软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007556] [CSTR: 32375.14.jos.007556]

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

面向复杂头文件的自动化分解与重构方法*

王 玥[1,2], 孙嘉旋[1,2], 邹艳珍[1,2], 李宇轩[1,2], 常文辉[1,2], 谢 冰[1,2]

[1](高可信软件技术教育部重点实验室 (北京大学), 北京 100871)

[2](北京大学 计算机学院, 北京 100871)

通信作者: 邹艳珍, E-mail: zouyz@pku.edu.cn

摘 要: 许多代码文件随着软件演化逐渐膨胀并承担了过多职责, 严重影响了软件的可维护性和可理解性. 开发者常需要重构这些文件, 将一个大的代码文件分解成多个较小的子文件. 现有研究工作主要聚焦类文件的分解重构, 并不完全适用于分解复杂头文件. 这是因为分解头文件面临一些独有的挑战: 既需要考虑整个软件项目的构建依赖以降低编译成本, 也需要确保分解后的子文件之间不会存在循环依赖. 为此, 提出了一种面向复杂头文件的自动化分解与重构方法——HeaderSplit. 该方法首先为复杂头文件构造蕴含多种代码关系的代码元素图, 其中就包括体现项目构建依赖的共同使用关系; 然后通过节点合并与多视图聚类算法识别关联紧密的代码元素聚类; 随后引入启发式的循环依赖修正算法生成可行的文件分解方案. 用户确认分解方案后, HeaderSplit 能够自动执行重构, 生成新的子文件内容, 并更新软件项目内直接或间接引用原头文件的代码语句. 在合成复杂头文件与真实复杂头文件上对 HeaderSplit 进行评估, 结果表明: 1) HeaderSplit 在准确率上比现有方法提升了 11.5%, 并且具有更强的跨软件项目稳定性; 2) HeaderSplit 分解得到的子文件模块度更高且无循环依赖, 具有更好的架构设计; 3) 使用 HeaderSplit 分解复杂头文件可以降低其演化历史中 15%–60% 的重编译成本; 4) HeaderSplit 可以高效实施自动化重构, 在 5 min 以内完成百万行软件项目内的头文件分解重构, 具有很高的实用价值.

关键词: 软件维护; 软件重构; 头文件

中图法分类号: TP311

中文引用格式: 王玥, 孙嘉旋, 邹艳珍, 李宇轩, 常文辉, 谢冰. 面向复杂头文件的自动化分解与重构方法. 软件学报. http://www.jos.org.cn/1000-9825/7556.htm

英文引用格式: Wang Y, Sun JX, Zou YZ, Li YX, Chang WH, Xie B. Automated Approach for Decomposing and Refactoring God Header Files. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7556.htm

Automated Approach for Decomposing and Refactoring God Header Files

WANG Yue[1,2], SUN Jia-Xuan[1,2], ZOU Yan-Zhen[1,2], LI Yu-Xuan[1,2], CHANG Wen-Hui[1,2], XIE Bing[1,2]

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

[2](School of Computer Science, Peking University, Beijing 100871, China)

Abstract: Many code files become oversized and take on excessive responsibilities as software evolves, which severely affects software maintainability and comprehensibility. Developers often need to refactor such files by decomposing a large code file into several smaller ones. Existing studies mainly focus on class file decomposition and are not fully applicable to decomposing complex header files. This is because header file decomposition faces unique challenges. It needs to consider the build dependencies of the entire software project to reduce compilation cost and ensure that the decomposed files are free of cyclic dependencies. To address these challenges, this study proposes an automated approach for decomposing and refactoring complex header files, HeaderSplit. It first constructs a code element graph that captures multiple types of code relationships, including co-usage relationships that reflect project build dependencies. Then, a

* 基金项目: 国家重点研发计划 (2023YFB4503803)
收稿时间: 2025-04-03; 修改时间: 2025-06-05, 2025-08-13; 采用时间: 2025-09-23; jos 在线出版时间: 2026-01-07

node coarsening process and a multi-view graph clustering algorithm are applied to identify clusters of closely related code elements. A heuristic algorithm is further introduced to eliminate cyclic dependencies in the clustering results. After the decomposition plan is confirmed, HeaderSplit automatically performs the refactoring, generating new sub-header files and updating the include statements in all code files that directly or indirectly include the original header file. HeaderSplit is evaluated on both synthetic and real complex header files. The results are as follows. 1) HeaderSplit improves accuracy by 11.5% compared with existing methods and demonstrates higher cross-project stability. 2) The decomposed sub-files have higher *Modularity* and no cyclic dependencies, indicating better architectural design. 3) Using HeaderSplit to decompose complex header files can reduce recompilation costs in their evolution history by15%–60%. 4) HeaderSplit efficiently performs automated refactoring, completing the decomposition and refactoring of header files in large-scale software projects with millions of lines of code within five minutes, showing high practical value.

Key words: software maintenance; software refactoring; header file

1 引 言

代码重构在生命周期较长的软件项目中起着至关重要的作用. 在软件的开发和维护过程中, 为了引入新功能、适应新应用场景以及移除软件缺陷, 软件系统不断演化, 其复杂性也不断增加. 与此同时, 在版本演化的过程中, 过短的开发周期往往会使得开发人员减少对高质量的软件设计、良好的开发习惯以及测试覆盖率等方面的关注, 从而导致软件逐渐偏离其最初的设计[1]. 而重构能够使开发者将设计不佳甚至混乱的代码转化为结构良好的代码, 进而增强软件的健壮性、可复用性以及其他关键特性[2].

随着软件演化, 代码文件往往会因为承担过多职责而变得臃肿, 导致代码理解和维护困难. 在与某嵌入式软件开发团队合作的过程中, 我们发现他们备受复杂头文件的困扰. 头文件是 C/C++等编程语言中常见的一种文件类型, 通常以.h 为后缀, 主要作用是提供程序中使用的函数、变量、数据结构等的声明, 使得多个源文件(.c 或.cpp 文件) 可以共享这些声明并进行模块化开发. 然而, 在该团队的持续开发过程中, 开发者常因交付压力将新的声明添加到已有头文件中, 使头文件的规模随时间急剧膨胀、复杂程度逐步上升. 这样的复杂头文件极大地影响了项目的编译效率. 当一个复杂头文件被修改时, 所有包含 (include) 该头文件的源文件, 无论是否直接使用了修改内容, 都需要重新编译. 而复杂头文件往往被许多源文件包含, 其修改会触发大量编译单元的重编译, 对软件的演化和维护产生了严重影响. 图 1 展示了一个复杂头文件的示例. 头文件 guc.h 来自数据库软件项目 PolarDB-for-PostgreSQL, 它包含 63 个宏定义、19 个数据结构定义和 493 个变量或函数的声明. 每当它被修改, 388 个直接或间接依赖该文件的代码文件都需要重新被编译, 总计超过 74 万行代码. 这样的复杂头文件因其规模庞大、功能复杂、涉及的代码依赖广泛, 严重降低了软件编译效率, 为开发者理解与维护软件代码带来极大的挑战.

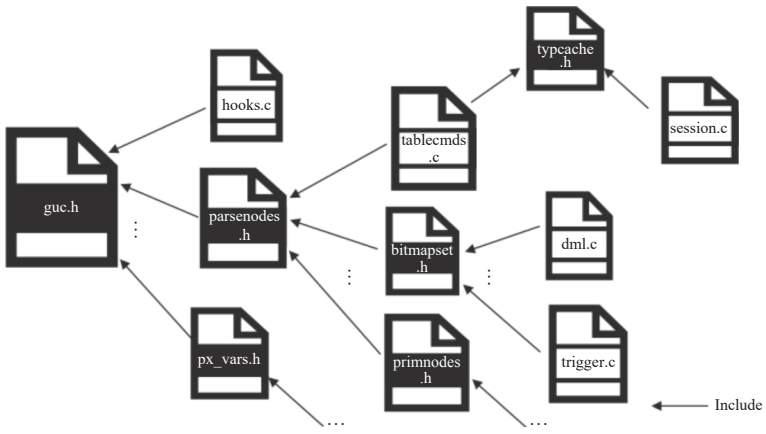


图 1 复杂头文件示例: guc.h

为了解决复杂头文件问题, 我们试图借鉴任务较为相似的复杂类 (god class) 重构研究. 这些研究设计了多种代码度量来衡量类内实体相似度, 例如共享变量[3]、方法调用[4]、文本相似度和位置相似度[5], 然后借助聚类算法从复杂类中提取内聚性更高的类. 代表性地, Bavota 等人[4]基于类内方法间的结构相似度、概念相似度与调用依赖相似度, 为复杂类构建带权图, 并识别其中的连通子图从而得到复杂类分解结果; Chen 等人[5]提出了位置相似度, 并借助自定义的分步聚类技术, 更精准地分解复杂类. 然而, 这些类文件分解方法并不直接适用于分解复杂头文件: (1) 这些研究工作主要关注类内方法 (method) 间的内部关系, 但分解头文件还需要考虑文件外部的构建依赖关系[6], 以提高编译效率; (2) 这些方法通常通过加权求和聚合多种代码关系, 然而, 确定每种关系的权重非常困难, 而且不同项目的最佳权重也可能会有差异[4]; (3) 这些方法忽视了分解后的文件之间可能存在的循环依赖, 这是一种架构反模式[7], 在分解头文件的场景下还可能会引发编译错误. 此外, 自动化分解重构复杂头文件并非易事, 需保留原有代码逻辑并正确更新项目内的 include 语句.

为了解决上述问题, 本文提出了 HeaderSplit, 一种利用多种类型代码关系与多视图聚类的复杂头文件分解方法, 并将其实现为自动化重构工具. HeaderSplit 首先解析待分解头文件内容并构造对应的代码元素图, 该图不仅考虑了文件内部的依赖关系和文本相似关系, 还引入了反映整个项目构建依赖的共同使用关系; 然后基于依赖关系对部分节点进行合并, 并使用多视图聚类算法, 根据代码元素的文本相似关系和共同使用关系将紧密相关的代码元素聚合在一起; 对于聚类结果中可能存在的循环依赖问题, 本文提出一种基于启发式搜索的方法进行修正, 生成恰当的头文件分解方案. 一旦用户确认了分解方案, HeaderSplit 会自动执行重构, 为每个子头文件生成与原始头文件逻辑一致的代码内容, 并更新直接或间接依赖原始头文件的其他代码文件中的 include 语句, 确保每个文件只包含其依赖的代码元素所在的子头文件.

本文在合成复杂头文件与真实复杂头文件上评估了本文算法与工具. 真实的复杂头文件数据来源于本文对 GitHub 上 557 个软件项目的实证研究. 实验结果表明: 1) 与现有方法相比, HeaderSplit 在分解合成复杂头文件时的准确率提升了 11.5%, 并且具有更强的跨软件项目稳定性; 2) HeaderSplit 分解真实复杂头文件时得到的子文件模块度更高且无循环依赖, 具有更好的架构设计; 3) 使用 HeaderSplit 分解复杂头文件可以降低其演化历史中 15%–60% 的重编译成本; 4) HeaderSplit 可以高效实施自动化重构, 在包含数百万行代码的大型项目中, 能够在 81–193 s 内分解重构数千行代码的复杂头文件.

