A 中的方法调用了类 B 对象的方法.

CCN 是一个加权有向网络, 它与一个权重矩阵 ψ 关联. ψ 中任一元素 ψ_{ij} 的定义如下:

$$\psi_{ij}=\begin{cases} w_{ij}, & \langle n_i,n_j\rangle\in L \\ 0, & \text{其他} \end{cases} \tag{2}$$

其中, w_{ij} 是有向边 $\langle i,j\rangle$ 的边权, 用于抽象类节点 i 和类节点 j 之间的耦合强度. 边权越大, 表示相应类之间的耦合强度越大. 如果边 $\langle i,j\rangle$ 存在, 则 $\psi_{ij}=w_{ij}$; 否则, $\psi_{ij}=0$. 因此, 求 ψ_{ij} 的关键在于求 w_{ij}.

在面向对象 (如 Java) 软件中, 不同类型的耦合会有不同的权值 (交互强度), 并且这些耦合类型在类之间发生的频率也存在着显著差异. 因此, 类之间的耦合强度不仅受耦合类型的影响, 同时也受耦合类型发生频次的影响[5,9−11,19,24]. 此外, 在真实的软件系统中, 两个类之间可能会同时发生多种不同类型的耦合. 因此, 在计算两个类之间的耦合强度时, 我们将由不同类型耦合产生的强度进行累加, 即类节点 i 和类节点 j 之间的耦合强度 w_{ij} 的计算公式如下:

$$w_{ij}=\sum_r f_{ij}^r\times w_r \tag{3}$$

其中, $r\in\{\text{INH, IMP, PAR, GVR, LVR, RET, INS, ACC, MEC}\}$ 是上述 9 种耦合类型之一; f_{ij}^r 是类 i 与类 j 之间类型为 r 的耦合发生的频次, 其值可以通过分析所收集的软件结构信息确定; w_r 是耦合类型 r 的权值. 因此, 求 w_{ij} 的关键在于确定 w_r 的值. 在本文中, 我们使用 3 种不同的方式为 w_r 赋权.

(1) Brito E Abreu 等人[29,30]提出了一种基于耦合分布的赋权方法 DWM (distribution-based weighting mechanism). 该方法中, 不同类型耦合的权值会随耦合类型在软件不同包的包内和包外的分布变化而变化. 计算方法如下:

$$w_r=\begin{cases} 10, & N_{\text{intra}}^r\neq 0\wedge N_{\text{inter}}^r=0 \\ 1, & N_{\text{intra}}^r=0\wedge N_{\text{inter}}^r=0 \\ round\left(0.5+10\times\dfrac{N_{\text{intra}}^r}{N_{\text{intra}}^r+N_{\text{inter}}^r}\right), & \text{其他} \end{cases} \tag{4}$$

其中, N_{intra}^r 和 N_{inter}^r 分别表示类型为 r 的类间耦合其两端类出现在同一个包和出现在不同包的次数.

(2) Kang 等人[31]提出了一种基于顺序尺度的赋权方法 OWM (ordinal-scale-based weighting mechanism). 该方法考虑了类之间的 6 种耦合类型, 并为它们分配 1–10 之间的数值作为其权值. 本文将所识别出的 9 种耦合类型与 Kang 等人识别出的 6 种耦合类型进行映射 (对应关系如表 1 所示). Kang 等人未考虑 ACC 和 INS 两种耦合类型. 因此, 在使用该赋权方法时, 我们将忽略这两种耦合类型.

(3) Şora 等人[20]提出了一种基于实验的赋权方法 EWM (empirical weighting mechanism). 在该方法中, 不同类型耦合的权值是通过软件重构实验获得的, 即: 软件重构实验取得最优结果时, 各耦合类型所对应的权值. 赋权方法如表 2 所示.

表 1 OWM 赋权方法[31]

权值	Kang等人的耦合类型	本文的耦合类型
1	Dependency	MEC, PAR, LVR, RET
5.5	Aggregation/Composition association	GVR
7	Generalization (concrete parent)	INH (concrete parent)
9	Generalization (abstract parent)	INH (abstract parent)
10	Realize	IMP

表 2 EWM 赋权方法[20]

权值	耦合类型
4	IMP
3	INH, PAR, RET, GVR
2	MEC, ACC, INS
1	LVR

1.2 *GEN* 指标的构建

如前所述, 尽管目前已有很多指标用于度量类的重要性, 但是大部分均未考虑类之间的间接 (非接触) 耦合和邻居节点度分布的多样性对类重要性的影响, 导致无法准确、全面地刻画软件网络中类节点的结构特征以及类的重要性. 为了解决上述问题, 我们引入了引力公式和信息论中的熵.

万有引力可用来描述两个物体之间由于质量而产生的非接触相互作用[32,33]. 引力公式可用于计算两物体之间的引力. 此外, 信息论中的熵可用来度量系统的多样性和不确定性[34]. 熵值越大, 系统的多样性越高、不确定性越大. 因此, 我们借鉴万有引力和熵的概念, 构建度量指标刻画类之间的间接 (非接触) 耦合以及邻居节点度分布的多样性对类重要性的影响. 具体而言, 我们基于引力公式和熵构建了一个用于度量类重要性的指标引力熵 *GEN*.

节点 i 的引力熵 $GEN(i)$ 的定义如下:

$$GEN(i)=\sum_{j\in sib_o(i)} INF_{i,j} \tag{5}$$

其中, $GEN(i)$ 表示节点 i 的引力熵, $sib_o(i)$ 表示与节点 i 最短路径长度小于等于 o 的所有节点构成的集合, $INF_{i,j}$ 表示节点 j 对节点 i 的间接影响力. 从公式 (5) 可知, $GEN(i)$ 通过聚合集合 $sib_o(i)$ 内节点 j 对节点 i 的间接影响力来量化节点 i 的重要性. 理论上, o 的最小取值是 1, 最大取值是网络的直径 (网络中节点对之间最短路径的最大值). 在本文中, o 取值为 3, 主要出于两点原因: 1) 相关研究表明, 类主要受其 3-阶内的邻居的影响[35,36]; 2) 我们需要计算任意两节点间的最短路径, 过大的 o 值会显著降低指标的计算效率.

公式 (5) 中, 节点 j 对节点 i 的间接影响力 $INF_{i,j}$ 的定义如下:

$$INF_{i,j}=r_{i,j}^{d_{i,j}}\times\frac{IE_i\times IE_j}{d_{i,j}^2} \tag{6}$$

其中, $d_{i,j}$ 表示节点 i 和节点 j 之间的最短路径长度; $r_{i,j}^{d_{i,j}}$ 表示节点 j 的重要性传递到节点 i 的概率; IE_i 是从熵的角度计算得到的类 i 的重要性值.

$r_{i,j}^{d_{i,j}}$ 的定义如下:

$$r_{i,j}^{d_{i,j}}=\frac{p_{i,j}^{d_{i,j}}}{\sum_{h\in V\wedge j\neq h} p_{h,j}^{d_{i,j}}} \tag{7}$$

其中, $p_{i,j}^{d_{i,j}}$ 表示节点 i 和节点 j 之间的 $d_{i,j}$ 阶路径数量; V 是 CCN 的节点集; $\sum_{h\in V\wedge j\neq h} p_{h,j}^{d_{i,j}}$ 表示节点 j 和 V 中其他节点 h 之间的 $d_{i,j}$ 阶路径总数. 因此, $r_{i,j}^{d_{i,j}}$ 实际上量化的是节点 i 到节点 j 的 $d_{i,j}$ 阶路径数占节点 j 到其他所有节点的 $d_{i,j}$ 阶路径数的比例. $r_{i,j}^{d_{i,j}}$ 值越大, 表明相比于其他节点, 节点 j 更容易将其重要性传递给节点 i.

IE_i 的定义如下:

$$P_w=\frac{deg_w}{\sum_{x\in sib(w)} deg_x},\ x,w\in V \tag{8}$$

$$e_w=-\sum_{x\in in(w)} P_x\ln P_x,\ x,w\in V \tag{9}$$

$$IE_w=e_w+\sum_{v\in in(w)}\left(\frac{w_{vw}}{\sum_{u\in on(v)} w_{vu}}e_v\right),\ u,v,w\in V \tag{10}$$

其中, deg_w 表示类节点 w 的加权度 (CCN 中与节点 w 相连的边的权和); $sib(w)$ 表示节点 w 及其 1-阶邻居节点构成的集合; P_w 表示节点 w 的加权度占其 1-阶邻域内节点加权度总和的比例, 用以刻画节点 w 在其 1-阶邻域内的相对重要性; $in(w)$ 表示 CCN 中类 w 入边上的邻居节点的集合; e_w 表示 P_x 的熵; $on(v)$ 表示 CCN 中类 v 出边上的邻居节点的集合; w_{vw} 表示有向边 $\langle v,w\rangle$ 的边权值. 从公式 (10) 可知, 在求解 IE_w 的过程中, 我们不但考虑了类 w 自身的 e_w, 还考虑了类 w 入度上的邻居 v 传递过来的部分 e_v.

需要注意的是, 在上述 *GEN* 的定义中, 我们通过公式 (6) 考虑了类之间的间接耦合对类重要性的影响. 实际上, 公式 (6) 的构建借鉴了万有引力公式[32,33]:

$$F = G\frac{Mm}{r^2} \tag{11}$$

其中, F 是两个物体之间的引力; G 是引力常量; M 和 m 是两物体的质量; r 是两物体之间的距离. 不难发现, 为了构建公式 (6), 我们将万有引力公式中物体的质量替换成了节点的重要性, 距离替换成了 CCN 中两节点间的最短路径长度 $d_{i,j}$, G 替换成了公式 (7) 中定义的 $r_{i,j}^{d_{i,j}}$.

此外, 我们还通过公式 (9) 考虑了邻居节点度分布的多样性对类节点重要性的影响. 公式 (9) 求的是 P_x 的熵, 反映了类节点 w 入度上的邻居节点的度分布情况. 节点的熵越大, 通常意味着这个节点连接了系统中更多不同的信息流或数据流, 它在系统中发挥着更重要的作用.

综上可知, 我们提出的 *GEN* 指标综合考虑了类之间的间接耦合和邻居节点度分布的多样性对类重要性的影响, 从而可以更全面地刻画软件网络中类节点的结构特征及类的重要性.

1.3 类排序及关键类识别

在得到了一个软件的 CCN 之后, 我们可以使用 *GEN* 指标度量每个类节点的重要性, 进而可以依据类的 *GEN* 值对所有类进行降序排列, 并通过设定阈值过滤非关键类, 从而得到候选的关键类. 在本文中, 降序排序过程包含两个阶段: 1) 依据 *GEN* 值进行初步排序; 2) 使用动态分析对初步排序的结果进行优化. 下文将从初步排序、排序优化和阈值设定这 3 个方面进行介绍. 同时, 因为排序优化依赖于阈值设定, 所以我们将先介绍阈值设定.

(1) 阈值设定

在识别候选关键类的时候, 我们将设置一个阈值 k, 并将排名的前 $k\times|V|$ (k 为百分比, $|V|$为 CCN 中的节点数) 个或前 k 个类视为从该软件中识别出的候选关键类. 在现有的研究中, 阈值 k 一般有两种取值方式.

- k 取值 15%, 即将软件中前 15%×$|V|$的类视为候选关键类[5−9,14,19]. 为了更全面地评估 CDAG 方法的性能, 我们还选取了 1%–14% (步长为 1%) 共 14 个值作为新的阈值.
- k 取值 25, 即将软件中 top-25 个类视为候选关键类[22]. 为了更全面地评估 CDAG 方法的性能, 我们还选取了 1–24 (步长为 1) 共 24 个值作为新的阈值.