本文的主要贡献为: 1) 对复杂头文件在开源社区的分布情况进行实证研究, 证明该问题是广泛存在的; 2) 分析了对复杂头文件进行分解的技术挑战, 提出了分解与重构方法, 与现有方法相比达到了更高的准确率和模块度, 并且能够节省 15%–60% 的历史重新编译成本; 3) 实现了复杂头文件自动化分解重构工具, 能够根据分解方案高效地生成子头文件并修改项目内的相关包含语句.

2 复杂头文件问题的普遍性与重要性

复杂头文件对软件的演化和维护产生了不良影响, 然而其影响范围仍是未知的. 因此, 本文首先进行了一项实证研究[8], 旨在探讨复杂头文件问题在开源项目中的普遍性与重要性.

2.1 数据收集与处理

本文首先从 GitHub 上下载了项目规模超过 10 MB 且收藏数超过 500 的 C 语言软件项目作为研究数据集. 其次, 过滤掉派生软件项目 (forked project), 并保留代码文件数量超过 100 且总代码行数超过 10 万的软件项目. 最终得到的数据集包含 557 个 C 语言软件项目, 涉及操作系统、数据库、开发工具包、网络服务等多个领域. 这些软件项目具有一定的复杂性, 与工业软件开发中要进行复杂头文件分解的实际场景更为相似.

表 1 展示了本文数据集中软件项目的代码文件数与代码行数的相关统计信息. 总体而言, 本文数据集中的软件项目平均包含 2 821 个代码文件, 项目总代码行数平均超过百万行; 代码文件数与代码行数的中位数是 869 和 426 905. 更详细的数据集信息请参见: https://zenodo.org/records/10989833.

表 1 数据集统计信息

详细信息	文件类型	平均值	最小值	中位数	最大值
代码文件数量	头文件	1287	12	376	23551
	源文件	1535	10	447	49994
	全部文件	2821	110	869	55733
代码行数	头文件	442662	1843	96860	15045600
	源文件	885379	1393	283991	22637270
	全部文件	1328042	101416	426905	31618270

2.2 实验过程与结果

为了研究复杂头文件问题在开源软件中的普遍性与重要性, 本文设计了识别复杂头文件的重要指标——代码规模与影响范围, 并人工标注了一些头文件用于估计识别复杂头文件的指标阈值, 随后分析复杂头文件的分布与其在软件演化过程中的影响.

● 头文件解析. 本文使用代码解析工具 tree-sitter (详见 https://tree-sitter.github.io/tree-sitter/) 解析数据集中的所有头文件, 计算每个文件的代码规模与影响范围. 头文件的代码规模是指头文件中代码元素的数量. 代码元素指的是无法被进一步分解的独立代码单元, 包括宏定义、数据类型定义 (struct、enum、union、typedef)、变量与函数的声明与定义. 使用代码元素数量而非代码行数来衡量代码规模是因为前者更能反映出头文件的复杂性及其分解重构潜力. 头文件的影响范围是指依赖该头文件的代码文件总行数占整个项目代码行数的比例. 这里的依赖既包括直接依赖, 也包括经由其他头文件引用产生的间接依赖. 这个指标衡量了当一个头文件被修改时, 整个项目中需要重新编译的代码行数百分比. tree-sitter 成功解析了 541 个软件项目中的 761 999 个头文件.

● 头文件采样与标注. 为了识别复杂头文件, 本文选取了 6 个文档完善的软件项目 (表 2), 从中采样了一些头文件进行人工标注. 本文使用分层采样方法: 针对每个软件项目, 根据头文件的代码规模和影响范围划分区域, 每个区域的边界为 $i\times100 <$ 代码规模 $< (i+1)\times100$ 且 $j\times0.1\times100\% <$ 影响范围 $< (j+1)\times0.1\times100\%$ $(i>1, 1<j<10)$. 每个区域中若存在头文件, 则随机选取一个文件作为标注数据. 这种方法确保了采样过程能够覆盖不同代码规模和影响范围的头文件. 随后, 两名研究生根据样本头文件的内容与其影响的代码文件列表, 分别将每个样本头文件分类为“复杂头文件”或“非复杂头文件”. 对于分类结果存在分歧的样本, 由作者进行最终裁定. 图 2 热力图展示了数据集中头文件的代码规模与影响范围的联合分布, 可以看出只有极少数头文件同时具有较大的代码规模和广泛的影响范围; 图 2 中的散点展示了人工标注的样本头文件, 其中红色圆圈代表“复杂头文件”, 蓝色叉号表示“非复杂头文件”. 结果表明, 代码规模较大且影响范围较广的头文件更可能是复杂头文件.

表 2 人工标注的软件项目与典型复杂头文件

软件项目	项目规模	收藏数	典型复杂头文件	代码元素数量	文件影响范围 (%)
FreeRDP	57040	7742	settings.h	745	98.0
PolarDB-for-PostgreSQL	389425	2499	guc.h	575	55.0
SDL	140832	584	SDL_dynapi_overrides.h	769	67.8
SoftEtherVPN	540135	9758	Network.h	680	86.3
stress-ng	25993	838	stress-ng.h	610	90.9
wiredtiger	126937	1974	extern.h	1274	94.0

● 复杂头文件挖掘. 根据标注复杂头文件的分布情况, 本文将代码规模超过 400 个代码元素且影响范围超过项目内 40% 代码行的头文件视为复杂头文件. 图 2 中的虚线标出了对应阈值. 可以看出, 该阈值设置得相较于真实的复杂头文件分界线更为严格, 这有助于降低复杂头文件识别的误报率, 但也可能导致部分实际存在的复杂头文件未被识别. 因此, 本文对复杂头文件普遍性的估计与实际情况相比偏低.

基于上述实验流程, 本文共识别出 649 个复杂头文件, 来自 203 个软件项目. 复杂头文件数量占头文件总数 761 999 的比例不到 1%, 这可能是因为本文阈值选取较为严格. 从项目角度看, 成功解析的 541 个软件项目中有

203 个包含复杂头文件, 这表明约有 37.5% 的软件项目存在复杂头文件的问题. 这些软件项目通常只包含少量的复杂头文件, 有 163 个软件项目包含的复杂头文件个数不超过 3 个; 数据集中有 4 个项目包含超过 20 个复杂头文件, 人工检查发现, 这些项目为了支持多版本硬件设备创建了类似的头文件, 从而导致了复杂头文件数量较多. 本文还检查了这 649 个复杂头文件的修改提交历史 (commit history). 其中, 有 103 个文件的修改提交次数超过 100 次, 14 个文件甚至超过 5 000 次. 这些复杂头文件较大的代码规模、广泛的影响范围以及频繁的修改次数增加了软件维护和演化的负担, 对这些复杂头文件进行分解重构是很有必要的.

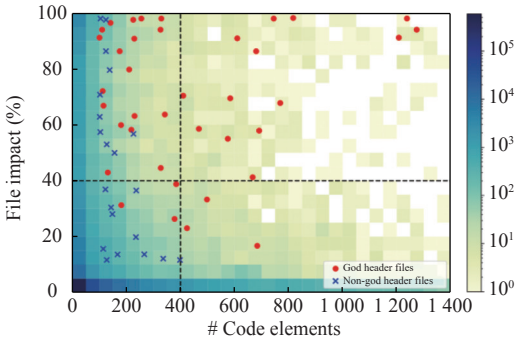


图 2 头文件代码规模与影响范围的联合分布

综上所述, 复杂头文件问题在开源 C 语言软件项目中广泛存在, 约有 37.5% 的软件项目受其影响; 许多复杂头文件在其生命周期中都经历了频繁的修改, 引起了大量的重编译开销, 增加了软件维护和演化的负担, 突显了分解重构复杂头文件的重要性.

3 复杂头文件自动化分解与重构方法

为了有效地对复杂头文件进行分解重构, 本文提出了一种利用多种类型代码关系与多视图聚类的自动化分解重构方法 HeaderSplit. 第 3.1 节给出方法的概览介绍, 之后各节将详细介绍方法的关键步骤.

3.1 方法概述

本文提出的复杂头文件自动化分解与重构方法流程如后文图 3 所示, 可以分为 5 个部分: 代码元素图构建、基于依赖关系的节点合并、多视图聚类、循环依赖修正和分解重构自动执行. HeaderSplit 首先解析复杂头文件, 提取代码元素及其依赖关系、文本相似关系和共同使用关系构建代码元素图 (第 3.2 节); 然后利用依赖关系合并紧密相关的代码元素节点 (第 3.3 节); 接着通过多视图聚类算法融合文本相似关系和共同使用关系对图进行划分 (第 3.4 节); 随后使用启发式算法解决聚类结果中的循环依赖问题 (第 3.5 节); 最后自动生成子头文件并更新项目内的 include 语句 (第 3.6 节).

3.2 代码元素图构建

为了分解复杂头文件, HeaderSplit 首先解析该文件中包含的代码元素以及代码元素之间的复杂关系, 将其表示成代码元素图以便后续分析与处理. 具体而言, 本文使用开源代码解析工具 tree-sitter 解析复杂头文件及其所属的软件项目. 抽取的代码元素包括宏定义、数据类型定义 (struct、enum、union、typedef)、变量与函数的声明与定义, 它们是无法被进一步分解的基本代码单元. 本文方法还抽取了 3 种与文件分解相关的代码关系, 分别是: 1) 依赖关系. 这指的是一个代码元素使用了另一个代码元素定义的名字 (定义-使用关系). 依赖关系强度较高的代码元素应当位于同一子头文件中, 否则将在子头文件间引入不必要的包含关系. 2) 文本相似关系. 该关系描述两个代码元素文本上的相似性. 文本相似的代码元素往往对应相同或类似的软件概念和功能, 将之放置于同一子头文件中可以提升代码的可读性. 3) 共同使用关系. 它表示了两个代码元素在源文件中被共同使用的频率. 经常被共同使用的代码元素更应当位于同一子头文件内, 否则使用它们的源文件需要包含更多子头文件, 增加了项目构建依

赖的复杂性并降低了重编译效率.

形式化地, 代码元素图被表示为 $G=(V,E_d,E_s,E_c)$, 其中 $V=\{v_i\}_{i=1}^{n}$ 表示复杂头文中的代码元素集合, E_d、E_s、E_c 分别表示依赖关系、文本相似关系和共同使用关系对应的边. 对于每种类型的边 $r\in\{d,s,c\}$, A^r 是该种类型边的邻接矩阵, $A^r_{i,j}$ 的值是代码元素 v_i 和 v_j 在代码关系 r 上的强度.

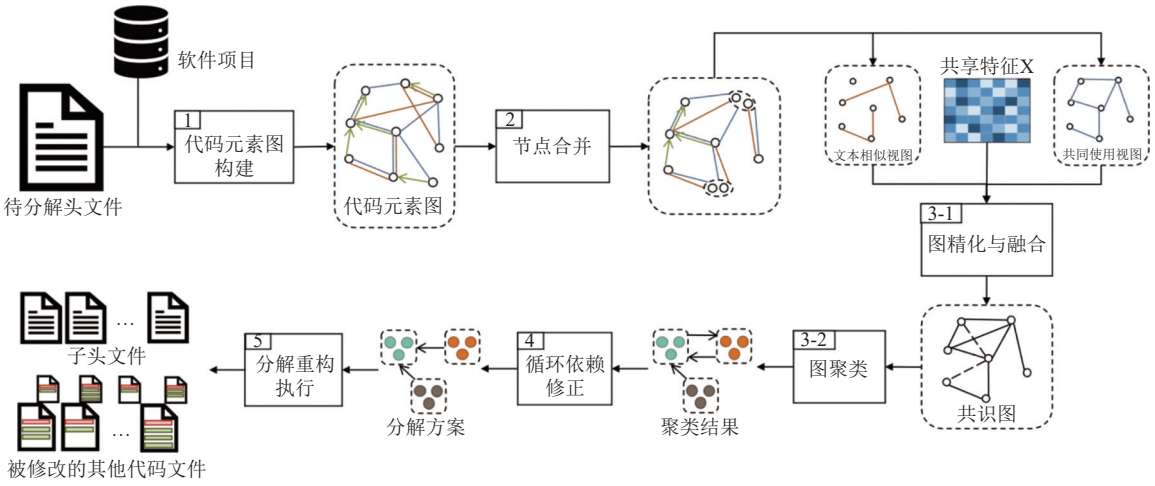


图 3 复杂头文件自动化分解与重构方法流程概览

本文在计算依赖关系的强度时借鉴了 Bavota 等人[4]提出的基于调用的依赖 (call-base dependence) 指标. 令 $Succesor_j$ 表示使用了 v_j 的代码元素集合, 那么, 有:

$$d_{i\to j}=\begin{cases}\dfrac{1}{Succesor_j}, & \text{如果}v_i\text{使用了}v_j\\ 0, & \text{否则}\end{cases} \tag{1}$$

$$A^d_{i,j}=\max(d_{i\to j},d_{j\to i}) \tag{2}$$

虽然依赖关系是有向的, 但其邻接矩阵被设计为对称矩阵以保证其可交换性. 如果 $A^d_{i,j}=1$, 则表示 v_j 仅被 v_i 使用, 或 v_i 仅被 v_j 使用, 这意味着 v_i 和 v_j 紧密相关, 应当属于同一个子头文件.

为了计算文本相似关系的强度, 本文方法对每个代码元素 v_i 中包含的标识符进行分词和词形还原, 并过滤掉“set”“get”等停用词, 得到对应的词汇集合 $Word_i$. 集合中的词汇可以体现与该代码元素相关的软件概念, 两个代码元素之间的文本相似关系强度则被设计为对应词汇集合的 Jaccard 距离:

$$A^s_{i,j}=\frac{|Word_i\cap Word_j|}{|Word_i\cup Word_j|} \tag{3}$$

对于共同使用关系, 令 $File_i$ 表示使用了代码元素 v_i 的源文件集合, 那么代码元素 v_i 和 v_j 之间的共同使用关系强度计算如下:

$$A^c_{i,j}=\frac{|File_i\cap File_j|}{|File_i\cup File_j|} \tag{4}$$

两个代码元素的共同使用关系强度越高, 表示它们更经常被一个代码文件同时调用, 这意味着它们在功能上更加紧密相关, 更有可能位于同一个子头文件中.

3.3 基于依赖关系的节点合并

代码元素图中的 3 种代码关系具有不同的特性: 依赖关系通常表示更强的连接, 而文本相似关系和共同使用关系数量更多、蕴含的信息更丰富. 因此, 本文在两个不同的阶段利用这些代码关系. 直观上, 如果两个代码元素仅依赖彼此, 它们应当被放置在同一个子头文件中; 而如果一个代码元素依赖许多其他代码元素, 它们是否应当位于同一个子文件中是不确定的, 需要引入对其他代码关系的考量再进行决策. 因此, 本节将具有较高依赖强度的代

码元素合并为一个节点, 确保紧密相关的代码元素在后续处理流程中始终位于同一个子文件中.

具体而言, 本文迭代地选取代码元素图中依赖强度最高的一对节点 v_i, v_j, 将它们合并得到一个新的节点 v'_i, 然后通过公式 (5) 更新节点 v'_i 与图中其他节点 v_k 之间的关系强度:

$$A^d_{i',k} = A^d_{k,i'} = \max\left(A^d_{i,k}, A^d_{j,k}\right),\ k \neq i 且 k \neq j \tag{5}$$

这一迭代合并的过程直到图中最高依赖强度低于设定阈值时停止. 此时得到了节点合并后的代码元素图, 记作 G'. G' 中的每个节点 v'_i 可能代表一个单独的代码元素或一组紧密依赖的代码元素集合. 此外, G' 的邻接矩阵中的每个值 $A^r_{i',j'}$ 都被赋值为节点 v'_i 和 v'_j 之间代码元素关系强度的最大值.

3.4 多视图聚类

在这一阶段, HeaderSplit 对节点合并后的代码元素图进行聚类, 旨在将功能相似的代码元素归于同一聚类. 为了实现这一目标, 本文方法在此阶段利用文本相似关系与共同使用关系, 因为这两种关系不仅更加稠密, 还蕴含丰富的潜在功能特征信息. 为有效地结合多种关系进行聚类, 本文采用了一种无监督多视图聚类算法 DuaLGR[9], 该算法基于输入图数据的内在结构进行自适应学习, 无需外部训练数据. 具体地, 该算法的输入是来自不同视图的共享特征和邻接矩阵, 通过提取不同视图之间的共性信息进行精化, 强调其共性信息并减弱噪音信息的影响. 然后, 该算法自适应地为不同视图分配权重和阶数 (order), 将它们聚合为一个共识图, 并使用图神经网络对共识图的每个节点进行图嵌入, 通过嵌入向量进行聚类并得到最终结果. 本文选取 DuaLGR 算法作为聚类算法的原因有以下几点: 首先, 该算法的图精化过程可以增强不同视图的共性信息并减弱噪音信息, 在多视图聚类任务上达到当前最佳效果; 其次, 该算法在融合多视图时动态分配每个视图的权重, 解决了最优权重在不同软件项目上不一致的问题; 最后, 该算法具有高度可扩展性, 可以容易地添加更多类型的代码关系以适应不同应用场景.

为了使用 DuaLGR 算法对代码元素图进行聚类, 首先拼接邻接矩阵 A^s和A^c 来构造共享特征矩阵 X. DuaLGR 算法设计了两个标签用于精化不同视图的邻接矩阵. 第 1 个标签是通过预训练的自编码器产生的软标签 Z_f, 它同时蕴含了来自共享特征矩阵和邻接矩阵的高层语义信息, 用于计算得到精化矩阵 $\Omega = Z_f Z_f^{\mathrm{T}}$. 第 2 个标签是从训练过程中的聚类结果中得到的伪标签, 用于评估每个视图对聚类结果的贡献, 并基于此对贡献更高的视图分配更高的阶数 od^r 以捕捉更复杂的节点关联. 最终每个视图的邻接矩阵被精化为:

$$\bar{A}^r = \alpha\left(\frac{1}{od^r}\sum_{i=1}^{od^r}(A^r)^i\right) + \Omega \tag{6}$$

其中, α 是一个超参数, 用于控制不同视图的影响. 精化后每个视图的邻接矩阵包含了高阶结构信息和全局共同信息. 为了融合不同视图的邻接矩阵, DuaLGR 利用伪标签动态计算每个视图的权重 w_s 与 w_c, 以计算得到蕴含全局信息的共识图:

$$A = w_s\bar{A}^s + w_c\bar{A}^c \tag{7}$$

共识图随后通过图卷积网络[10] 进行嵌入表示, 并使用 K-means 算法[11]基于节点的嵌入向量进行聚类. 聚类完成后我们得到节点合并后代码元素图 G' 的聚类结果. 如果 G' 中的一个节点代表原始代码元素图 G 中多个代码元素, 那么这些代码元素都被视为属于该节点所在的聚类.

3.5 循环依赖修正

第 3.4 节已经获得了一组不相交的代码元素聚类 $C = \{C_1, \ldots, C_k\}$, 每个聚类代表一个子头文件. 如果聚类 C_i 中存在依赖于 C_j 的代码元素, 那么 C_i 代表的子头文件需要包含 C_j 代表的子头文件, 形成子头文件间的依赖关系. 若子头文件间的依赖关系形成了环, 称之为产生了循环依赖. 存在循环依赖的分解方案可能会引发编译错误, 无法直接应用于头文件的分解重构.

针对上述问题, 本文提出了一种启发式循环依赖修正算法. 该算法扩展 Herrmann 等人[12] 提出的两节点循环依赖修正算法, 将长度大于 2 的循环归约为两节点的循环, 并设计增益函数来指导归约方式. 当处理两节点的循环依赖时, 设 C_i 依赖 C_j 且 C_j 依赖 C_i, 可以选择消除 C_i 到 C_j 的依赖或 C_j 到 C_i 的依赖, 每种选择又分为两种移动方

式: 移动祖先节点或后代节点, 因此共有 4 种修正方案. 为了选出最优方案, 本文设计了移动增益函数来为每种方案计算其增益. 移动增益是根据修正方案中需要移动的节点与其相关的依赖关系强度来计算的. 设 V_m 表示从 C_i 移动到 C_j 的节点集合, 则移动增益的计算公式如下:

$$gain = \sum_{\substack{v_r \in V_m \\ v_s \in C_j}} \left(A^d_{r,s} + A^d_{s,r}\right) - \sum_{\substack{v_r \in V_m \\ v_s \in C_i - V_m}} \left(A^d_{r,s} + A^d_{s,r}\right) - |V_m| \tag{8}$$

公式 (8) 计算了 V_m 与 C_j 之间所有依赖关系强度总和, 并减去了 V_m 与 C_i 之间所有依赖关系强度总和. 此外, 修正循环依赖时应该尽可能少地移动节点以减少影响, 因此公式 (8) 中还减去了修正方案所需移动节点的数量.

在修正长度大于 2 的循环依赖时, 本文方法将其归约为两节点的循环依赖, 具体算法如算法 1 所示. 对于一个长度为 $l > 2$ 的循环, 算法首先从循环中选择一个聚类 C_{i_j}, 并将其他聚类合并为一个聚类 (line 8); 然后应用上述两节点循环依赖修正算法得到一种修正方案 (line 9), 并根据方案更新循环中的聚类: 如果方案中将一些代码元素移入聚类 C_{i_j}, 这些代码元素将被移出原先所在的聚类; 反之, 如果方案将一些代码元素从聚类 C_{i_j} 中移出, 它们将被分配到与之依赖关系强度最高的聚类中. 算法遍历所有可能的归约方式并选择移动增益最大的修正方案. 该算法迭代地选择当前最长的循环依赖进行处理 (line 2), 在每轮迭代中将其长度减 1. 这种处理方式可能会引入新的更短的循环, 但不会引入长度相同或更长的循环; 若最长循环的长度为 2, 也不会引入新的循环. 因此该算法可以正确终止并消除所有循环. 本文的启发式循环依赖修正算法的最坏时间复杂度是 $O(2^n)$, 但在本文的应用场景中, 通常不会将一个复杂头文件分解为数量很多的子文件, 因此算法中的迭代次数可以保持在可控范围内, 其时间开销是可接受的.

算法 1. 循环依赖修正.