(2) 初步排序

我们以类的 *GEN* 值作为排序的依据, 并使用快速排序算法对所有类进行降序排列. 尽管节点的引力熵都是小数, 但是仍然无法避免多个节点具有相同的引力熵而无法准确排序的问题. 在本文中, 当这种情况发生时, 我们通过“求均值位置”的方式确定它们的最终排序位置, 即: 计算这些相同值中排在第 1 的类节点的位置 (设为 a) 以及排在最后的类节点的位置 (设为 b), 并将这些具有相同值的类节点的位置设置为 $(a+b)/2$.

(3) 排序结果优化

如前所述, 静态分析构建的 CCN 会存在“不准”的问题. “不准”的 CCN 会导致所识别的关键类也不准. 为了提高关键类识别的准确性, 我们使用动态分析技术对初步排序的结果进行优化. 优化过程主要包含 5 个步骤: ① 生成测试用例; ② 执行测试用例, 从而获得执行轨迹; ③ 分析执行轨迹, 从而获得执行过程中涉及的类的集合; ④ 对类的集合进行扩充; ⑤ 微调排序结果. 步骤①–③是动态分析的主要过程.

① 生成测试用例

在测试用例的构建过程中, 我们需要使生成的测试用例能够尽可能全面地覆盖系统的功能. 对于 GUI (图形用户界面) 系统, 用户操作手册通常能够较为完整地描述软件的功能和执行场景, 这为测试用例的自动生成提供了丰富的语义信息. 在本文中, 我们通过分析用户操作手册中的文本信息来自动提取与操作相关的动宾结构 (即代表操作的动词和代表操作对象的名词), 进而自动生成符合实际操作流程的测试用例.

我们使用 Stanford Parser[37]工具对用户操作手册中的文本进行分析. 首先, 我们对句子进行预处理, 将其拆分为多个子句, 并对每个子句逐词标注词性 (如动词、名词、形容词等). 其次, 根据依存关系分析句子的结构, 进而构建句子的依赖树, 并明确句子结构中的主语、宾语、动词等核心要素, 以及它们之间的相互关系. 其中, 动词代

表操作行为, 宾语代表操作的对象. 在本文中, 我们考虑了“nsubj” (主语)、“dobj” (直接宾语) 等关键依存关系, 从而确保操作动词能准确关联到目标对象. 与此同时, 我们还通过对“amod” (形容词修饰语) 等修饰语的提取, 以捕获更多操作对象的细节. 最后, 将提取到的操作动词、主语、宾语等信息作为一个测试用例保存到文件中. 文件中的每一行包括操作行为 (操作动词)、操作对象 (名词) 以及相关描述信息.

图 2 展示了软件 argoUML 用户操作手册中的一段文本. 我们将其作为 Stanford Parser 工具的输入, 从而可以得到图 3 所示的结果. 在图 3 中, 每一行的输出由 3 个部分 (图中以红蓝绿三色标识) 组成, 分别以“1;”“2;”“3;”开头. 其中, “1;”开头的部分表示动作的发出者. “2;”开头的部分表示动作及动作的对象, 它们之间以“|”分隔, 并标注了各自的词性 (“%”开头的部分). “3;”开头的部分用于记录操作对象的描述; 该部分通常以介词短语 (如“in|list”) 的形式描述动作发生的条件、位置或方式. 本文通过这种基于操作手册的测试用例生成方法, 自动生成测试步骤, 不仅可以减少人工编写测试用例的代价, 还可以在一定程度上保障所生成的测试用例集的完整性.

8.2.2.1. Selection
Here button 1 is used to choose (select) a model element (in a list or tree or on a diagram) on which subsequent operations will take place. Multiple model elements may be selected by using Shift and/or Ctrl in combination with button 1, see Section 8.2.5, “ Shift and Ctrl modifiers with Button 1 ”. Selection is always clearly indicated by a colored background. On a diagram, the selected model element is indicated with colored "blocks" at the corners/ends of the object. Model elements can be selected or deselected in different ways:
• Button 1 click. Deselects all model elements, and selects the one clicked on.
• Button 1 motion. Button motion (moving the mouse with the button down) in the diagram, not on any model element, allows to draw a rectangle around model elements which will be selected when the button 1 is released.
• Menu functions and shortcuts. Many menu operations change selection as side-effect, e.g. creating a new diagram. Many keyboard shortcuts for menu operations change the selection, e.g. Ctrl-A, which stands for the Select All function.

图 2 argoUML 操作手册中的部分文本

1;1:button model multiple element selection model select element model element2:choose%VB|element%NN;select%VBN|model multiple element%NN;select%VBN|model element%NN;3:in|list;in|combination;with|button;with|";by|background;On|diagram;with|block;at|corner;of|object;
2;1:2:3:
3;1:2:move%VBG|mouse%NN;draw%VB|rectangle%NN;release%VBN|button%NN;3:with|button;in|diagram;on|element;around|element;
4;1:selection selection 2:change%VBP|selection%NN;create%VBG|diagram%NN;change%VBP|selection%NN;3:for|operation;for|function;

图 3 使用 Stanford Parser 解析图 2 文本得到的结果

非 GUI 软件没有界面, 因而缺少关于软件如何操作的说明文档. 但是非 GUI 软件通常会提供测试业务功能的脚本, 并附带业务功能的说明和测试规范. 这些测试脚本是基于系统的核心业务逻辑和需求设计的, 能够覆盖系统的主要功能. 因此, 对于非 GUI 软件, 我们直接将软件源码中提供的业务功能测试脚本作为其测试用例, 并通过执行测试脚本获得执行轨迹.

② 执行测试用例, 从而获得执行轨迹

为了收集软件运行过程中软件元素的执行轨迹, 我们基于 Kieker[38]开发了一个脚本. 该脚本将嵌入到软件的源代码中, 实现对软件运行时行为的实时监控, 从而收集方法调用、消息传递和资源使用等信息. 这些轨迹信息详细反映了系统运行时软件元素之间真实的交互行为.

图 4 所示的是软件 argoUML 执行后收集到的部分执行轨迹 (trace) 信息. 图 4 中的每一行对应一条执行记录 (records), 代表一次方法调用. 每条记录均由 10 个字段组成, 字段之间以“;”分隔. 以图 4 中第 1 行记录 (红色框内的内容) 为例: “$1”表示该记录标识了一个方法调用事件. “1712916717267212400”是一个时间戳, 表示该调用事件发生的时间. “public boolean org.argouml.cognitive.AndCM.isRelevant(org.argouml.cognitive.Critic, org.argouml.cognitive.Designer)”是一个方法的完整签名, 用于描述该记录监控的方法. “<no-session-id>”表示会话 ID, 用于跟踪特定用户的会话. “3821726496288932241”为该记录所属 trace 的 ID (traceID). “1712916717267193700”和“1712916717267212300”分别表示方法调用的开始时间戳 (tin) 和结束时间戳 (tout). “DESKTOP-0JCC7QM”用于记录发生该调用事件的主机 (hostname). “973”用于记录该方法的执行顺序 (eoi). “2”表示该方法在堆栈中的深度 (ess). 各字段更详细的解释请参考 Kieker 的用户手册[39].

```
$1;1712916717267212400;public boolean org.argouml.cognitive.AndCM.isRelevant(org.argouml.cognitive.Critic,
org.argouml.cognitive.Designer);<no-session-id>;3821726496288932241;1712916717267193700;1712916717267212300;DESKTOP-
0JCC7QM;973;2
$1;1712916717267213100;public void org.argouml.cognitive.Critic.beActive();<no-session-
id>;3821726496288932241;1712916717267212800;1712916717267213000;DESKTOP-0JCC7QM;993;2
$1;1712916717267214300;protected java.util.List org.argouml.cognitive.CompositeCM.getMechList();<no-session-
id>;3821726496288932241;1712916717267214100;1712916717267214300;DESKTOP-0JCC7QM;995;3
$1;1712916717267218700;public java.lang.String org.argouml.cognitive.Critic.getCriticName();<no-session-
id>;3821726496288932241;1712916717267216700;1712916717267218600;DESKTOP-0JCC7QM;998;5
$1;1712916717267220300;public java.lang.String org.argouml.cognitive.Critic.getCriticName();<no-session-
id>;3821726496288932241;1712916717267219800;1712916717267220200;DESKTOP-0JCC7QM;999;5
$1;1712916717267221800;public java.lang.Object org.argouml.cognitive.Critic.getControlRec(java.lang.String);<no-session-
id>;3821726496288932241;1712916717267221000;1712916717267221700;DESKTOP-0JCC7QM;1000;5
$1;1712916717267222100;public boolean org.argouml.cognitive.Critic.isEnabled();<no-session-
id>;3821726496288932241;1712916717267216200;1712916717267222100;DESKTOP-0JCC7QM;997;4
$1;1712916717267222500;public boolean org.argouml.cognitive.EnabledCM.isRelevant(org.argouml.cognitive.Critic,
org.argouml.cognitive.Designer);<no-session-id>;3821726496288932241;1712916717267215600;1712916717267222400;DESKTOP-
0JCC7QM;996;3
```

图 4　argoUML 软件执行后得到的部分执行轨迹

③ 分析执行轨迹, 从而获得执行过程中涉及的类

在软件执行过程中, 一个方法调用通常会引发多个后续的方法调用. 例如, 当方法 *A* 调用方法 *B* 时, 方法 *B* 可能进一步调用方法 *C* 和方法 *D*. 因此, 一个方法调用往往对应着一组相关的执行记录, 这一组记录称为一个 trace. 在一个 trace 中, 所有的记录具有相同的 traceID. 此外, eoi 表示当前记录在整个 trace 中的执行顺序, 最先被执行的方法其 eoi 值为 0; ess 表示当前方法在调用栈中的深度, 最先被执行的方法其 ess 值为 0. 因此, 通过分析一个 trace 内各记录的 traceID、eoi 和 ess, 我们可以得到该 trace 内方法之间的真实调用关系. 算法 1 描述了从执行轨迹 (一个 trace) 提取方法之间调用关系的大致过程. 算法 1 的时间复杂度为 O(*n*) (*n* 为该 trace 内的执行记录数).

算法 1. 从执行轨迹提取方法之间的调用关系.

输入: 一个 trace (一组按照 eoi 值升序排列后的 record 记录);
输出: 方法之间的调用关系.

```
FOR EACH record IN trace DO
    IF 栈不为空 THEN
        IF record.ess≤栈顶 record.ess THEN
            WHILE 栈不为空且 record.ess≤栈顶 record.ess DO //回溯
                弹出栈顶 record
            END WHILE
            IF 栈为空 THEN
                跳出 FOR 循环
            END IF
            PRINT “栈顶 record.methodName###record.methodName” //###: 调用
            将当前 record 入栈
        ELSE // 如果当前 record 的 ess 大于栈顶元素的 ess
            PRINT “栈顶 record.methodName###record.methodName”
            将当前 record 入栈
        END IF
    ELSE // 栈为空的情况
        将当前 record 入栈 // 初始情况下, 将记录入栈
```