输入: 存在循环依赖的聚类结果 $\{C_1,\ldots,C_K\}$;
输出: 调整后无循环依赖的聚类结果 $\{C_1,\ldots,C_K\}$.

1. **while** $\{C_1,\ldots,C_K\}$ 存在循环依赖 **do**
2. $\{C_{i_1},\ldots,C_{i_l}\} \leftarrow FindLongestCycle(\{C_1,\ldots,C_K\})$; // 找出最长的环
3. **if** $\left|\{C_{i_1},\ldots,C_{i_l}\}\right| == 2$ **then**
4. $C_{i_1}, C_{i_2} \leftarrow FixTwoNodeCycle(C_{i_1}, C_{i_2})$;
5. **else**
6. $best\_cluster \leftarrow \varnothing$; $best\_gain \leftarrow -\infty$;
7. **foreach** $C_{i_j} \in \{C_{i_1},\ldots,C_{i_l}\}$ **do**
8. $C_{\text{else}} \leftarrow \bigcup\{C_{i_k} \mid k \neq j\}$;
9. $C'_{i_j}, C'_{\text{else}} \leftarrow FixTwoNodeCycle(C_{i_j}, C_{\text{else}})$;
10. **foreach** $C_{i_r} \in \{C_{i_k} \mid k \neq j\}$ **do**
11. $C'_{i_r} \leftarrow MoveCodeElements(C_{i_r}, C_{i_j}, C'_{i_j})$;
12. $gain \leftarrow MovingGain(\{C_{i_1},\ldots,C_{i_l}\}, \{C'_{i_1},\ldots,C'_{i_l}\})$;
13. **if** $gain > best\_gain$ **then**
14. $best\_cluster \leftarrow \{C'_{i_1},\ldots,C'_{i_l}\}$;
15. $best\_gain \leftarrow gain$;
16. $\{C_{i_1},\ldots,C_{i_l}\} \leftarrow best\_cluster$;
17. **return** $\{C_1,\ldots,C_K\}$

3.6 分解重构自动执行

消除了循环依赖的聚类结果会被作为分解方案呈现给用户, 一旦用户确认该分解方案, HeaderSplit 会自动执

行重构. 该过程主要包括两个步骤: 1) 生成每个子头文件的内容; 2) 修改所有直接或间接依赖原始头文件的代码文件中的 include 语句.

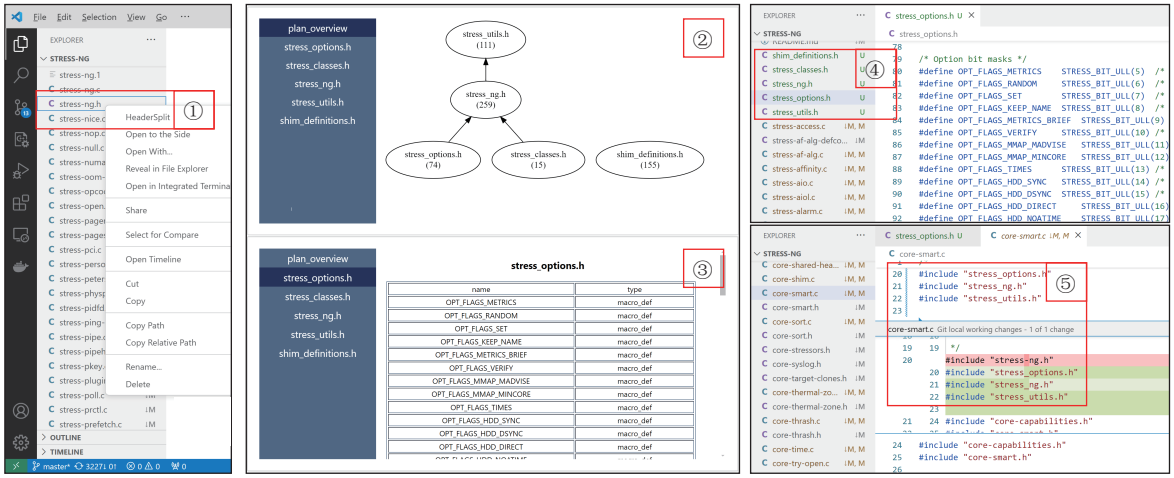
● 子头文件内容生成. 在该步骤中, HeaderSplit 已经确定每个子头文件中包含哪些代码元素, 并在建立代码元素图时记录了每个代码元素的起止位置. 在通常情况下, 该工具会直接将每个代码元素对应的代码字符串复制到相应的子文件中. 为了保持原始代码逻辑, 当代码元素位于一个或多个嵌套的条件编译块中时 (通常以 #ifdef sth 为开始标志), 代码元素必须被放置在新子文件中的相同环境中. 因此, 工具为原始头文件和每个子文件都维护了一个环境栈, 记录处理时的环境状态. 在将代码元素复制到子文件时, 工具会比较并确保原始文件和子文件的当前环境状态完全一致.

● Include 语句更新. 在这个步骤中, HeaderSplit 会为新生成的子头文件以及直接或间接包含了原始头文件的其他代码文件更新或添加 include 语句, 目标是确保重构后的项目代码文件仍可访问必要的代码声明、保持原有语义以及能够顺利进行编译. 为此, 工具解析每个代码文件的内容, 识别它们所需的子文件, 并将原来的 include 语句替换为一个或多个指向新子文件的 include 语句. 值得注意的是, 某些代码文件可能通过中间头文件间接包含了原始头文件. 如果中间头文件在更新后没有包含所需的子文件, 可能会导致编译错误. 对此, 本文会检查所有间接包含原始头文件的文件, 并添加必要的 include 语句. 这也符合 Google 公司提出的“include what you use”原则[13].

4 方法实现与实验分析

本节介绍本文方法的工具实现, 并在合成复杂头文件与真实复杂头文件上验证本文方法的有效性.

本文将 HeaderSplit 实现为一个面向 C 语言项目头文件分解的 VS Code 插件 (代码开源在 https://github.com/wangyue0502/HeaderSplit), 后端使用 Python, 前端使用 JavaScript. 具体地, 本文工具使用开源软件 tree-sitter 进行代码解析, 识别头文件中的代码元素以及依赖关系. 图 4 展示了该插件的用户界面, 共有 5 个截图: 用户右键点击待分解的头文件并选择 HeaderSplit 以启动分解重构流程 (①). 然后, 工具会返回一个分解方案, 展示每个子头文件中代码元素的数量及其包含关系的概览 (②). 其他标签页会显示每个子头文件包含哪些代码元素 (③). 确认方案后, 工具将自动执行重构过程. 在这个示例中, 头文件 stress-ng.h 被成功分解为 5 个子头文件. 其中一个子头文件 stress_option.h (④) 包含 74 个代码元素, 主要是关于选项配置的宏定义. 而在其他代码文件中, 如 core-smart.c (⑤), 原有的 include 语句被更新为 3 个新的 include 语句, 确保所有必要的定义和声明都能被访问.



(a) 插件入口 (b) 分解方案 (c) 分解结果

图 4 HeaderSplit 工具界面

4.1 实验设置

4.1.1 研究问题

本节通过实验评估本文提出的复杂头文件分解与重构方法的有效性, 旨在回答下述研究问题.

RQ1. HeaderSplit 分解复杂头文件时的准确率如何?

在该研究问题中, 本文通过实验探索现有的复杂类重构方法是否适用于分解复杂头文件, 以及本文方法在准确性方面的提升有多少.

RQ2. HeaderSplit 在优化头文件架构设计方面表现如何?

该研究问题关注本文方法在分解真实复杂头文件时带来的架构方面的提升.

RQ3. HeaderSplit 分解复杂头文件后可以减少多少重新编译?

在该研究问题中, 本文基于复杂头文件的历史修改提交估算分解文件后在降低重新编译成本方面的收益.

RQ4. HeaderSplit 的运行效率如何?

该研究问题旨在探究本文工具执行复杂头文件分解重构时的时间开销.

4.1.2 实验数据

为了评估本文方法的有效性, 我们构造了两个数据集, 分别包含人工合成的复杂头文件以及真实世界开源项目中的复杂头文件. 在构造合成复杂头文件数据集时, 我们遵循了 Bavota 等人[4] 的方法, 选取内聚度高的较小头文件合并在一起形成复杂头文件. 具体而言, 我们从 3 个开源项目中, PolarDB-for-PostgreSQL (PolarDB)、fontforge 和 FreeRDP, 分别选择 8 个内聚度高的头文件, 然后将其中的 4、6、8 个头文件合并, 得到了 9 个人工合成的复杂头文件, 并将合成前的头文件视为期望分解结果, 用于评估本文方法的准确性. 在选取头文件时, 我们选择了内聚度高于项目平均内聚度的头文件, 这样可以尽可能筛除设计质量较差的头文件, 保证数据集的质量. 为了评估本文方法在真实数据上的效果, 我们还将其应用于第 2 节识别出的 6 个典型复杂头文件上 (表 2). 这些文件是每个软件项目内经过人工确认的复杂头文件中修改提交次数最多的, 包含大量代码元素, 影响项目内数十万行代码. 同时, 它们拥有丰富的修改提交历史, 便于根据历史提交信息估算文件分解后能够减少多少重新编译成本.

4.1.3 对比方法

由于本文是关注于复杂头文件分解重构的研究工作, 因此选择了问题场景较为类似的复杂类重构方法作为对比方法, 分别如下.

1) Bavota 等人[4] 提出的基于方法间结构相似度、概念相似度与调用依赖相似度的复杂类重构算法. 该方法使用加权求和的方式结合 3 种相似度度量构建了带权图, 识别其中的连通子图从而得到复杂类分解结果.

2) Wang 等人[14] 提出的基于多维度软件复杂网络模型的自动化重构技术. 该方法提出了功能耦合关系的度量, 并使用基于模块度的聚类算法来划分得到新的类.

3) Akash 等人[15] 提出的基于图自编码器 (graph auto-encoder)[16] 的复杂类重构方法. 该方法使用图自编码器为类中的每个方法节点学习向量表示, 基于向量表示将方法聚类为不同的组以推荐作为重构类.

本文选取 Bavota 等人和 Wang 等人的方法是因为它们代表了该领域的最新进展, 性能均超过了更广泛使用的 JDeodorant[17]; 选择 Akash 等人的方法是因为该方法与本文方法都应用了基于图神经网络的聚类算法. 为了将上述复杂类分解方法应用于复杂头文件分解任务, 本文重新实现了每个方法内部的代码相似性度量模块, 将原文中“属性”和“方法”之间的关系映射为“代码元素”之间的关系, 例如将属性访问关系与方法调用关系替换为代码元素之间的定义-使用关系 (依赖关系); 后续的聚类过程严格按照原文表述操作.

4.1.4 评价指标

为了评估方法在合成复杂头文件上的准确性, 本文选取了一组衡量生成分解结果与期望分解结果接近程度的指标.

1) *MoJoFM*[18] 量化了将生成结果 A 转换为期望结果 B 所需的移动 (move) 或合并 (join) 操作的次数, 其计算公式为:

$$MoJoFM(A,B)=\left(1-\frac{mno(A,B)}{\max(mno(\forall A,B))}\right)\times 100\% \tag{9}$$

其中, $mno(A,B)$ 是将 A 转换为 B 所需的最小操作次数, $\max(mno(\forall A,B))$ 是从任何一种分解结果转换为 B 所需的最小操作次数的最大值, 用于将指标归一化. *MoJoFM* 的取值范围为 0–100%, 取值越高意味着生成结果与期望结果越接近.

2) 归一化互信息 (normalized mutual information, NMI)[19] 计算生成结果和期望结果的互信息, 并将其归一化到 0–1 的范围内, 值越接近 1 聚类一致性越高.

3) 调整兰德系数 (adjusted rand index, ARI)[20] 基于成对元素的聚类相似性计算生成结果与期望结果之间的相似性, 其取值范围在−1 到 1 之间, 取值越高表明两个聚类越相似.

4) 准确率 (accuracy)[21] 利用 Kuhn-Munkres 算法[22] 建立了生成聚类与期望聚类之间的映射, 基于此计算处于正确聚类中的元素比例.

5) *F*1 分数 (*F*1 score, 简写为 *F*1) 也是基于上述聚类映射结合精确率与召回率得出.

为了评估方法在真实复杂头文件上的性能, 本文使用以下度量指标评估分解结果的架构质量和重新编译的减少量.

1) 模块度 (*Modularity*)[23] 是一种常用的评估图划分质量的指标. 它衡量同一个聚类中节点间连接关系的密度超过随机分布的关系密度的程度. 具体计算公式为:

$$Modularity=\frac{1}{2m}\sum_{i,j}\left[A_{ij}-\frac{k_ik_j}{2m}\right]\delta(c_i,c_j) \tag{10}$$

其中, A_{ij} 表示节点 v_j 和 v_j 之间的关系强度, $k_i=\sum_j A_{ij}$ 是与节点 v_i 相连接的所有边的关系强度之和, m 是图中所有边的关系强度总和, 若 v_i 和 v_j 在同一个聚类中 $\delta(c_i,c_j)$ 取值为 1, 否则为 0. 在本文的代码元素图中, 两节点之间的边的关系强度计算为: $A_{ij}=A^d_{i,j}+A^s_{i,j}+A^c_{i,j}$. 模块度得分越高说明一个图划分的架构设计质量越好.

2) 重新编译成本. 本文基于复杂头文件的修改提交历史计算分解文件后所需的重新编译成本, 分析软件的构建依赖关系识别出每次修改提交下需要重新编译的代码文件集合, 并依此计算出需要重新编译的代码文件数和代码行数. 此外, 本文参照 McIntosh 等人[6] 的方法, 记录了每个编译单元所需的编译时间, 并基于此估算特定修改提交下所需的重新编译时间.

4.2 实验结果与分析

4.2.1 准确率

本节探讨 RQ1. HeaderSplit 分解复杂头文件时的准确率如何? 为了评估方法的准确率, 我们将 3 种对比方法与本文方法应用于分解合成复杂头文件. 由于每个合成复杂头文件存在固定的预期子文件数量, 我们在应用这些分解方法时调整了每个方法的超参数, 以确保分解结果中恰好包含对应数量的子文件.

表 3 展示了这些分解算法在分解合成复杂头文件时的表现. 本文方法 HeaderSplit 在分解合成复杂头文件时均优于对比方法. 一方面, 本文方法在所有合成复杂头文件以及所有度量指标上都取得了超过对比方法的准确率, 与表现最好的对比方法 (Wang 等人[14]的方法) 相比, 本文方法的 *MoJoFM* 值平均提高了 11.5%, 更符合开发者预期的分解结果. 另一方面, 本文方法具有更好的跨项目稳定性, 在来自不同软件项目的合成复杂头文件上均表现良好. 相比之下, Bavota 等人[4] 和 Akarsh 等人[15] 的方法在不同软件项目上的表现差异较大, 它们在 FreeRDP 项目中取得了 *MoJoFM* 值超过 80% 的高准确率, 但在 fontforge 项目中的准确率较低, *MoJoFM* 值只有 50% 左右; 而 Wang 等人[14]的方法在 PolarDB 和 fontforge 项目中优于其他两种对比方法, 但在 FreeRDP 项目中表现较差, 稳定性有待提升. Wang 等人[14]的方法比其他对比方法表现更好的原因在于它考虑了功能耦合关系, 这与本文方法中的共同使用关系较为相似. 然而, 该方法依赖一组固定的权重来聚合多种代码关系, 而这些代码关系在不同软件项目中的重要程度是有区别的, 这也解释了该方法在不同软件项目上表现不够稳定的原因. 本文方法应用的多视图聚类算法为不同软件项目动态分配代码关系的权重, 解决了上述问题, 因此具备更好的跨项目稳定性.

表 3 不同分解算法在合成复杂头文件的准确率 (%)

方法	PolarDB (#subfiles=4, #code elements=138)					PolarDB (#subfiles=6, #code elements=276)					PolarDB (#subfiles=8, #code elements=362)				
	MoJoFM	NMI	ARI	ACC	*F*1	*MoJoFM*	NMI	ARI	ACC	*F*1	*MoJoFM*	NMI	ARI	ACC	*F*1
Bavota等人[4]	69.9	63.0	50.5	64.2	61.0	56.1	44.2	20.9	56.7	36.2	42.9	51.7	32.1	42.9	15.8
Wang等人[14]	96.2	90.6	90.9	96.4	96.2	87.4	90.2	89.7	87.3	76.0	87.9	87.6	87.1	87.8	79.4
Akash等人[15]	75.9	59.2	51.4	76.6	75.2	75.8	63.2	73.1	76.0	66.9	68.8	61.3	66.8	66.8	58.0
HeaderSplit	**100**	**100**	**100**	**100**	**100**	**89.2**	**89.3**	**92.1**	**89.5**	**82.6**	**90.1**	**91.4**	**90.8**	**90.0**	81.7
w/o 节点合并	96.2	92.0	92.7	97.1	96.9	88.1	88.1	90.0	87.3	77.9	89.5	88.2	86.6	89.8	**85.4**
w/o 多视图	93.9	87.0	87.0	94.9	94.8	87.7	87.3	78.2	81.8	75.4	88.1	88.5	78.7	83.9	79.8

方法	fontforge (#subfiles=4, #code elements=145)					fontforge (#subfiles=6, #code elements=292)					fontforge (#subfiles=8, #code elements=342)				
	MoJoFM	NMI	ARI	ACC	*F*1	*MoJoFM*	NMI	ARI	ACC	*F*1	*MoJoFM*	NMI	ARI	ACC	*F*1
Bavota等人[4]	41.4	13.1	0.4	38.2	31.6	51.6	25.9	26.4	45.7	30.2	45.9	23.5	15.1	40.8	26.0
Wang等人[14]	97.9	95.2	96.2	98.6	98.5	71.9	65.2	47.3	71.5	66.0	65.0	61.9	41.6	64.5	56.2
Akash等人[15]	53.6	24.0	14.8	52.1	49.9	53.7	28.7	20.6	49.5	46.2	49.5	29.1	18.1	45.2	41.5
HeaderSplit	**100**	**100**	**100**	**100**	**100**	**89.1**	**82.8**	**75.7**	**89.6**	**89.1**	**88.0**	**82.8**	**72.9**	**88.6**	**88.8**
w/o 节点合并	99.3	97.8	98.8	99.3	98.9	78.9	68.4	50.4	73.9	72.7	80.8	72.8	54.3	78.0	79.8
w/o 多视图	100	100	100	100	100	75.0	67.7	46.4	68.0	62.5	87.6	81.3	71.1	88.3	88.6

方法	FreeRDP (#subfiles=4, #code elements=217)					FreeRDP (#subfiles=6, #code elements=397)					FreeRDP (#subfiles=8, #code elements=534)				
	MoJoFM	NMI	ARI	ACC	*F*1	*MoJoFM*	NMI	ARI	ACC	*F*1	*MoJoFM*	NMI	ARI	ACC	*F*1
Bavota等人[4]	87.2	71.5	57.2	65.6	58.9	91.2	85.0	85.0	78.9	68.2	91.8	85.2	80.1	73.2	61.3
Wang等人[14]	90.5	79.4	61.3	68.4	63.6	80.2	76.6	66.5	71.8	42.6	81.0	79.5	67.7	72.1	41.6
Akash等人[15]	89.1	74.3	60.2	67.0	61.2	88.4	78.9	80.7	76.1	66.6	80.3	68.3	67.6	65.1	55.7
HeaderSplit	**99.1**	**96.3**	**97.8**	**99.1**	**98.8**	**99.2**	**98.3**	**98.9**	**99.5**	**99.3**	**99.0**	**98.5**	**99.1**	**99.4**	**98.9**
w/o 节点合并	98.6	95.4	98.0	98.6	97.3	98.7	96.8	98.5	99.0	98.0	98.9	97.5	98.6	99.1	98.3
w/o 多视图	67.7	50.9	27.7	59.6	56.1	81.1	69.1	53.4	81.7	84.3	68.1	55.5	27.4	59.6	56.1

表 3 中的消融实验部分 (w/o 多视图) 也证明了多视图聚类模块的有效性. 在使用 K-means 算法替换这一部分时, 本文方法几乎在全部合成复杂头文件上都出现了准确率下降的现象, 在 PolarDB 和 fontforge 项目上的性能下降相对较轻微, 而在 FreeRDP 项目中的下降则相当显著, 约为 30%. 这种差异源于不同项目中各种代码关系的重要性不同. 在这种情况下, 使用固定权重来组合多种代码关系是不合适的, 这突显了多视图聚类算法中动态分配权重机制的优越性. 与此同时, 删除节点合并步骤也会导致准确率下降 (w/o 节点合并), 展现了该步骤对方法整体表现的贡献. 对于大多数合成头文件而言, 节点合并的影响较小, 准确率降低幅度低于 5%. 然而, 对于由 fontforge 项目中的 6 个和 8 个文件组成的两个合成头文件, 节点合并的影响较为显著, 原因在于这些文件中的依赖关系占比高于其他文件, 使得节点合并步骤在这些文件中的作用尤为关键.

综上所述, 与现有方法相比, HeaderSplit 在分解合成复杂头文件时的准确率提升了 11.5%, 并且具有更强的跨软件项目稳定性; 方法中的节点合并与多视图聚类两个步骤都对方法性能具有显著贡献.

4.2.2 架构设计提升

本节探讨 RQ2. HeaderSplit 在优化头文件架构设计方面表现如何? 本文还将 3 种对比方法与本文方法应用于分解真实世界的复杂头文件以评估其效果. 这些实验数据不存在预期子文件的数量, 因此在实验过程中, 本文对每个实验数据进行 7 次子文件数量不同的分解, 取值为 4–10. 这样设置的原因是, 开发人员在实际重构中通常不会将一个头文件分解为超过 10 个子文件. 本节将从循环依赖与模块度两个角度讨论本文方法在分解复杂头文件时的架构优势.