END IF
END FOR

我们以图 5(a) 中的 Java 代码片段为例, 展示从执行轨迹获取方法之间调用关系的过程. 假设该代码片段对应的 GUI 界面仅包含一个“搜索书本”按钮, 且测试用例仅涉及“点击搜索书本按钮”的操作. 为了模拟用户在 GUI 界面上的操作, 我们通过 main() 方法 (该方法未在代码片段中定义) 调用 Bookstore 类的 searchBook() 方法来替代 GUI 操作, 以获取对应的执行轨迹. 通过对调用过程的监控, 可以获得图 5(b) 所示的执行轨迹, 并通过算法 1 获得了方法之间的调用关系 (如图 5(c) 所示). 具体的分析过程如下: 首先, 将执行记录按照 traceID 分组, traceID 相同的归为一组, 表示它们属于同一个 trace, 并按照 eoi 值对属于同一个 trace 的记录进行升序排列. 以图 5(b) 为例, 由于图 5(b) 是一次方法调用 (main() 调用 searchBook()) 得到的执行轨迹, 故它们具有相同的 traceID (6675601285158273025), 属于同一个 trace. 经过升序排列后, 各记录的位置如①–⑧所示. 其次, 我们将该排序后的 trace (记录集) 输入到算法 1 中, 由算法 1 根据各记录的 ess 值自动确定方法之间的调用关系. 根据算法 1 可知, 1) 若当前记录的 ess 值大于前一条记录的 ess 值, 这意味着前一条记录对应的方法调用了当前记录的方法. 例如, 在图 5(b) 中, 记录⑥的 ess 值 (ess=2) 大于其前一条记录⑤的 ess 值 (ess=1), 这表示记录⑤的方法 searchBook() 调用了记录⑥的方法 getBook(Boolean). 因此, 可以得到图 5(c) 中的调用关系①. 2) 若当前记录的 ess 值小于或等于前一条记录的 ess 值, 这说明前一条记录所代表的方法已经执行完毕. 此时, 算法 1 将执行一系列的出栈操作, 直到栈顶记录的 ess 值小于当前记录的 ess 值, 这表示栈顶记录中的方法调用了当前记录中的方法. 例如, 在图 5(b) 中, 记录⑦的 ess 值 (ess=2) 等于其前一条记录⑥的 ess 值, 这表示记录⑥对应的方法 getBook(Boolean) 已执行完毕, 此时我们需要回溯到调用 getBook(Boolean) 的 searchBook() 方法处, 并得到 searchBook() 和 getOffers() 之间的调用关系 (图 5(c) 中的调用关系②). 同理, 我们可以得到示例代码中的其他调用关系 (如图 5(c)).

```
public class BookStore {
    private final Catalog catalog/* GVR */ = new Catalog()/* INS */;
    private final CRM crm/* GVR */ = new CRM(catalog)/* INS */;
    public void searchBook() {
        catalog/* ACC*/.getBook(false);/* MEC */
        crm/* ACC */.getOffers();/* MEC */
    }
}

public interface CatalogInterface {
    void getBook(final boolean complexQuery);
}

public class Catalog implements CatalogInterface /* IMP */{
    public void getBook(final boolean complexQuery) {
        // nothing to do here
    }
}

public class CRM {
    private final Catalog catalog;/* GVR */
    public CRM(Catalog catalog/* PAR */) {
        this.catalog/* ACC */ = catalog;
    }
    private void getOffers() {
        catalog/* ACC */.getBook(false);/* MEC */
    }
}
```

(a) 示例代码

```
$1;1731815295558386900;public org.example.Catalog.<init>();<no-session-
id>;6675601285158273025;1731815295558183600;1731815295558203300;DESKTOP-0JCC7QM;2;2 ③
$1;1731815295562346300;public org.example.CRM.<init>(org.example.Catalog);<no-session-
id>;6675601285158273025;1731815295562335100;1731815295562342600;DESKTOP-0JCC7QM;3;2 ④
$1;1731815295562375900;public org.example.BookStore.<init>();<no-session-
id>;6675601285158273025;1731815295554350300;1731815295562373300;DESKTOP-0JCC7QM;1;1 ②
$1;1731815295563238100;public void org.example.Catalog.getBook(boolean);<no-session-
id>;6675601285158273025;1731815295563217900;1731815295563233800;DESKTOP-0JCC7QM;5;2 ⑥
$1;1731815295563287100;public void org.example.Catalog.getBook(boolean);<no-session-
id>;6675601285158273025;1731815295563281300;1731815295563284900;DESKTOP-0JCC7QM;7;3 ⑧
$1;1731815295563295200;public void org.example.CRM.getOffers();<no-session-
id>;6675601285158273025;1731815295563270800;1731815295563293600;DESKTOP-0JCC7QM;6;2 ⑦
$1;1731815295563302000;public void org.example.BookStore.searchBook();<no-session-
id>;6675601285158273025;1731815295562400500;1731815295563300400;DESKTOP-0JCC7QM;4;1 ⑤
$1;1731815295563311700;public static void org.example.Main.main(java.lang.String[]);<no-session-
id>;6675601285158273025;1731815295550527700;1731815295563310200;DESKTOP-0JCC7QM;0;0 ①
```

(b) 获得的trace

```
public static void org.example.Main.main(java.lang.String[])###public org.example.BookStore.<init>()
public org.example.BookStore.<init>()###public org.example.Catalog.<init>()
public org.example.BookStore.<init>()###public org.example.CRM.<init>(org.example.Catalog)
public static void org.example.Main.main(java.lang.String[])###public void
org.example.BookStore.searchBook()
public void org.example.BookStore.searchBook()###public void org.example.Catalog.getBook(boolean) ①
public void org.example.BookStore.searchBook()###public void org.example.CRM.getOffers() ②
public void org.example.CRM.getOffers()###public void org.example.Catalog.getBook(boolean)
```

(c) 获得的方法调用关系

图 5 方法调用关系提取示例

在获得了方法之间的调用关系后, 我们可以从方法的完整签名中获得方法所在的类, 进而将方法之间的调用关系映射为方法所在类之间的调用关系. 例如, 在图 5(c) 中, ②所标识的调用关系 (searchBook() 和 getOffers() 之间的调用关系) 可以映射为类 BookStore 和 CRM 之间的关系. 最终, 我们可以获得软件执行过程中涉及的类的集合, 以及类之间的调用关系.

④ 对类的集合进行扩充

除了类之外, 软件中的接口 (或纯抽象类) 和枚举也发挥着非常重要的作用. 然而, 接口和枚举通常不能直接

通过动态分析的方式进行跟踪和捕获, 原因主要包括两个方面: 首先, 动态分析主要依赖运行时的方法调用信息, 而大部分的接口定义往往仅包含了方法的声明 (抽象方法), 并没有具体的实现逻辑, 因此不会在运行时直接参与方法的调用; 其次, 枚举在运行时往往是作为常量来使用, 如果枚举没有定义任何方法, 那么它就不会直接参与到方法的调用中. 当然对于包含了默认或静态方法的接口, 以及包含了方法的枚举, 动态分析仍然是可以捕获的. 在本文中, 我们使用动态分析获得的类集对排序的结果进行优化. 如果动态分析过程中缺失了一些比较重要的类 (抽象类)、接口、枚举, 这可能会对排序结果产生一定的影响. 因此, 为了尽量消除这种影响, 我们将对直接动态分析获得的类集进行扩充, 即将那些无法直接捕获的抽象类、接口和枚举加进来.

类集的扩充依赖于前面静态分析收集的软件结构信息以及构建的 CCN. 具体如下.

● 接口 (或抽象) 类的扩充: 静态分析通过识别代码中的“implements”和“extends”关键字, 可以准确识别类与抽象类、接口与接口之间的继承关系, 以及类与接口之间的实现关系, 并得到完整的继承和实现链. 我们基于静态分析得到 CCN, 并以动态分析识别出的类为起点, 沿着静态关系 (implements 和 extends) 结构向上追溯, 找到其直接或间接实现的接口及其继承链上的抽象类, 将它们扩充到类集中. 以图 5(a) 代码中的类 Catalog 为例, 从 CCN 中我们能识别出类 Catalog 实现了接口 CatalogInterface 这一关系, 即使在动态分析中 CatalogInterface 接口并未直接出现在方法调用链上, 我们仍然可以将接口 CatalogInterface 与类 Catalog 进行关联, 并扩充到类集中.

● 枚举的扩充: 我们基于静态分析收集的软件结构信息及构建的 CCN, 从中识别出动态分析中捕获的所有类, 并进一步分析这些类在其实现中是否直接或间接使用了枚举. 由于枚举类通常以常量的形式被使用, 因此我们只需要通过查找类的 ACC 关系 (属性访问关系) 来寻找潜在的枚举使用情况. 例如, 当某个方法直接调用了某枚举的常量, 或者通过调用链间接依赖了枚举类型的字段或方法, 我们便可以将该枚举扩充到类集中.

⑤ 微调排序结果

我们使用扩充后的类集对排序结果进行微调, 即对于在初步排序中排在前 $k\times|V|$ (k 为百分比) 或前 k (k 为整数) 的类, 从头开始依次检查每个类, 若该类不存在于扩充后的类集中, 则将其与前 $k\times|V|+1$ 或前 $k+1$ 位置的类进行交换, 以期将重要的类往前移动, 从而提高真正的关键类出现在前 $k\times|V|$或前 k 个类中的概率.

2 实验设计

为了检验 CDAG 方法在软件关键类识别上的有效性, 我们设计了一系列的实验. 在本文中, CDAG 方法是使用 JDK 17.0.9 和 Eclipse 4.30.0 开发的, 所有的实验是在一台装有 Windows 11 64 位操作系统、2.60 GHz 的 13th Gen Intel(R) Core(TM) i9-13900H CPU 和 32 GB 内存的 ThinkPad 笔记本上实施的.

第 2.1 节描述实验对象. 第 2.2 节介绍对比方法. 第 2.3 节描述用于评估方法效果的度量指标. 第 2.4 节描述实验的结果及对结果的分析. 第 2.5 节对结果的有效性进行分析.

2.1 实验对象

文献 [10] 收集了现有关键类识别研究中使用的所有软件, 构建了一个包含 18 个 Java 软件及其关键类的数据集. 据我们所知, 这是迄今为止最大规模的关键类识别数据集. 本文从该数据集中筛选出了 8 个软件作为本文的实验对象. 我们未选用其余的 10 个软件, 主要出于以下 4 点原因: 1) 用户操作手册的质量比较低 (对软件操作过程没有详细说明), 甚至缺失; 2) 缺失 (或无法找到) 软件运行必须的文件或依赖库; 3) JDK 版本过早, 无法兼容当前运行环境; 4) 非 GUI 系统缺少内置的测试用例 (或内置的测试用例极少).

本文使用的软件如表 3 所示. 其中, argoUML 是一款开源的 UML 建模工具, 用于创建和编辑 UML 图形; jEdit 是一款开源的程序员文本编辑器, 支持多种编程语言的语法高亮和插件扩展; jHotDraw 是一款 Java 框架, 用于创建 2D 图形应用程序, 并提供了图形编辑功能; jMeter 是一款开源的 Java 应用程序, 用于负载测试和性能测量; Mars 是一个开源 Java 项目, 致力于构建计算机模型, 以模拟和探索在火星上建立人类定居点的各种关键要素; Maze 是一款用 Java 实现的迷宫游戏; PDFBox 是一个 Java 库, 用于创建、操作和提取 PDF 文档内容; wro4j 是一个 Web 资源优化框架, 用于压缩和管理 JavaScript 和 CSS 资源.

表 3 还给出了这 8 个软件的基本信息, 包括项目名称、版本号、代码行的数量、包的数量、类的数量、方法/

属性的数量、关键类的数量. 尽管我们仅使用了 8 款软件, 但是这并不会对我们的结论产生大的影响, 因为这些系统具有很好的多样性 (来自不同领域、具有不同的规模).

表 3 实验系统的基本情况

项目	版本号	代码行数	包数	类数	方法/属性数	关键类数
argoUML	0.9.5	74 334	67	846	6 178/2 851	12
jEdit	5.1.0	112 492	41	1 082 (9)	7 601/4 085	7
jHotDraw	6.0b.1	28 330	30	544	5 205/865	9
jMeter	2.0.1	22 701	42	260	2 000/834	14
Mars	3.06	132 589	95	1 085 (32)	11 105/5 738	29
Maze	1	8 881	6	63 (6)	563/284	27
PDFBox	2.0.7	135 514	109	1 285 (33)	10 482/4 792	12
wro4j	1.6.3	33 736	99	567 (9)	3 256/1 274	12

2.2 对比方法

为了检验 CDAG 方法的有效性, 我们从现有研究中选择了 11 种方法作为对比方法: *h*-index[14]、*a*-index[14]、PageRank[17,20]、PageRankBR[17,20]、*k*-core[16]、ICOOK[19]、PageRankIVOL[23]、ElementRank[9]、Pride[5]、MinClass[22]和 iFit[24]. 我们实现了这些对比方法, 并在应用过程中对软件网络进行了必要的调整 (如忽略部分耦合类型、忽略边的方向/权重等), 以便生成与这些对比方法匹配的软件网络, 从而得到相应的结果. 部分现有工作未被选为对比方法, 主要是因为我们缺乏足够的信息来复现这些工作. 同时, 有监督的关键类识别方法均未被选作对比方法, 主要原因是: 有监督方法需要划分训练集和测试集, 结果是在测试集上得到的, 而无监督方法的结果是在整个数据集上得到的. 将无监督方法与有监督方法进行对比意义不大[5]. 同时, 若这些对比方法也产生了相同的类重要性值, 导致无法对类进行准确排序时, 我们也采用了第 1.3 节中的“求均值位置”的方法.

这 11 种对比方法简述如下.

- *h*-index: 该方法将类节点在软件网络中的 *h*-指数作为其重要性值.
- *a*-index: 该方法将类节点在软件网络中的 *a*-指数作为其重要性值.
- PageRank: 该方法使用 PageRank 算法计算类节点在加权有向软件网络中的重要性.
- PageRankBR: 该方法与 PageRank 方法类似, 主要区别在于: 若原始软件网络中存在类 *A* 到类 *B* 的有向边, 那么该方法将增加类 *B* 到类 *A* 的有向边, 并且边权是原来的一半.
- *k*-core: 该方法将节点在软件网络中的核数 (coreness) 作为其重要性值. 类节点的核数是通过 *k*-核分解 (*k*-core decomposition) 算法计算得到的.
- ICOOK: 该方法将类节点在软件网络中的一般化核数 (generalized coreness) 作为其重要性值. 节点的一般化核数是通过一般化 *k*-core 分解 (generalized *k*-core decomposition) 算法计算得到的.
- PageRankIVOL: 该方法与 PageRank 方法类似, 主要区别在于: 在跟随出链将类 *A* 的重要性传递给类 *B* 的过程中, PageRankIVOL 进一步考虑了 *A* 与 *B* 之间有向边的边权; 在随机跳转部分, PageRankIVOL 进一步考虑了网络的总节点数.
- ElementRank: 该方法构建软件类级的多层软件网络, 并使用 PageRank 算法计算每一层网络中类节点的重要性, 进而通过聚合类各层的重要性值来得到类最终的重要性值.
- Pride: 该方法与 PageRankIVOL 方法类似, 主要区别在于: 在随机跳转部分, Pride 进一步考虑了网络中节点的度、加权度及网络的总度及总加权度.
- MinClass: 该方法基于熵的概念构建了一个新指标 *OSE*, 并使用 *OSE* 衡量每个类在系统中的重要性.
- iFit: 该方法基于 PageRank 算法和物理中场的概念构建了一个新指标 *CG*, 并使用 *CG* 度量类的重要性.

2.3 评价指标

关键类识别领域广泛使用的评价指标包括: 准确率 (*Precision*)、召回率 (*Recall*) 和 *F*1. 各指标的具体定义如下:

$$Recall = \frac{TP}{TP + FN} \tag{12}$$

$$Precision = \frac{TP}{TP+FP} \tag{13}$$

$$F1 = \frac{2\times Precision\times Recall}{Precision+Recall} \tag{14}$$

其中, *TP* (true positive) 指识别为关键类的实际也是关键类的数量; *FP* (false positive) 指识别为关键类的实际是非关键类的数量; *FN* (false negative) 指识别为非关键类的实际是关键类的数量; *TN* (true negative) 指识别为非关键类的实际也是非关键类的数量. 此外, 对于关键类识别问题, *Recall*、*Precision* 和 *F*1 的变化趋势是一致的, 即 *Recall* 大的方法其 *Precision* 和 *F*1 也大[5]. 因此, 我们无需同时报道 *Recall*、*Precision* 和 *F*1 这 3 个结果. 在公式 (12) 中, 分子表示识别为关键类的实际也是关键类的数量, 分母实际上表示的是一个软件中关键类的数量. 因此, *Recall* 实际上求的是特定方法正确识别的关键类的比例. 方法越好, 其 *Recall* 值越大. 鉴于 *Recall* 已经可以直接反映某一方法在关键类识别方面的性能, 在数据实验部分, 本文仅报道了 *Recall* 的结果, 其他相关数据及结果可以从 https://github.com/wfpan/CDAG 处下载.

2.4 实验结果及分析

本文将重点围绕以下 5 个研究问题展开实验.

- RQ1: 在检查不超过前 15% 的节点时, CDAG 是否比现有方法更有效?
- RQ2: 在检查不超过 top-25 的节点时, CDAG 是否比现有方法更有效?
- RQ3: 动态分析对关键类识别是否均有效?
- RQ4: 不同的赋权方式对 CDAG 的性能是否有影响?
- RQ5: 在运行效率方面, CDAG 是否可行?

我们按照图 1 所示的步骤构建 CCN、计算类的重要性、对类进行排序和对结果进行优化, 最终得到候选关键类. 在本节中, 我们将围绕所提出的几个研究问题开展数据实验, 详细检验方法的有效性.

RQ1: 在检查不超过前 15% 的节点时, CDAG 是否比现有方法更有效?

目前已有不少关键类识别方法, 本节将检验我们提出的 CDAG 方法在关键类识别上是否比现有的方法效果更好. 首先关注阈值 *k* 取值 1%–15% (步长为 1%) 的情况, 即将软件中前 $k\times|V|$ ($|V|$为 CCN 中的节点数) 的类视为候选关键类. 图 6 所示的是我们的方法 CDAG 及 11 种对比方法在 8 个软件、15 个阈值上取得的结果 (*Recall* 值). 在构建 CCN 时, 我们采用 DWM 为不同类型的耦合赋权.

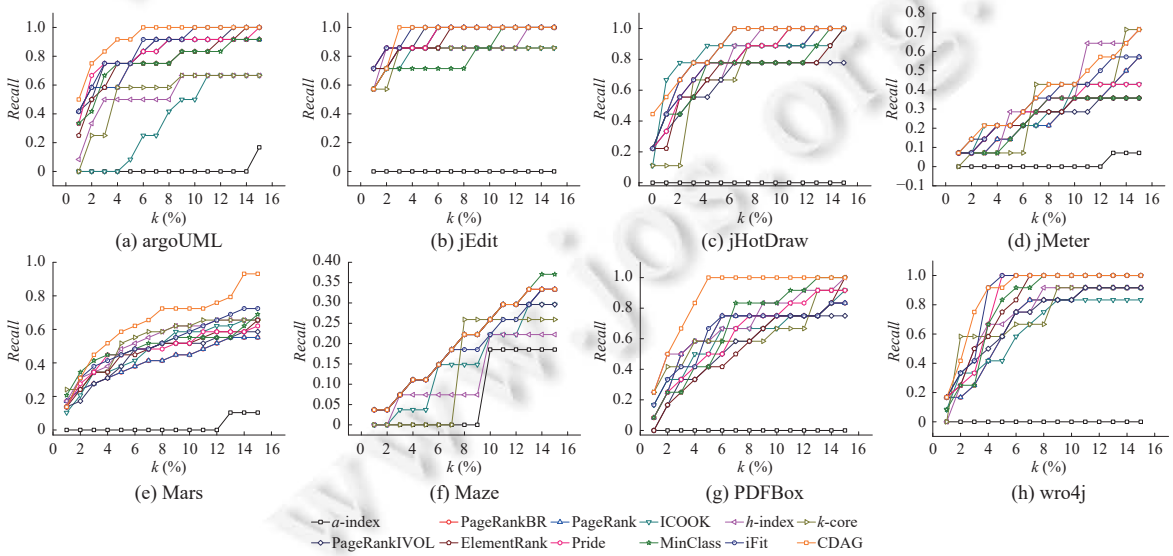

图 6 *Recall*@top-*k* (使用 DWM 为不同类型的耦合赋权, *k* 为百分比形式)

从图 6 可知, 当采用 DWM 为 CCN 中不同类型的耦合赋权时: 1) 没有一种方法在这 8 个系统的 15 个阈值处均取得最好的结果; 2) 当阈值等于 15% 时, 除了 Maze 系统之外, CDAG 在其他 7 个系统上均优于 (或不差于) 11 种对比方法; 3) 当阈值从 1% 以步长 1% 增加到 14% 的过程中, CDAG 在大部分情况下优于 (或不差于) 其他 11 种对比方法; 4) 在部分阈值处, CDAG 比 iFit (如 jEdit 的 1% 和 2% 处)、ICOOK (如 jHotDraw 的 2% 和 5% 处)、*h*-index (如 jEdit 的 2% 处)、Pride (如 jEdit 的 2% 处) 略差. 当采用 OWM 或 EWM 为 CCN 中不同类型的耦合赋权时, 我们也得到了类似的结果 (结果如图 7 和图 8 所示).

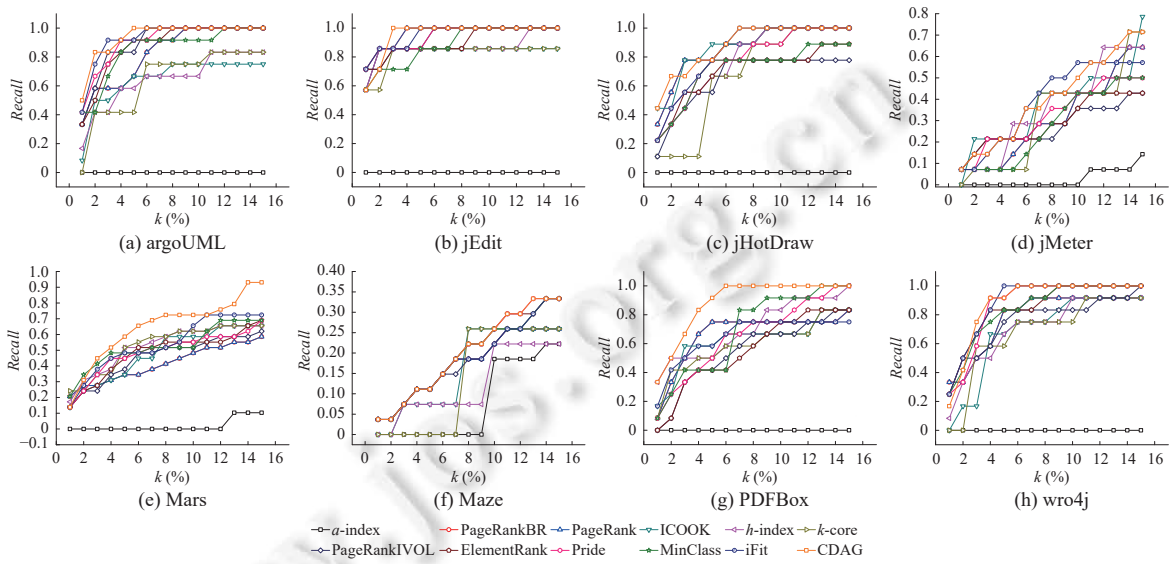

图 7 *Recall*@top-*k* (使用 OWM 为不同类型的耦合赋权, *k* 为百分比形式)