我们计算了每个分解方法在不同实验数据以及实验设置上的分解结果模块度, 并将其绘制成折线图, 如图 5 所示. 图 5 中用黑色×符号标记了存在循环依赖的分解结果, 这些结果会引发编译错误因此无法实际应用. 在实验

中, 每种分解方法被应用于 6 个真实复杂头文件上, 每个文件尝试了 7 个不同的子文件个数, 共计 42 个分解结果. 在这些结果中, Wang 等人[14]和 Akash 等人[15]的方法分别有 15 和 28 个分解结果存在循环依赖. 相比之下, Bavota 等人[4]的方法因其基于连通子图的聚类算法未生成存在循环依赖的结果, 但这些结果的模块度往往比较低, 架构设计不够合理. 而本文方法的循环依赖修正模块确保分解后的子头文件之间没有循环依赖, 多视图聚类模块则保证了较高的模块度水平, 从而使分解结果的架构设计更加简洁和清晰.

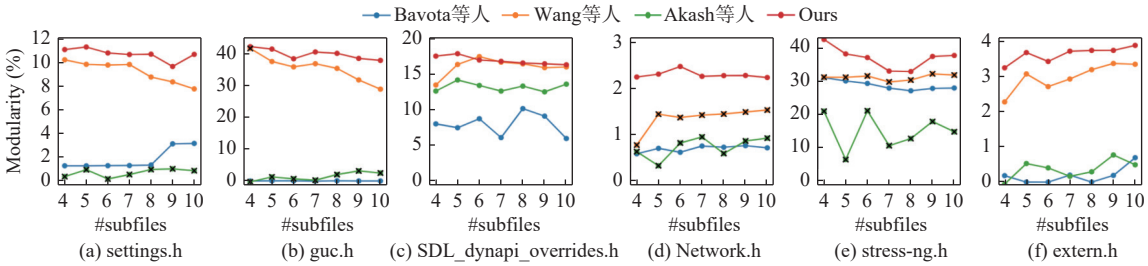
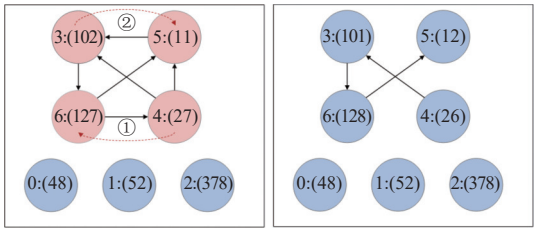


图 5 不同分解算法在真实复杂头文件上的模块度

在模块度方面, 本文方法几乎在所有实验设置上都取得了最高的模块度值, 相较于最佳对比方法提升了 9.1%. 这表明本文方法分解生成的子头文件更符合高内聚、低耦合的软件设计原则, 具有更好的架构设计. 对于文件 SDL_dynapi_overrides.h, Wang 等人[14]的方法在模块度方面与本文方法相当. 这是因为在该文件中, 只有一种占主导地位的代码关系——文件内没有依赖关系, 共同使用关系的数量比文本相似关系少一个数量级. 在这种情况下, 本文方法的多视图聚类模块无法发挥其优势. 此外, 图 5 显示出模块度值在不同文件之间存在显著差异, 例如 guc.h 达到了 40% 的模块度值, 而 Network.h 则难以超过 3%. 这种差异可能源于 Network.h 中代码元素关系的复杂性. 此类复杂头文件可能超出了自动分解重构方法的能力, 需要专家介入以取得更好的结果.

图 6 展示了一个示例以说明循环依赖修正算法的具体运行过程. 图 6(a) 是本文方法在多视图聚类阶段为复杂头文件 settings.h 生成的分解结果, 其中存在 3 个环, 分别是: 3-6-4-5、3-6-4 和 3-6-5. 循环依赖修正算法优先处理最长的环 3-6-4-5. 启发式搜索过程中得出的移动增益最高的修正方案是将代码元素 rdpSettings 从子文件 4 移动到子文件 6. 这个方案只移动一个代码元素, 获得了 61 的移动增益, 消除了从 4-5 和从 6-4 的依赖关系. 经过这次移动后的中间结果只包含一个环: 3-6-5. 针对这个环, 算法搜索出的最佳修正方案是将代码元素 ALIGN64 从子文件 3 移动到子文件 5. 这次移动后所有循环依赖都已经被消除, 最终的分解子文件间依赖关系如图 6(b) 所示.



(a) 聚类阶段后的分解结果 (b) 最终分解结果

图 6 循环依赖修正算法在 settings.h (#subfiles=7) 上的运行过程

综上所述, 与现有方法相比, HeaderSplit 分解复杂头文件得到的子文件具有更好的架构设计. 一方面, 它在模块度方面比最佳对比方法提高了 9.1%; 另一方面, 它可以确保子文件之间不存在循环依赖, 依赖关系更简洁.

4.2.3 编译效率提升

本节探讨 RQ3. HeaderSplit 分解复杂头文件后可以减少多少重新编译? 为了估算重编译成本, 本文为数据集内的每个复杂头文件收集其修改提交 (commit) 历史, 并提取出每次修改提交中修改的代码元素. 对一个复杂头文件的任何修改都会引起所有依赖它的代码文件重新编译, 但将该头文件分解为子文件后, 一次修改可能仅影响部

分子文件, 因此仅依赖未修改子文件的代码文件就不再需要重新编译了. 通过分析软件的构建依赖关系, 我们能够识别出每次修改提交需要重新编译的代码文件集合, 并依此计算出需要重新编译的代码文件数和代码行数. 然而, 获取特定的修改提交实际所需的编译时间较为困难, 重新执行这些修改并进行实际编译并非易事, 尤其是涉及软件系统的中间版本. 因此, 本文参照 McIntosh 等人[6]的方法, 记录了每个编译单元所需的编译时间, 并基于此估算特定修改提交所需的重编译时间.

表 4 展示了每个复杂头文件在分解前平均每次修改提交所需的重编译成本, 以及使用每种分解算法分解后平均所需的重编译成本. 对于每种分解算法, 表 4 中只保留了减少重编译最多的子文件数量设置. 总的来说, 本文方法 HeaderSplit 生成的分解结果减少重编译成本的幅度最为显著, 减少幅度从 15%–60% 不等. 本文方法只在 guc.h 上未能超过 Wang 等人的对比方法, 这是因为该文件的修改提交数据分布不均衡, 其总计 61 次修改提交中有 58 次都未修改 Wang 等人分解结果中最大规模的子文件. 然而, 尽管 Wang 等人的方法在部分头文件上取得了更好的效果, 但由于其不能解决循环依赖的问题, 在 Network.h 和 stress-ng.h 上未能生成实际可行的分解结果. Bavota 等人[4]的方法成功为全部文件生成可行的分解结果, 但这些结果在减少重编译成本方面表现较差. 只有本文方法可以在稳定生成可行分解结果的同时显著减少重编译成本.

表 4　分解复杂头文件后的平均重编译减少量与总节省时间

方法	settings.h					guc.h				
	子文件数	重编译代码文件数	重编译代码行数	重编译时间(s)	节省总时间(min)	子文件数	重编译文件数	重编译代码行数	重编译时间(s)	节省总时间(min)
分解前	—	368	279 617	252.4	—	—	388	741 265	260.3	—
Bavota等人[4]	10	250.3 (↓31.9%)	197 735 (↓29.3%)	177.5 (↓29.7%)	385.5	10	337.7 (↓13.0%)	648 162 (↓12.6%)	227.2 (↓12.7%)	15.8
Wang等人[14]	8	225.5 (↓38.7%)	180 629 (↓35.4%)	160.6 (↓36.3%)	472.5	10	**214.3** (↓44.8%)	**432 930** (↓41.6%)	**147.9** (↓43.2%)	**114.3**
Akash等人[15]	—	—	—	—	—	—	—	—	—	—
HeaderSplit	9	**163.0** (↓55.7%)	**145 612** (↓47.9%)	**130.7** (↓48.2%)	**626.7**	7	281.5 (↓27.4%)	562 000 (↓24.2%)	190.6 (↓26.8%)	70.8

方法	SDL_dynapi_overrides.h					Network.h				
	子文件数	重编译代码文件数	重编译代码行数	重编译时间(s)	节省总时间(min)	子文件数	重编译文件数	重编译代码行数	重编译时间(s)	节省总时间(min)
分解前	—	127	100 654	215.4	—	—	77	268 084	113.4	—
Bavota等人[4]	9	75.9 (↓40.2%)	63 911 (↓36.5%)	133.5 (↓38.0%)	152.9	10	64.8 (↓15.9%)	242 437 (↓9.6%)	101.6 (↓10.4%)	9.8
Wang等人[14]	10	50.4 (↓60.3%)	50 811 (↓49.5%)	96.0 (↓55.4%)	223.0	—	—	—	—	—
Akash等人[15]	9	69.5 (↓45.3%)	63 171 (↓37.2%)	128.4 (↓40.4%)	162.4	—	—	—	—	—
HeaderSplit	10	**42.9** (↓66.2%)	**45 594** (↓54.7%)	**86.2** (↓60.0%)	**241.1**	8	**56.1** (↓27.1%)	**210 925** (↓21.3%)	**88.9** (↓21.6%)	**20.4**

方法	stress-ng.h					extern.h				
	子文件数	重编译代码文件数	重编译代码行数	重编译时间(s)	节省总时间(min)	子文件数	重编译文件数	重编译代码行数	重编译时间(s)	节省总时间(min)
分解前	—	314	156 573	284.0	—	—	324	171 335	1 047.0	—
Bavota等人[4]	10	268.6 (↓14.5%)	139 526 (↓10.9%)	234.0 (↓17.6%)	291.2	10	305.4 (↓5.7%)	162 749 (↓5.0%)	987.5 (↓5.7%)	1 350.5
Wang等人[14]	—	—	—	—	—	7	283.0 (↓12.6%)	152 740 (↓10.8%)	916.1 (↓12.5%)	2 974.3
Akash等人[15]	—	—	—	—	—	9	291.4 (↓10.1%)	159 601 (↓6.8%)	943.6 (↓9.9%)	2 348.7
HeaderSplit	6	**256.1** (↓18.4%)	**135 175** (↓13.7%)	**222.5** (↓21.6%)	**358.6**	9	**273.2** (↓15.7%)	**144 880** (↓15.4%)	**886.2** (↓15.4%)	**3 653.3**

分解复杂头文件带来的重编译成本减少量在不同文件之间也存在差异. 例如, 分解 extern.h 平均只减少了 15% 的重编译成本, 而分解 SDL_dynapi_overrides.h 的减少幅度达到了 60%. 尽管对于 extern.h, 重编译减少的比例不是很高, 但由于该文件已被修改超过 1 000 次, 分解该文件在其演化过程中可以总计节省 60 h 的重编译时间. 实际上, 对于构建成本较大、修改频繁的软件项目, 分解复杂头文件能够带来更显著的重编译效率提升.

综上所述, 本文方法分解复杂头文件可以显著降低其在软件项目演化过程中的重新编译成本, 幅度在 15%–60% 之间, 至多可以为一个复杂头文件在其演化历史中节省 60 h 的编译时间.