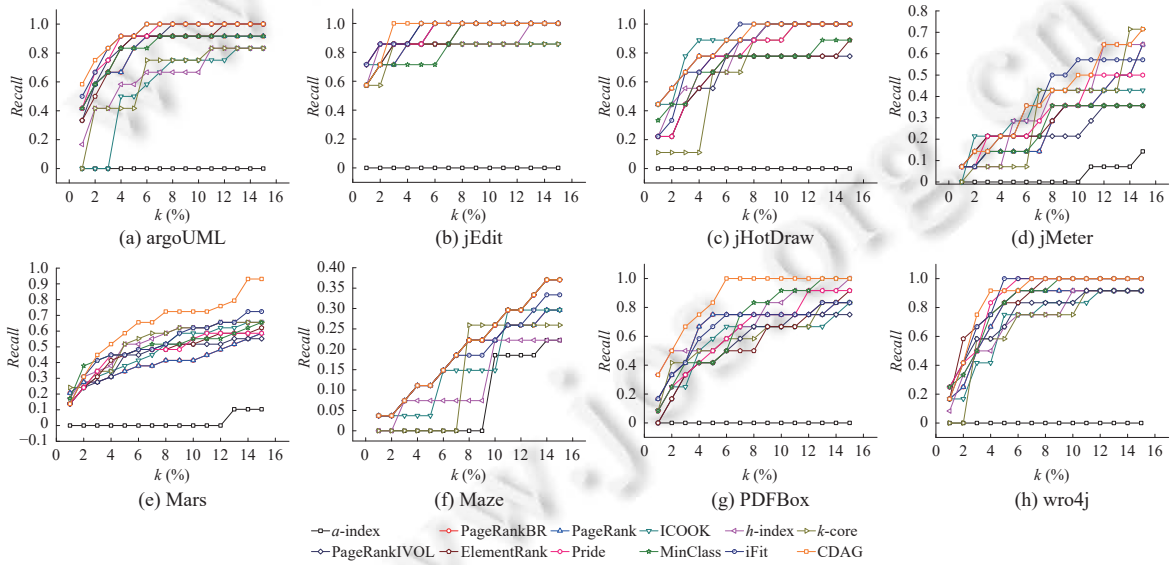

图 8 *Recall*@top-*k* (使用 EWM 为不同类型的耦合赋权, *k* 为百分比形式)

如前所述, 没有一种方法在所有 8 个系统的 15 个阈值上都表现的最好. 这促使我们进一步分析不同方法在整个测试集 (8 个系统) 上的整体性能. 为此, 我们引入了 Friedman 检验 (Friedman test)[40]. Friedman 检验是一种无参数统计检验方法, 可以根据不同方法在不同软件某一相同阈值处的 *Recall* 值, 计算特定阈值处不同方法在整个测

试集上的平均排名. 在本文中, 12 种方法的平均排名如图 9 所示, 其相应的 P 值 (显著性水平) 均远小于 0.05, 这表明我们的结果拒绝原假设 (不同方法之间没有差异), 即这些方法之间存在显著差异. 由于一个方法具有较大的 *Recall* 值, 表示该方法性能越好. 在根据 *Recall* 值对不同方法进行排名时, Friedman 检验将为较好的方法分配较小的排名值 (*Rank* 值). 从图 9 可知, CDAG 在 3 种不同的耦合类型赋权方式下, 均排在第 1, 即 CDAG 取得了最好的整体性能.

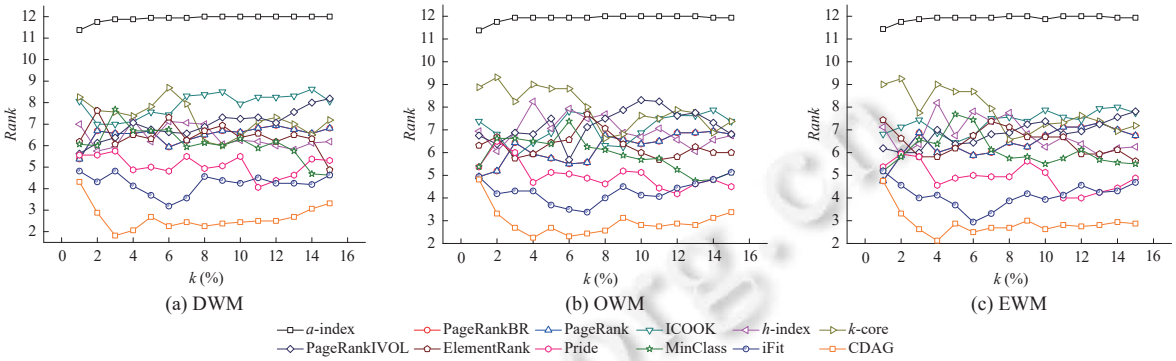

图 9 12 种方法在不同赋权方式下的平均排名 (*Rank*@top-*k*, *k* 为百分比形式)

根据 Friedman 检验的结果, 我们可以得出结论: 在检查不超过前 15% 的节点时, CDAG 从整体而言显著优于其他方法.

RQ2: 在检查不超过 top-25 的节点时, CDAG 是否比现有方法更有效?

尽管现有工作中普遍采用 15% 作为阈值, 但是对于一些规模比较大的系统, 返回前 15% 的类作为候选关键类, 仍然可能包含过多的非关键类. 例如, PDFBox 包含 1318 个类 (如表 3 所示), 返回前 15% 的类将得到将近 198 个类作为候选关键类, 然而该系统实际包含 12 个真正的关键类. 因此, 198 个候选关键类中 93.9% 都是非关键类. 在本节中, 我们关注阈值 k 取值 1–25 (步长为 1) 时的情况 (即将软件中 top-k 的类视为候选关键类), 以检验各个方法在仅检查极少的类时的性能. 图 10–图 12 所示的是我们的方法 CDAG 及 11 种对比方法在 8 个软件、25 个阈值上取得的结果 (*Recall* 值).

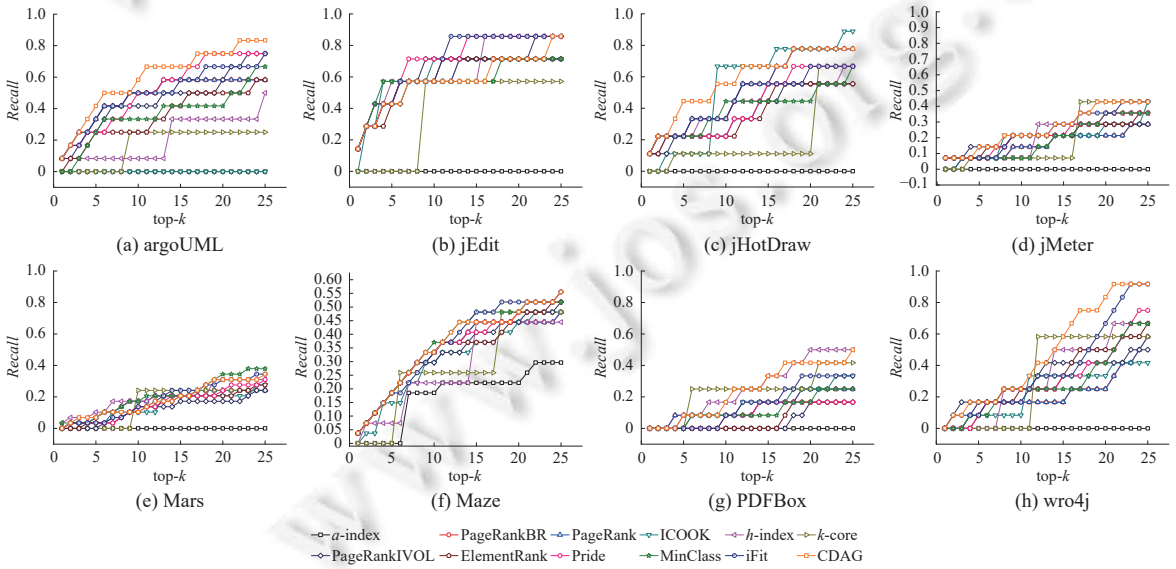

图 10 *Recall*@top-*k* (使用 DWM 为不同类型的耦合赋权)

从图 10 可知, 当采用 DWM 为 CCN 中不同类型的耦合赋权时: 1) 没有一种方法在这 8 个系统的 25 个阈值处都取得了最好的结果; 2) 在 argoUML 系统中, CDAG 均优于 (或不差于) 其他 11 种对比方法; 3) 除 argoUML 系统外, CDAG 在部分阈值处比 iFit、Pride、MinClass、ICOOK、*k*-core、*h*-index 等方法差. 当采用 OWM 或 EWM 为 CCN 中不同的耦合类型赋权时, 我们也得到了类似的结果 (结果如图 11 和图 12 所示).

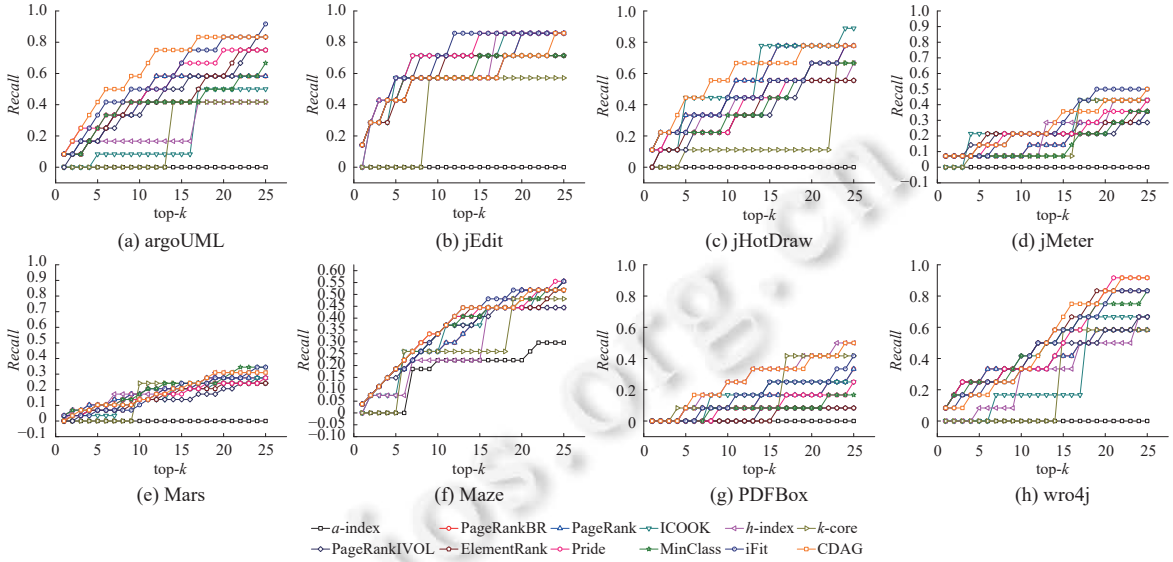

图 11 *Recall*@top-*k* (使用 OWM 为不同类型的耦合赋权)

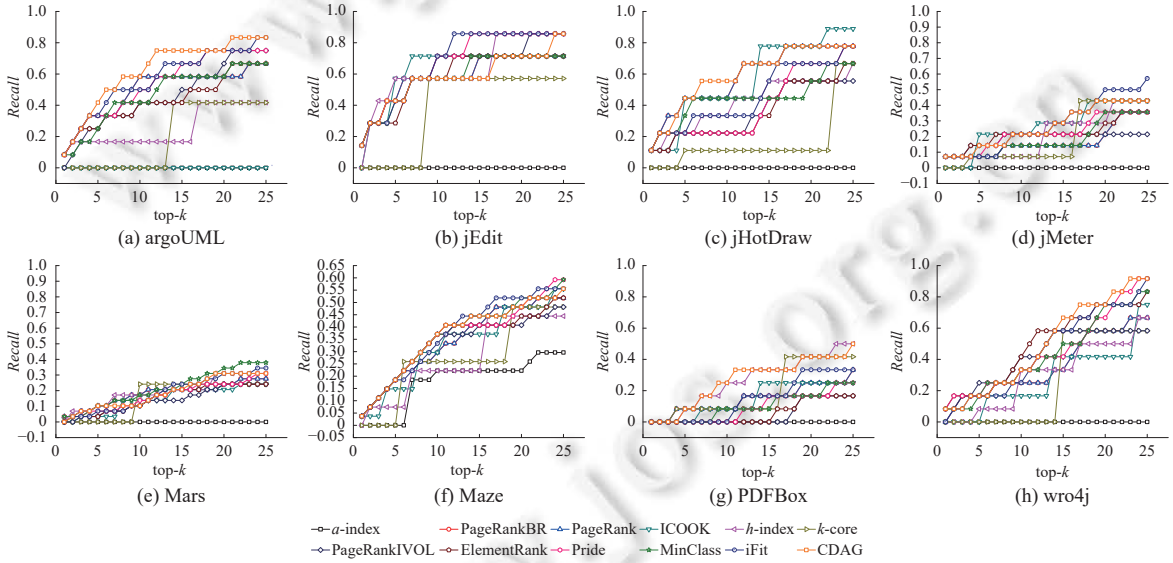

图 12 *Recall*@top-*k* (使用 EWM 为不同类型的耦合赋权)

与 RQ1 中类似, 我们也使用 Friedman 检验进一步分析不同方法在整个测试集 (8 个系统) 上的整体性能. 12 种方法的平均排名如图 13 所示, 其相应的 P 值均远小于 0.05. 这表明我们的结果拒绝原假设 (不同方法之间没有差异), 即这些方法之间存在显著差异. 从图 13 可知: 1) 当使用 DWM 为不同耦合类型赋权时, CDAG 明显优于其他 11 种方法, 仅在 top-14、top-15 处略差于 iFit 方法; 2) 在 OWM 和 EWM 两种不同的耦合类型赋权方式下, CDAG 与 iFit 不相上下, 但是明显优于其他 10 种对比方法.

为了更清晰地对比 CDAG 与 iFit 的性能, 我们进一步计算了 CDAG 相比 iFit 的 Win/Tie/Loss 值 (基于 Friedman 检验的结果计算), 结果如表 4 所示. 在表 4 中, Win 值表示 CDAG 优于 iFit 的次数, Tie 值表示两个方法 *Recall* 值一样的次数, Loss 值表示 CDAG 差于 iFit 的次数, DWM/OWM/EWM 代表 CCN 使用不同的方式为不同类型的耦合赋权. 从表 4 可知, 在 DWM 赋权方式下, CDAG 在 20 个阈值处优于 iFit 方法, 在 3 个阈值处与 iFit 性能一样, 在 2 个阈值处劣于 iFit 方法; 在 OWM 赋权方式下, CDAG 在 14 个阈值处优于 iFit 方法, 在 3 个阈值处与 iFit 性能一样, 在 8 个阈值处劣于 iFit 方法; 在 EWM 赋权方式下, CDAG 在 13 个阈值处优于 iFit 方法, 在 1 个阈值处与 iFit 性能一样, 在 11 个阈值处劣于 iFit 方法. 因此, 从整体而言, CDAG 在 47 个阈值处优于 iFit 方法, 在 7 个阈值处与 iFit 性能一样, 在 21 个阈值处劣于 iFit 方法; 换言之, CDAG 在 54/75 个阈值处不差于 iFit.

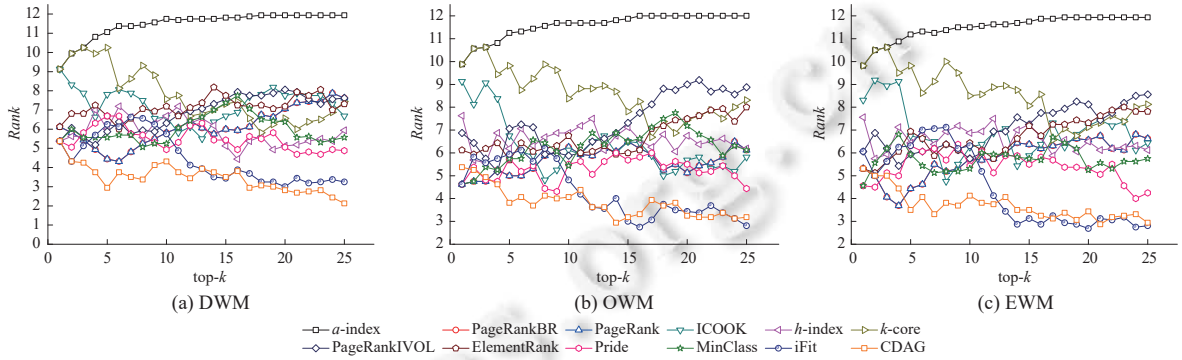

图 13 12 种方法在不同赋权方式下的平均排名 (*Rank*@top-*k*)

表 4 CDAG 相比 iFit 的 Win/Tie/Loss 值

指标	DWM	OWM	EWM	合计
Win	20	14	13	47
Tie	3	3	1	7
Loss	2	8	11	21

根据 Friedman 检验的结果和 Win/Tie/Loss 值, 我们可以得出结论: 在检查不超过 top-25 的节点时, CDAG 从整体而言显著优于其他方法.

RQ3: 动态分析对于关键类识别是否均有效?

如第 1.3 节所述, 本文使用动态分析对基于 CCN 和 *GEN* 识别的候选关键类 (初步排序结果) 进行优化. 在此部分, 我们将检验该优化操作是否有效. 为此, 我们将 CDAG 中的“排序结果优化”步骤去掉, 构建了一种新的关键类识别方法 CG (identifying key classes using gravitational formula), 并通过对比 CDAG 和 CG 在同一个系统、同一个阈值处的 *Recall* 值来检验“优化”操作的效果. 与上文类似, 我们分别检验了阈值不超过 15% (前 15%) 和阈值不超过 25 (top-25) 两种情况下的结果 (结果如图 14 和图 15 所示).

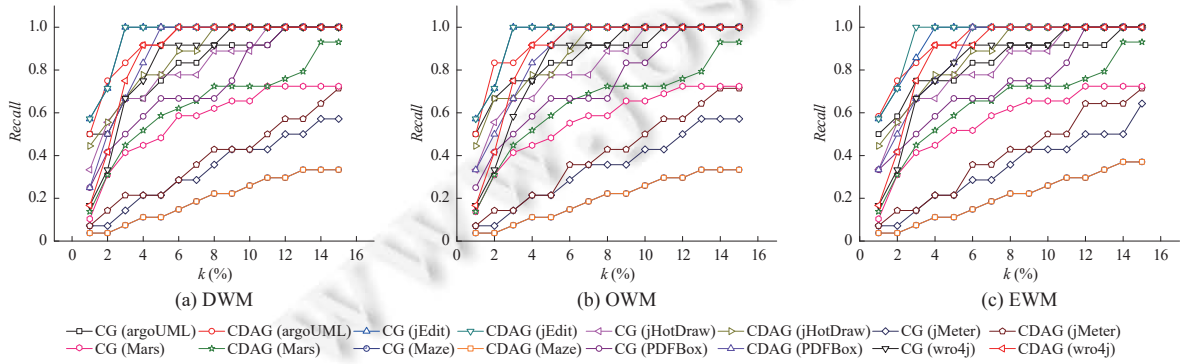

图 14 CDAG vs. CG (*Recall*@top-*k*, *k* 为百分比形式)

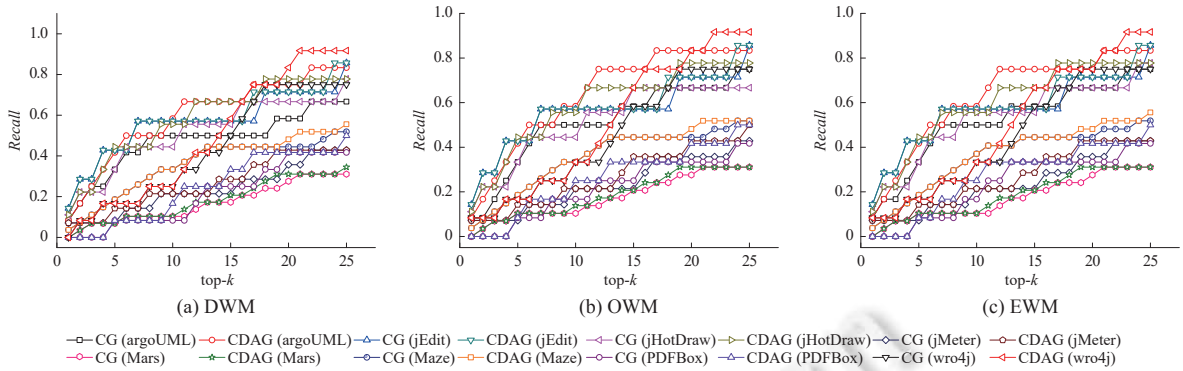

图 15 CDAG vs. CG (*Recall*@top-*k*, *k* 为整数形式)

为了减少图的数量, 图 14 和图 15 的每个子图都包含了 CDAG 和 CG 在 8 个系统上的对比结果. 方法名后面括号内的是软件名 (如 CG (argoUML)), 比较需要在同一个软件的 CDAG 和 CG 之间进行. 从图 14 和图 15 的结果可知, 在所有的系统上、在所有的阈值处, CDAG 都优于相应的 CG.

从 CDAG 与 CG 的对比结果可知: 在检查不超过前 15% (或 top-25) 的节点时, 使用动态分析对结果进行优化有助于提升 CDAG 方法的性能.

RQ4: 不同的赋权方式对 CDAG 的性能是否有影响?

本文使用 3 种方式为 CCN 中不同类型的耦合赋权. 从公式 (5) 可知, 边权的差异会直接影响类的 *GEN* 值. 本节将检验不同的赋权方式对 CDAG 方法性能的影响.

从图 6–图 8 可知, 在不同的赋权方式下, CDAG 在同一个系统的某个特定阈值处并非都具有相同的值. 例如在 argoUML 的阈值 k=2% 处, CDAG 在不同赋权方式下的值分别是 0.500 (DWM)、0.667 (OWM) 和 0.583 (EWM), 这些值互不相同. 这表明不同的赋权方式对结果有一定的影响. 此外, 我们使用 Friedman 检验进一步分析在整个测试集 (8 个系统) 上, 不同赋权方式对 CDAG 性能的影响. CDAG 在不同赋权方式下的平均排名如图 16 所示, 其相应的 P 值 (显著性水平) 均远大于 0.05, 这表明我们的结果接受原假设, 即不同方法之间没有显著差异.

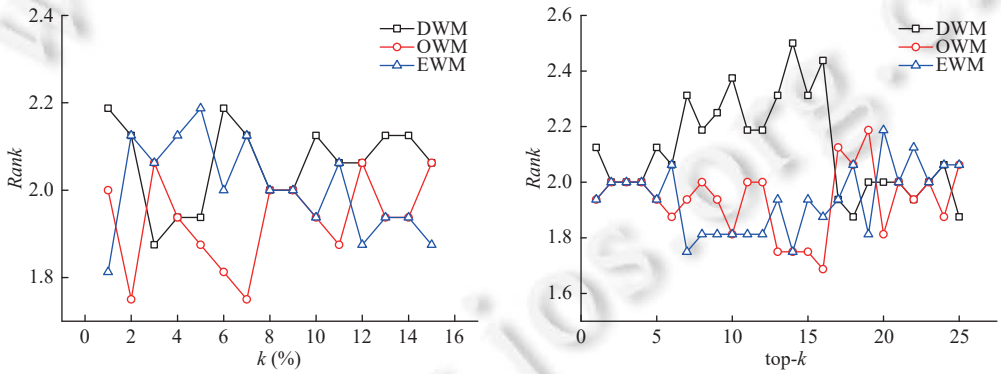

图 16 CDAG 方法在不同赋权方式下的平均排名

根据 Friedman 检验的结果及相应的 P 值, 我们可以得出结论: 不同的赋权方式对 CDAG 的性能没有显著影响.

RQ5: 在运行效率方面, CDAG 是否可行?

作为一种实用的关键类识别方法, CDAG 需要具备在不同规模软件系统中高效运行的能力. 为了评估 CDAG 的效率, 我们跟踪了其在 8 款实验软件上进行关键类识别时的时间消耗.