4.2.4 工具运行效率

本节探讨 RQ4. HeaderSplit 的运行效率如何? 为了评估工具的运行效率, 本文将 HeaderSplit 在每个真实头文件上运行 3 次, 并在表 5 中报告了 HeaderSplit 每个阶段的平均执行时间以及总用时. 其中, 代码解析阶段负责从目标头文件中提取代码元素, 并解析项目中的所有其他代码文件以识别构建依赖关系, 这些信息既用于构建代码元素图, 也用于重构执行; 分解方案生成阶段包括代码元素图的构建、节点合并、多视图聚类以及循环依赖修正; 分解重构执行阶段则是根据分解方案生成新的子文件内容并更新其他代码文件中的 include 语句. 实验在 1 台 Ubuntu 服务器上进行, 配备 2 核 Intel CPU、32 GB 内存和 1 块 RTX 4090 GPU. 总体而言, HeaderSplit 成功分解了所有 6 个真实复杂头文件, 分解结果均能正常编译通过, 工具运行时间在 81.5–193.0 s 之间. 考虑这些头文件及其所属项目的复杂程度和规模, 该运行时间在实际应用中是完全可以接受的. 在代码解析阶段, HeaderSplit 解析项目中的所有代码文件以提取构建依赖关系, 因此处理时间取决于项目的规模. 该工具的大部分时间花在分解方案生成阶段, 特别是多视图聚类算法. 这一阶段的时间波动较大, 与代码元素图的复杂程度有关: 当图中的边较少且聚类比较明显时, 聚类算法的收敛速度更快. 相比之下, 执行重构所需的时间始终保持在 10 s 以内.

表 5 HeaderSplit 工具运行时间

待分解复杂头文件	代码元素数量	所属软件项目规模 (KLOC)	代码解析用时 (s)	分解方案生成用时 (s)	分解重构执行用时 (s)	总用时 (s)
settings.h	745	507.3	15.8	101.9	9.7	127.4
guc.h	575	1 674.0	36.3	73.1	7.1	116.5
SDL_dynapi_overrides.h	769	349.9	12.1	117.5	9.9	139.6
Network.h	680	360.8	10.6	172.8	9.6	193.0
stress-ng.h	610	189.8	2.6	72.9	6.0	81.5
extern.h	1 274	221.0	6.1	105.1	3.5	114.6

综上所述, HeaderSplit 可以高效地分解重构复杂头文件, 在包含数百万行代码的大型项目中, 能够在 81–193 s 内分解重构数千行代码的复杂头文件.

4.3 有效性分析

本文工具仍然存在一些局限: 1) 本文方法在理论上可以为 C/C++语言的软件项目分解重构复杂头文件, 但 HeaderSplit 目前仅支持 C 语言项目, 未来将扩展支持 C++语言项目. 2) HeaderSplit 依赖 tree-sitter 进行代码解析, 该工具对 C 语言的支持基于 C99 标准, 因此, 对于包含非 C99 语法的代码文件, HeaderSplit 可能会因为代码解析错误无法顺利完成重构过程. 未来将探索更加全面的代码解析方式以支持更多场景.

本文实验结果可能受到一些有效性威胁. 内部有效性威胁包括: 1) 实验使用的合成复杂头文件数据来自合并若干小型头文件, 并将原始的小型头文件作为基准计算分解结果的准确率. 若原始头文件存在设计问题则基准可能包含错误. 为了降低这一问题的影响, 本文选取了内聚度较高的原始头文件用于数据合成. 2) 本文方法与对比方法中的聚类算法均存在一定程度的随机性, 聚类算法的初始划分或合并聚类过程的随机性均有可能影响最终聚类结果, 因此方法对同样的输入产生的结果可能存在差异. 为了减轻这些潜在的偏差, 本文在每组数据上均对本文方法与对比方法运行 3 次并报告了最佳结果. 3) 本文选用的评价指标的有效性也会影响本文结果的有效性. RQ3 的重编译成本是基于历史修改提交数据计算的, 体现了分解结果在历史数据上降低重编译成本的表现, 无法推断出在未来的修改提交中是否会有同样的表现.

本文实验验证的主要外部有效性威胁来自所选取的实验数据是否具备泛化性. 本文分别在合成复杂头文件与真实复杂头文件对 HeaderSplit 进行评估, 并与几种不同类型的复杂类重构方法进行对比. 选取的实验数据来自多个不同领域的真实软件项目, 一定程度上保证了实验数据的客观性以及真实性. 但由于衡量重编译成本较为复杂, 实验数据集的规模仍然比较有限, 在将本文结果推广到其他数据或算法时应当保持谨慎.

5 相关工作

本文提出了基于多视图聚类的复杂头文件分解重构方法, 关注通过分解头文件来减少重编译成本, 在方法设计中借鉴了复杂类重构技术并应用了多视图聚类算法, 因此本节从以下 3 个方面介绍相关工作: 头文件构建优化技术、复杂类重构技术和多视图聚类技术.

5.1 头文件构建优化技术

软件开发人员通常依靠快速构建系统来增量地编译源代码更改, 并为测试和部署生成修改后的可交付成果, 而头文件往往会导致缓慢的重新编译过程. McIntosh 等人[6]提出了头文件热点 (header file hotspot) 的概念, 代指频繁更改并会触发长时间编译的头文件. 他们提出了通过分析构建依赖图和软件系统的更改历史来识别头文件热点的方法. 并通过对 4 个软件系统的案例研究, 展示了改进该方法识别的头文件热点将对系统未来的总构建成本带来很大的优化. 还有许多研究工作致力于减少头文件修改引起的编译时间. 其中一些工作对头文件本身进行优化. Yu 等人[24]提出了一个图形算法和编程工具来发现和消除文件之间的错误依赖关系, 通过实验证明该工具得到的预处理代码更加紧凑, 从而有助于加快编译构建过程. Spinellis 等人[25]提出了删除头文件中非必要的 include 语句的方法, 通过定义 4 个连续细化的标识符等价类, 准确得出标识符之间的依赖关系并且安全地删除不必要被包含的文件. Reisch 等人[26]提出了一个优化模型, 最小化包含需要因此编译但不使用的代码行数. Google 公司提出了 include what you use 工具[27]用于删除多余的 include 语句, 并在可能的情况下用前向声明替换 include 语句. 此外, 还有一些研究工作通过优化编译过程来缩短头文件的编译时间, 例如通过缓存编译的中间结果来加快下一次编译[28]、进行预编译[29]以及检测冗余的编译[30]. 这些研究工作突显了头文件优化的重要性, 本文工作则关注复杂头文件并提出通过分解重构降低构建成本.

5.2 复杂类重构技术

复杂类的重构一直是软件工程领域的重要研究方向. Fowler 等人[2]首次提出复杂类 (god class) 概念, 并建议通过提取类 (extract class) 的操作来进行重构, 即将内聚度较高且功能相关的数据和方法从复杂类中提取为独立的类. 这一过程虽具有显著的改善作用, 但手工分解复杂类往往耗时且易出错. 因此, 许多研究者提出了自动化或半自动化[31]的方法与工具, 以辅助开发者高效完成这一任务.

这些方法通常遵循一个类似的步骤: 首先定义并计算类内实体之间的相似度, 然后根据相似度对类内实体进行划分或聚类. de Lucia 等人[3]提出了一种自动化拆分复杂类的方法, 他们利用方法间的结构相似性和概念相似性构建带权图, 并通过最大流-最小割算法进行划分. Fokaefs 等人[32]则引入 Jaccard 距离来度量实体间的相似性, 结合层次聚类算法给出重构建议. Bavota 等人[4]进一步改进了上述方法, 新增调用依赖相似度作为衡量标准, 提出了一种综合性的图划分方法. 此外, 他们还尝试通过主成分分析确定相似度权重, 并验证了权重设置对不同项目的影响. Wang 等人[14]基于多维度软件复杂网络模型提出了功能耦合权重, 通过模块度聚类算法进一步优化复杂类的分解. 其他研究也探讨了新的相似度度量方式, 例如基于潜在狄利克雷分布[33]和注释文档相似性[34], 以捕捉更丰富的语义关系. 近期 Chen 等人[5]提出的 ClassSplitter 方法被认为是最先进的技术之一. 该方法提出了位置相似度, 通过自定义的分步聚类技术, 能够更精准地分解复杂类, 得到内聚性更高、规模更小的新类. 本文借鉴了上述工作提出的概念相似度与调用依赖相似度, 并加入反映构建依赖的共同使用关系, 有效地分解复杂头文件.

5.3 多视图聚类技术

图聚类是网络分析中的一个基础问题, 传统算法通常依赖图的数学和结构特性, 例如谱聚类[35]、基于模块度

的聚类 (如 Louvain 方法[36])、层次聚类[37]等. 然而, 这些算法往往缺乏可扩展性, 难以有效处理大型复杂图. 基于深度学习的算法对传统算法进行了优化, 通过图神经网络学习图节点的低维表示进行聚类, 典型的工作包括 GraphSAGE[38]、图卷积网络[10]和变分图自动编码器[16]等. 然而这些方法面对真实的高异质性图数据时仍然表现不佳.

多视图聚类技术通过结合来自不同来源或特征子集的信息, 利用不同视图之间的一致性和互补性, 相较于单视图聚类展现出更优的效果和泛化能力. 多视图聚类方法涵盖了多个方向, 包括多核聚类[39,40]、子空间聚类[41,42]、基于非负矩阵分解的多视图聚类[43]、基于集成的多视图聚类[44]等. 最新的代表性工作, 如 O2MAC[45]、MvAGC[46]、MCGC[47]和 DuaLGR[9], 已取得了卓越的表现. 本文算法中采用了 DuaLGR 聚类算法, 原因在于其能够有效处理非同质边, 通过动态权重分配机制解决了项目间最优权重不一致的问题, 并且具有很高的可扩展性, 可以灵活加入多种类型的代码关系信息.

6 总结与展望

本文提出了复杂头文件的问题与一种面向复杂头文件的自动化分解与重构方法——HeaderSplit. 该方法首先构造代码元素图并通过多视图聚类算法和循环依赖修正算法生成有效的头文件分解方案, 然后根据方案自动执行重构, 生成新的子文件内容并更新软件项目内所有直接或间接包含原头文件的代码语句. 本文在合成复杂头文件与真实复杂头文件上对本文方法进行评估, 其结果表明: HeaderSplit 分解复杂头文件具有更高的准确率与更强的跨软件项目稳定性; 分解生成的子文件模块度更高且无循环依赖, 架构设计更良好; 基于历史修改提交计算可以降低 15%–60% 的重编译成本; 基于本文方法实现的工具可以高效运行并实施自动化重构. 在未来工作中, 我们将探索如何推荐最佳的文件分解数量以及如何利用大语言模型生成更恰当的子文件名, 进一步提升自动化软件重构工具的智能化程度.

References

[1] Mens T, Tourwe T. A survey of software refactoring. IEEE Trans. on Software Engineering, 2004, 30(2): 126–139. [doi: 10.1109/TSE.2004.1265817]

[2] Fowler M. Refactoring: Improving the Design of Existing Code. 2nd ed., Addison-Wesley Professional, 2018.