从图 1 可知, CDAG 的时间消耗主要来源于 5 个部分: (1) 提取结构信息, 并构建软件网络; (2) 计算所有类的

GEN 值, 并对类按照 *GEN* 值进行降序排列; (3) 获取测试用例; (4) 执行测试用例以获取执行轨迹; (5) 分析执行轨迹以获取方法调用关系及类集, 进而对类集进行扩充, 并对结果进行优化.

从表 5 可知, 在 8 个实验系统上, CDAG 均可以在 35.5 min 内完成关键类识别任务. CDAG 在 jEdit 系统上耗时最多, 约 35.46 min; 在 wro4j 系统上耗时最少, 仅用了约 3.53 min. 此外, 因为我们生成的测试用例数量比较大, 执行一个测试用例的耗时平均在 22.98 s. 为了提高实验的效率, 测试用例的执行是由实验室的 10 位硕士生在同一时间段内共同完成的. 换言之, 表 5 中 (4) 这个部分的耗时是这 10 个硕士生中耗时最多的那个学生的用时. 在表 5 中, PDFBox 和 wro4j 第 (3) 部分的耗时为 0, 这主要是因为它们是非 GUI 软件, 我们使用了源码仓库中自带的测试脚本, 这些脚本的执行无需 10 名硕士生的参与. 此外, 构建网络时我们用的是 DWM 这种赋权方式.

在实验过程中我们发现, 相对于软件的静态分析而言, 软件的动态分析是很耗时间的, 为了识别 jEdit (仅包含 1 091 个类) 中的关键类, CDAG 就已经耗时约 35.46 min. 若系统规模更大, 功能更复杂, CDAG 识别其中的关键类将耗时更多. 但是相对于某个软件的整个维护阶段而言, 这点时间消耗还是可以接受的. 此外, 我们还可以在系统上线前将获取执行轨迹的脚本嵌入其中, 这样便可在系统运行时收集执行轨迹, 从而跳过“获取测试用例”和“执行测试用例”这两个最为耗时的环节, 大幅节省方法运行的时间.

表 5 CDAG 在各实验系统上的时间消耗 (s)

耗时部分	argoUML	jEdit	jHotDraw	jMeter	Mars	Maze	PDFBox	wro4j
(1)	37.3	43.4	17.3	12.4	40.2	1.1	28.5	13.2
(2)	22.8	112.4	21.8	2.9	72.7	0.3	183.8	24.3
(3)	1 028.7	1 623.1	819.1	1 664.8	945.3	648.6	0	0
(4)	282	324	210	225	528	108	299.5	169.3
(5)	67.2	24.8	9.1	9.4	387.7	24.5	1 440.3	4.99
总耗时	1 438.0	2 127.7	1 077.3	1 914.5	1 973.9	782.5	1 952.1	211.79

根据对 CDAG 在实验系统上的时间消耗的分析, 我们可以得出结论: CDAG 在运行效率方面是可以接受的.

2.5 有效性分析

本节从构造有效性、内部有效性和外部有效性 3 个方面讨论实验所得结果的有效性.

(1) 构造有效性

构造有效性主要关注实验中使用的指标是否准确度量了想要度量的概念. 本文使用 *GEN* 评价类的重要性, 并使用 *Recall* 评价方法的性能. 其中, *GEN* 从软件结构角度评价类的重要性; *Recall* 是同类工作中普遍采用的评价指标, 主要用于度量软件中被成功识别的关键类的比例. 为了保障 *GEN* 和 *Recall* 值计算的准确性, 减少构造有效性威胁, 我们对所编写的代码进行了充分的测试.

(2) 内部有效性

内部有效性主要关注实验过程中一些处理方式对结果的影响. 在第 1.3 节中, 对于一些类节点由于具有相同的重要性而无法准确排序的问题, 我们采用了“求均值位置”的方式, 这实际上求的是一个概率意义上的期望位置. 当然此处也可以采用随机确定位置的方式, 即: 在具有相同重要性的几个类节点中, 随机选一个作为它们中的第 1 个, 再在剩余的中随机选一个作为它们中的第 2 个, 以此类推. 但是这会引入一定的随机性, 导致求得的 *Recall* 值在两次独立的运行中结果值不同. 即使运行 1 000 次求均值, 也无法绝对避免两个独立的 1 000 次的均值具有不同结果的情况. 为了消除这种随机性, 我们采用了“求均值位置”的方式.

(3) 外部有效性

外部有效性主要关注结果的可扩展性. 本文仅使用了 8 个 Java 开发的开源项目作为实验对象. 因此, 本文实验得出的结论可能无法直接推广到其他非 Java 系统或闭源系统. 但是这 8 个系统广泛应用在关键类识别研究中, 它们来自不同的领域、规模各异, 具有一定的多样性和代表性, 在一定程度上可以减少这种有效性威胁. 实际上, 这种有效性威胁普遍存在于实证研究中. 在未来工作中, 我们计划将本研究扩展到闭源的 Java 系统以及非 Java 系统.

3 相关研究

在过去的 10 多年中, 研究者提出了大量的方法用于识别面向对象软件 (主要是 Java 软件) 中的关键类. 这些方法大致可以分为两类: 无监督学习方法和有监督学习方法.

无监督学习方法通常通过构建类粒度的软件网络, 进而构建评价类重要性的度量指标, 并通过对类的排序和过滤实现关键类的识别. Zaidman 等人[6,7]通过设计软件运行场景 (execution scenarios) 来收集软件的运行轨迹, 进而构建类依赖图, 并使用 HITS 算法计算类的重要性. Perin 等人[13]通过静态分析构建类依赖图, 并应用 PageRank 算法识别软件中的关键类. 周毓明等人[8]和王木生等人[14]使用类依赖图抽象类和类之间的依赖关系, 进而采用 PageRank、*h* 指数、*a* 指数等指标度量类的重要性, 并通过阈值识别关键类. Steidl 等人[15]提出了软件的依赖图模型, 并使用 HITS、PageRank 等多种指标度量依赖图中类节点的重要性. Meyer 等人[16]将软件抽象为无权无向网络, 并用 *k*-core 分解方法计算节点的核数 (coreness), 进而识别候选关键类. 姜淑娟等人[18]采用有限状态机模型来表示软件系统, 并采用“唯一的输入/输出序列”来量化类的重要性. Pan 等人[19]提出了一种识别关键类的 ICOOK 方法. 该方法构建了类级加权有向软件网络模型, 并提出了一种一般化 *k*-core 分解方法来计算节点的一般化核数 (generalized coreness), 进而以一般化核数为类的重要性, 识别候选关键类. Şora 等人[17,20]基于 PageRank 算法及其变体 (PageRankBR) 识别关键类, 并开发了一套关键类推荐系统, 旨在帮助新开发人员更快地理解项目结构. Do Nascimento Vale 等人[21]基于对软件系统执行路径的动态分析, 提出了一个名为 Keecle 的半自动方法, 用于识别系统中的关键类. Pan 等人[10]提出了一种基于多层软件网络的关键类识别方法 ElementRank. 该方法构建软件类级的多层软件网络, 并使用 PageRank 算法计算每一层网络中类节点的重要性, 进而通过聚合类在各层的重要性值, 得到类最终的重要性值. Liu 等人[23]提出了一种关键类识别方法 PageRankIVOL. 该方法在 PageRank 的基础上进一步考虑了边权和网络节点数对重要性传递的影响. Pan 等人[5]在 PageRankIVOL 的基础上提出了一种关键类识别方法 Pride. 该方法在随机跳转部分进一步考虑了网络中节点的度、加权度、网络的总度及总加权度的影响. Li 等人[22]提出了一种识别关键类的 MinClass 方法. 该方法将软件抽象为类依赖网络, 并基于熵的概念构建了一个评价类重要性的指标 *OSE*. Pan 等人[24]提出了一种关键类识别方法 iFit. 该方法构建了类级的加权有向网络模型, 并基于 PageRank 算法和物理中场的概念提出了一个度量类重要性的指标 *CG*.

有监督学习方法依赖于度量指标集和机器学习技术构建分类器以预测关键类. Osman 等人[25]使用一组设计指标 (design metrics) 刻画类的特征, 并将这些指标作为机器学习分类器的特征来训练关键类预测模型. Thung 等人[26]构建了一组无权网络指标, 并与 Osman 等人的设计指标结合, 作为机器学习分类器的特征以训练关键类预测模型. Yang 等人[27]提出了一种基于机器学习的关键类识别方法 MCCondenser, 从而将逆向工程得到的类图压缩得更加紧凑. 周纯英等人[28]提出了一种基于图神经网络的关键类识别方法. 然而, 有监督学习方法依赖于带有标签的数据集. 若一个系统不存在带标签的数据集, 则现有方法都将无法使用. 同时, 现有的数据集存在严重的“类不平衡问题”. 例如, 在 jEdit 中, 关键类的数量仅占系统总类数的 0.641 6%. 这使得如何划分“训练集”和“测试集”成为一个难题, 同时也限制了这类方法的发展.

本文提出的 CDAG 方法不依赖于带标签的数据集和机器学习算法来构建分类器, 因而属于无监督学习方法. 但是, CDAG 与现有的无监督学习方法又存在一些区别, 主要包括: 1) 现有工作基本都是依赖于类之间的直接依赖关系来构建类重要性度量指标. 例如, *h* 指数[8,14]、*a* 指数[8,14]、核数[16]、一般化核数[19]之类的指标主要考虑了节点本身的度 (直接相连的节点数) 或加权度 (直接相连的边上的权值和), 基于 PageRank 算法的指标[5,10,13,15,17,20,23,24]主要考虑了直接相连的节点之间的“投票”关系. 这些方法均未考虑没有直接边相连的两个节点之间的相互影响, 也未考虑“某个节点的邻居节点的度存在差异”的事实. 我们提出的 *GEN* 指标综合考虑了类之间的间接耦合和邻居节点度分布的多样性对类重要性的影响, 因而可以更加全面地刻画软件网络中类节点的结构特征及类的重要性. 2) 目前基于动态分析的关键类识别方面的工作极少[6,7,21], 且均未提供可复用的测试用例集, 也未提供测试用例的获取方法, 导致工作的可重复性较差. 此外, 现有工作也未对动态获取的类集进行扩充, 存在缺失关键类的风险. 我们提出的测试用例获取方法具有一定的普适性, 可用于后续动态分析相关的工作. 同时, 本文提出的类集扩

充策略可以有效缓解关键类缺失的问题. 3) 现有工作未能将动态分析和静态分析结合起来, 以至于无法利用动态分析方法“准”的特点来排除“死代码”, 也无法利用静态分析方法“全”的特点来规避关键类缺失的问题. 本文使用动态分析技术对基于静态分析的初步结果进行优化, 从而部分解决静态分析构建的软件网络“不准”的问题, 提高了关键类识别的准确性.

4 结论和下一步工作

本文提出了一种基于动态分析及引力公式的关键类识别方法, 通过静态分析构建类级加权有向软件网络, 并基于熵和引力公式构建引力熵指标度量类的重要性, 进而实现类重要性的初步排序. 同时, 本文引入动态分析技术收集运行时类之间真实的交互关系, 进而对初步排序的结果进行优化, 并通过设定阈值来过滤非关键类, 得到候选的关键类. 8 个开源 Java 软件上的实验结果表明, 在检查不超过前 15% (或 top-25) 的节点时, CDAG 从整体而言显著优于其他对比方法; 使用动态分析对结果进行优化有助于提升 CDAG 方法的性能; 不同的赋权方式对 CDAG 的性能没有显著影响; CDAG 在运行效率上是可以接受的.

在下一步工作中, 我们计划在其他非 Java 软件或闭源软件中检验 CDAG 方法的有效性. 此外, 我们拟研究静态分析与动态分析更好的结合方式 (如用动态分析优化软件网络的边权), 以期进一步提高关键类识别的效果.