[3] de Lucia A, Oliveto R, Vorraro L. Using structural and semantic metrics to improve class cohesion. In: Proc. of the 2008 IEEE Int'l Conf. on Software Maintenance. Beijing: IEEE, 2008. 27–36. [doi: 10.1109/ICSM.2008.4658051]

[4] Bavota G, De Lucia A, Marcus A, Oliveto R. Automating extract class refactoring: An improved method and its evaluation. Empirical Software Engineering, 2014, 19(6): 1617–1664. [doi: 10.1007/s10664-013-9256-x]

[5] Chen TY, Jiang YJ, Fan F, Liu B, Liu H. A position-aware approach to decomposing god classes. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Automated Software Engineering. Sacramento: ACM, 2024. 129–140. [doi: 10.1145/3691620.3694992]

[6] McIntosh S, Adams B, Nagappan M, Hassan AE. Identifying and understanding header file hotspots in C/C++ build processes. Automated Software Engineering, 2016, 23(4): 619–647. [doi: 10.1007/s10515-015-0183-5]

[7] Taibi D, Lenarduzzi V. On the definition of microservice bad smells. IEEE Software, 2018, 35(3): 56–62. [doi: 10.1109/MS.2018.2141031]

[8] Wang Y, Chang WH, Zou YZ, Xie B. An exploratory study on god header files in open-source C projects. In: Proc. of the 15th Asia-Pacific Symp. on Internetware. Macao: ACM, 2024. 477–486. [doi: 10.1145/3671016.3671391]

[9] Ling YW, Chen JP, Ren YZ, Pu XR, Xu J, Zhu XF, He LF. Dual label-guided graph refinement for multi-view graph clustering. In: Proc. of the 37th AAAI Conf. on Artificial Intelligence. Washington: AAAI press, 2023. 8791–8798. [doi: 10.1609/aaai.v37i7.26057]

[10] Kipf TN, Welling M. Semi-supervised classification with graph convolutional networks. In: Proc. of the 5th Int'l Conf. on Learning Representations. Toulon: OpenReview.net, 2017.

[11] Lloyd S. Least squares quantization in PCM. IEEE Trans. on Information Theory, 1982, 28(2): 129–137. [doi: 10.1109/TIT.1982.1056489]

[12] Herrmann J, Özkaya MY, Uçar B, Kaya K, Çatalyürek UV. Multilevel algorithms for acyclic partitioning of directed acyclic graphs. SIAM Journal on Scientific Computing, 2019, 41(4): A2117–A2145. [doi: 10.1137/18M1176865]

[13] Google. Google C++ style guide. 2025. https://google.github.io/styleguide/cppguide.html#Include_What_You_Use

[14] Wang Y, Yu H, Zhu ZL, Zhang W, Zhao YL. Automatic software refactoring via weighted clustering in method-level networks. IEEE Trans. on Software Engineering, 2018, 44(3): 202–236. [doi: 10.1109/TSE.2017.2679752]

[15] Akash PS, Chang KCC. Exploring variational graph auto-encoders for extract class refactoring recommendation. arXiv:2203.08787, 2022.

[16] Kipf TN, Welling M. Variational graph auto-encoders. arXiv:1611.07308, 2016.

[17] Fokaefs M, Tsantalis N, Stroulia E, Chatzigeorgiou A. JDeodorant: Identification and application of extract class refactorings. In: Proc. of the 33rd Int'l Conf. on Software Engineering. Honolulu: IEEE, 2011. 1037–1039. [doi: 10.1145/1985793.1985989]

[18] Wen ZH, Tzerpos V. An effectiveness measure for software clustering algorithms. In: Proc. of the 12th IEEE Int'l Workshop on Program Comprehension. Bari: IEEE, 2004. 194–203. [doi: 10.1109/WPC.2004.1311061]

[19] McDaid AF, Greene D, Hurley N. Normalized Mutual Information to evaluate overlapping community finding algorithms. arXiv:1110.2515, 2011.

[20] Yeung KY, Ruzzo WL. Principal component analysis for clustering gene expression data. Bioinformatics, 2001, 17(9): 763–774. [doi: 10.1093/bioinformatics/17.9.763]

[21] Lutov A, Khayati M, Cudré-Mauroux P. Accuracy evaluation of overlapping and multi-resolution clustering algorithms on large datasets. In: Proc. of the 2019 IEEE Int'l Conf. on Big Data and Smart Computing (BigComp). Kyoto: IEEE, 2019. 1–8. [doi: 10.1109/BIGCOMP.2019.8679398]

[22] Lovász L, Plummer MD. Matching Theory. Providence: AMS Chelsea Publishing, 2009.

[23] Newman MEJ, Girvan M. Finding and evaluating community structure in networks. Physical Review E, 2004, 69(2): 026113. [doi: 10.1103/PhysRevE.69.026113]

[24] Yu YJ, Dayani-Fard H, Mylopoulos J. Removing false code dependencies to speedup software build processes. In: Proc. of the 2003 Conf. of the Centre for Advanced Studies on Collaborative Research. Toronto: IBM Press, 2003. 343–352. [doi: 10.5555/961322.961375]

[25] Spinellis D. Optimizing header file include directives. Journal of Software Maintenance and Evolution: Research and Practice, 2009, 21(4): 233–251. [doi: 10.1002/smr.369]

[26] Reisch J, Grossmann P. Automatic refactoring and compile time optimization of cpp projects by directly including header files. In: Proc. of the 8th Int'l Conf. on Computer Technology Applications. Vienna: ACM, 2022. 71–76. [doi: 10.1145/3543712.3543724]

[27] A tool for use with clang to analyze #includes in C and C++ source files. 2025. https://github.com/include-what-you-use/include-what-you-use

[28] Koehler B, Horspool RN. A caching compiler for C. In: Proc. of the 1995 IEEE Pacific Rim Conf. on Communications, Computers, and Signal Processing. Victoria: IEEE, 1995. 141–144. [doi: 10.1109/PACRIM.1995.519428]

[29] Yu Y, Dayani-Fard H, Mylopoulos J, Andritsos P. Reducing build time through precompilations for evolving large software. In: Proc. of the 21st IEEE Int'l Conf. on Software Maintenance. Budapest: IEEE, 2005. 59–68. [doi: 10.1109/ICSM.2005.73]

[30] Dietrich C, Rothberg V, Füracker L, Ziegler A, Lohmann D. cHash: Detection of redundant compilations via AST hashing. In: Proc. of the 2017 USENIX Annual Technical Conf. Santa Clara: USENIX, 2017. 527–538.

[31] Anquetil N, Etien A, Andreo G, Ducasse S. Decomposing god classes at siemens. In: Proc. of the 2019 IEEE Int'l Conf. on Software Maintenance and Evolution. Cleveland: IEEE, 2019. 169–180. [doi: 10.1109/ICSME.2019.00027]

[32] Fokaefs M, Tsantalis N, Chatzigeorgiou A, Sander J. Decomposing object-oriented class modules using an agglomerative clustering technique. In: Proc. of the 2009 IEEE Int'l Conf. on Software Maintenance. Edmonton: IEEE, 2009. 93–101. [doi: 10.1109/ICSM.2009.5306332]

[33] Akash P, Sadiq A, Kabir A. An approach of extracting god class exploiting both structural and semantic similarity. In: Proc. of the 14th Int'l Conf. on Evaluation of Novel Approaches to Software Engineering. Heraklion: Science and Technology Publications, 2019. 427–433. [doi: 10.5220/0007743804270433]

[34] Jeba T, Mahmud T, Akash PS, Nahar N. God class refactoring recommendation and extraction using context based grouping. Int'l Journal of Information Technology and Computer Science, 2020, 12(5): 14–37. [doi: 10.5815/ijitcs.2020.05.02]

[35] Ng AY, Jordan MI, Weiss Y. On spectral clustering: Analysis and an algorithm. In: Proc. of the 15th Int'l Conf. on Neural Information Processing Systems: Natural and Synthetic. Vancouver: MIT Press, 2001. 849–856. [doi: 10.5555/2980539.2980649]

[36] Blondel VD, Guillaume JL, Lambiotte R, Lefebvre E. Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment, 2008, 2008: P10008. [doi: 10.1088/1742-5468/2008/10/P10008]

[37] Ward Jr JH. Hierarchical grouping to optimize an objective function. Journal of the American Statistical Association, 1963, 58(301): 236–244. [doi: 10.1080/01621459.1963.10500845]

[38] Hamilton WL, Ying R, Leskovec J. Inductive representation learning on large graphs. In: Proc. of the 31st Int'l Conf. on Neural Information Processing Systems. Long Beach: Curran Associates Inc., 2017. 1025–1035. [doi: 10.5555/3294771.3294869]

[39] Gönen M, Margolin AA. Localized data fusion for kernel k-means clustering with application to cancer biology. In: Proc. of the 28th Int'l Conf. on Neural Information Processing Systems (Vol. 1). Montreal: MIT Press, 2014. 1305–1313. [doi: 10.5555/2968826.2968972]

[40] Zhou SH, Liu XW, Li MM, Zhu E, Liu L, Zhang CW, Yin JP. Multiple kernel clustering with neighbor-kernel subspace segmentation. IEEE Trans. on Neural Networks and Learning Systems, 2020, 31(4): 1351–1362. [doi: 10.1109/TNNLS.2019.2919900]

[41] Brbić M, Kopriva I. Multi-view low-rank sparse subspace clustering. Pattern Recognition, 2018, 73: 247–258. [doi: 10.1016/j.patcog.2017.08.024]

[42] Li RH, Zhang CQ, Fu HZ, Peng X, Zhou JT, Hu QH. Reciprocal multi-layer subspace learning for multi-view clustering. In: Proc. of the 2019 IEEE/CVF Int'l Conf. on Computer Vision. Seoul: IEEE, 2019. 8171–8179. [doi: 10.1109/ICCV.2019.00826]

[43] Chen MS, Huang L, Wang CD, Huang D. Multi-view clustering in latent embedding space. In: Proc. of the 34th AAAI Conf. on Artificial Intelligence. New York: AAAI press, 2020. 3513–3520. [doi: 10.1609/aaai.v34i04.5756]

[44] Tao ZQ, Liu HF, Li S, Ding ZM, Fu Y. Marginalized multiview ensemble clustering. IEEE Trans. on Neural Networks and Learning Systems, 2020, 31(2): 600–611. [doi: 10.1109/TNNLS.2019.2906867]

[45] Fan SH, Wang X, Shi C, Lu EM, Lin K, Wang B. One2Multi graph autoencoder for multi-view graph clustering. In: Proc. of the 2020 Web Conf. Taipei: ACM, 2020. 3070–3076. [doi: 10.1145/3366423.3380079]

[46] Lin ZP, Kang Z. Graph filter-based multi-view attributed graph clustering. In: Proc. of the 30th Int'l Joint Conf. on Artificial Intelligence. IJCAI.org, 2021. 2723–2729. [doi: 10.24963/ijcai.2021/375]

[47] Pan EL, Kang Z. Multi-view contrastive graph clustering. In: Proc. of the 35th Int'l Conf. on Neural Information Processing Systems. 2021. 165. [doi: 10.5555/3540261.3540426]

作者简介

王玥, 博士生, 主要研究领域为软件工程, 软件重构, 智能软件开发.

孙嘉旋, 硕士生, CCF 学生会员, 主要研究领域为软件工程, 软件复用.

邹艳珍, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为软件工程, 软件复用, 知识图谱, 智能软件开发.

李宇轩, 博士生, 主要研究领域为软件工程, 软件复用, 智能软件开发.

常文辉, 硕士, 主要研究领域为软件工程, 软件复用, 智能软件开发.

谢冰, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为软件工程, 形式化方法, 软件复用, 智能软件开发.