References

[1] Bennett KH, Rajlich VT. Software maintenance and evolution: A roadmap. In: Proc. of the 2000 Conf. on the Future of Software Engineering. Limerick: ACM, 2000. 73–87. [doi: 10.1145/336512.336534]

[2] Cornelissen B, Zaidman A, van Deursen A, Moonen L, Koschke R. A systematic survey of program comprehension through dynamic analysis. IEEE Trans. on Software Engineering, 2009, 35(5): 684–702. [doi: 10.1109/TSE.2009.28]

[3] Yang YM, Xia X, Lo D, Bi TT, Grundy J, Yang XH. Predictive models in software engineering: Challenges and opportunities. ACM Trans. on Software Engineering and Methodology, 2022, 31(3): 56. [doi: 10.1145/3503509]

[4] Maalej W, Tiarks R, Roehm T, Koschke R. On the comprehension of program comprehension. ACM Trans. on Software Engineering and Methodology, 2014, 23(4): 31. [doi: 10.1145/2622669]

[5] Pan WF, Ming H, Kim DK, Yang ZJ. Pride: Prioritizing documentation effort based on a PageRank-like algorithm and simple filtering rules. IEEE Trans. on Software Engineering, 2023, 49(3): 1118–1151. [doi: 10.1109/TSE.2022.3171469]

[6] Zaidman A, Calders T, Demeyer S, Paredaens J. Applying webmining techniques to execution traces to support the program comprehension process. In: Proc. of the 9th European Conf. on Software Maintenance and Reengineering. Manchester: IEEE, 2005. 134–142. [doi: 10.1109/CSMR.2005.12]

[7] Zaidman A, Demeyer S. Automatic identification of key classes in a software system using webmining techniques. Journal of Software Maintenance and Evolution: Research and Practice, 2008, 20(6): 387–417. [doi: 10.1002/smr.370]

[8] Zhou YM, Xu BW. Dependence structure analysis-based approach for measuring importance of classes. Journal of Southeast University (Natural Science Edition), 2008, 38(3): 380–384 (in Chinese with English abstract). [doi: 10.3969/j.issn.1001-0505.2008.03.004]

[9] Pan WF, Ming H, Chang CK, Yang ZJ, Kim DK. ElementRank: Ranking Java software classes and packages using a multilayer complex network-based approach. IEEE Trans. on Software Engineering, 2021, 47(10): 2272–2295. [doi: 10.1109/TSE.2019.2946357]

[10] Pan WF, Kessentini M, Ming H, Yang ZJ. EASE: An effort-aware extension of unsupervised key class identification approaches. ACM Trans. on Software Engineering and Methodology, 2024, 33(4): 84. [doi: 10.1145/3635714]

[11] Pan WF, Wu W, Ming H, Kim DK, Yang JK, Liu RC. Improving the condensing of reverse engineered class diagrams using weighted network metrics. In: Proc. of the 46th IEEE/ACM Int’l Conf. on Software Engineering: Companion Proc. Lisbon: ACM, 2024. 374–375. [doi: 10.1145/3639478.3643520]

[12] Yau SS, Collofello JS. Design stability measures for software maintenance. IEEE Trans. on Software Engineering, 1985, SE-11(9): 849–856. [doi: 10.1109/TSE.1985.232544]

[13] Perin F, Renggli L, Ressia J. Ranking software artifacts. In: Proc. of the 4th Workshop on FAMIX and Moose in Reengineering, 2010. 1–4. [doi: 10.7892/boris.4966]

[14] Wang MS, Lu HM, Zhou YM, Xu BW. Identifying key classes using *h*-index and its variants. Journal of Frontiers of Computer Science and Technology, 2011, 5(10): 891–903 (in Chinese with English abstract).

[15] Steidl D, Hummel B, Juergens E. Using network analysis for recommendation of central software classes. In: Proc. of the 19th Working

Conf. on Reverse Engineering. Kingston: IEEE, 2012. 93–102. [doi: 10.1109/WCRE.2012.19]

[16] Meyer P, Siy H, Bhowmick S. Identifying important classes of large software systems through *k*-core decomposition. Advances in Complex Systems, 2014, 17(77): 1550004. [doi: 10.1142/S0219525915500046]

[17] Şora I. A pagerank based recommender system for identifying key classes in software systems. In: Proc. of the 10th IEEE Jubilee Int'l Symp. on Applied Computational Intelligence and Informatics. Timisoara: IEEE, 2015. 495–500. [doi: 10.1109/SACI.2015.7208254]

[18] Jiang SJ, Ju XL, Wang XY, Li HY, Zhang YM, Liu YQ. Measuring the importance of classes using UIO sequence. Acta Electronica Sinica, 2015, 43(10): 2062–2068 (in Chinese with English abstract). [doi: 10.3969/j.issn.0372-2112.2015.10.027]

[19] Pan WF, Song BB, Li KS, Zhang KJ. Identifying key classes in object-oriented software using generalized *k*-core decomposition. Future Generation Computer Systems, 2018, 81: 188–202. [doi: 10.1016/j.future.2017.10.006]

[20] Şora I, Chirila CB. Finding key classes in object-oriented software systems by techniques based on static analysis. Information and Software Technology, 2019, 116: 106176. [doi: 10.1016/j.infsof.2019.106176]

[21] Do Nascimento Vale L, de Almeida Maia M. Key classes in object oriented systems: Detection and assessment. Int'l Journal of Software Engineering and Knowledge Engineering, 2019, 29(10): 1439–1463. [doi: 10.1142/S0218194019500451]

[22] Li H, Wang T, Pan WF, Wang MC, Chai CL, Chen PY, Wang JL, Wang J. Mining key classes in Java projects by examining a very small number of classes: A complex network-based approach. IEEE Access, 2021, 9: 28076–28088. [doi: 10.1109/ACCESS.2021.3058450]

[23] Liu SR, Guo ZQ, Li YH, Lu HM, Chen L, Xu L, Zhou YM, Xu BW. Prioritizing code documentation effort: Can we do it simpler but better? Information and Software Technology, 2021, 140: 106686. [doi: 10.1016/j.infsof.2021.106686]

[24] Pan WF, Du X, Ming H, Kim DK, Yang ZJ. Identifying key classes for initial software comprehension: Can we do it better? In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 1878–1889. [doi: 10.1109/ICSE48619.2023.00160]

[25] Osman MH, Chaudron MRV, Putten PVD. An analysis of machine learning algorithms for condensing reverse engineered class diagrams. In: Proc. of the 2013 IEEE Int'l Conf. on Software Maintenance. Eindhoven: IEEE, 2013. 140–149. [doi: 10.1109/ICSM.2013.25]

[26] Thung F, Lo D, Osman MH, Chaudron MRV. Condensing class diagrams by analyzing design and network metrics using optimistic classification. In: Proc. of the 22nd Int'l Conf. on Program Comprehension. Hyderabad: ACM, 2014. 110–121. [doi: 10.1145/2597008.2597157]

[27] Yang XL, Lo D, Xia X, Sun JL. Condensing class diagrams with minimal manual labeling cost. In: Proc. of the 40th IEEE Annual Computer Software and Applications Conf. Atlanta: IEEE, 2016. 22–31. [doi: 10.1109/COMPSAC.2016.83]

[28] Zhou CY, Zeng C, He P, Zhang Y. GKCI: An improved GNN-based key class identification method. Ruan Jian Xue Bao/Journal of Software, 2023, 34(6): 2509–2525 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6846.htm [doi: 10.13328/j.cnki.jos.006846]

[29] Brito E Abreu F, Goulão M. Coupling and cohesion as modularization drivers: Are we being over-persuaded? In: Proc. of the 5th European Conf. on Software Maintenance and Reengineering. Lisbon: IEEE, 2001. 47–57. [doi: 10.1109/CSMR.2001.914968]

[30] Brito E Abreu F, Pereira G, Sousa P. A coupling-guided cluster analysis approach to reengineer the modularity of object-oriented systems. In: Proc. of the 4th European Conf. on Software Maintenance and Reengineering. Zurich: IEEE, 2000. 13–22. [doi: 10.1109/CSMR.2000.827300]

[31] Kang DZ, Xu BW, Lu JJ, Chu WC. A complexity measure for ontology based on UML. In: Proc. of the 10th IEEE Int'l Workshop on Future Trends of Distributed Computing Systems. Suzhou: IEEE, 2004. 222–228. [doi: 10.1109/FTDCS.2004.1316620]

[32] Newton I. The Principia: Mathematical Principles of Natural Philosophy. Berkeley: University of California Press, 1999.

[33] Ma LL, Ma C, Zhang HF, Wang BH. Identifying influential spreaders in complex networks based on gravity formula. Physica A: Statistical Mechanics and Its Applications, 2016, 451: 205–212. [doi: 10.1016/j.physa.2015.12.162]

[34] Liu ZM, Huang BN, Hu XG, Du PB, Sun QY. Blockchain-based renewable energy trading using information entropy theory. IEEE Trans. on Network Science and Engineering, 2024, 11(6): 5564–5575. [doi: 10.1109/TNSE.2023.3238110]

[35] Lee J, Kim DK, Kim S, Park S. Decomposing class responsibilities using distance-based method similarity. Frontiers of Computer Science, 2016, 10(4): 612–630. [doi: 10.1007/s11704-015-5001-5]

[36] Christakis NA, Fowler JH. Social contagion theory: Examining dynamic social networks and human behavior. Statistics in Medicine, 2013, 32(4): 556–577. [doi: 10.1002/sim.5408]

[37] Stanford Parser. 2024. https://stanfordnlp.github.io/CoreNLP/parser-standalone.html

[38] van Hoorn A, Waller J, Hasselbring W. Kieker: A framework for application performance monitoring and dynamic software analysis. In: Proc. of the 3rd ACM/SPEC Int'l Conf. on Performance Engineering. Boston: ACM, 2012. 247–248. [doi: 10.1145/2188286.2188326]

[39] Kieker. 2024. https://kieker-monitoring.net/

[40] García S, Fernández A, Luengo J, Herrera F. Advanced nonparametric tests for multiple comparisons in the design of experiments in

computational intelligence and data mining: Experimental analysis of power. Information Sciences, 2010, 180(10): 2044–2064. [doi: 10.1016/j.ins.2009.12.010]

附中文参考文献

[8] 周毓明, 徐宝文. 基于依赖结构分析的类重要性度量方法. 东南大学学报 (自然科学版), 2008, 38(3): 380–384. [doi: 10.3969/j.issn.1001-0505.2008.03.004]

[14] 王木生, 卢红敏, 周毓明, 徐宝文. 利用 *h* 指数及其衍生度量识别关键类. 计算机科学与探索, 2011, 5(10): 891–903.

[18] 姜淑娟, 鞠小林, 王兴亚, 李海洋, 张艳梅, 刘颖祺. 基于 UIO 序列的类重要性度量. 电子学报, 2015, 43(10): 2062–2068. [doi: 10.3969/j.issn.0372-2112.2015.10.027]

[28] 周纯英, 曾诚, 何鹏, 张龑. GKCI: 改进的基于图神经网络的关键类识别方法. 软件学报, 2023, 34(6): 2509–2525. http://www.jos.org.cn/1000-9825/6846.htm [doi: 10.13328/j.cnki.jos.006846]

作者简介

潘伟丰, 博士, 教授, CCF 高级会员, 主要研究领域为软件工程, 复杂网络, 数据挖掘.

杨燕微, 硕士生, 主要研究领域为软件工程, 复杂网络.

杨子江, 博士, 教授, 主要研究领域为软件工程, 数据挖掘.

姜波, 博士, 教授, CCF 杰出会员, 主要研究领域为软件工程, 服务计算.

王家乐, 博士, 副教授, CCF 专业会员, 主要研究领域为软件工程, 数据挖掘.

杨柏林, 博士, 教授, CCF 杰出会员, 主要研究领域为数据挖掘.