P-Slicer:面向路径表示学习的程序切片方法

刘天阳[1],石剑君[2],叶嘉威[1],计卫星[2*]
（1. 北京理工大学计算机学院,北京 100081;2. 北京师范大学人工智能学院,北京 100875）

摘　要： 程序切片技术作为软件分析中的基础性手段,在程序理解、缺陷定位、代码重构等任务中具有重要作用. 其核心挑战在于如何在复杂控制流和数据流结构中准确识别与切片准则相关的代码片段. 近年来,基于预训练大语言模型的切片方法因其对程序语义建模能力较强而展现出良好性能,然而受限于模型输入长度限制,难以有效处理长方法体及跨过程依赖等实际场景. 针对以上问题,本文提出一种面向路径表示学习的程序切片方法 P-Slicer. 该方法首先通过构建基于语法结构的控制流图,从中提取多条可能的执行路径,以实现高代码覆盖率并保留上下文信息;随后,采用基于学习的分类模型对方法内部语句进行切片相关性判断;最后,结合变量的定义-使用传播机制,实现跨过程切片的递归分析. 该方法在保持可扩展性的同时,融合了语义理解能力,提升了切片结果的准确性与实用性. 实验结果表明,P-Slicer在切片任务中取得了 95.95% 的准确率、86.89% 精确度和 88.95% 的召回率,且在处理长方法和跨过程切片时仍能保持良好性能,表明其在软件工程领域中的良好应用前景.

关键词： 程序切片;路径提取;跨过程分析
基金项目： 国家自然科学基金(No.62232003)
中图分类号： TP311　　**文献标识码：** A　　**文章编号：** 0372-2112(2025)11-3894-16
电子学报 URL:http://www.ejournal.org.cn　　**DOI:**10.12263/DZXB.20250824

P-Slicer: A Program Slicing Approach Based on Learning Path Representations

LIU Tian-yang[1], SHI Jian-jun[2], YE Jia-wei[1], JI Wei-xing[2*]
(1. *School of Computer Science and Technology*, *Beijing Institute of Technology*, *Beijing* 100081, *China*;
2. *School of Artificial Intelligence*, *Beijing Normal University*, *Beijing* 100875, *China*)

Abstract: Program slicing is a foundational technique in software analysis, indispensable for tasks such as program understanding, defect localization, and code refactoring. Its primary challenge is to precisely identify code fragments related to a given slicing criterion within complex control and data flow structures. Recently, program slicing approaches based on pre-trained large language models have shown promising results, owing to their strong capability in capturing program semantics. However, due to the model's limitation on input length, it is difficult to handle practical scenarios such as long methods and interprocedural dependencies. To address these problems, this paper proposes P-Slicer, a program slicing approach based on learning path representations. This approach first extracts multiple execution paths by building a control flow graph based on the syntactic structure to achieve high code coverage while preserving contextual information. Then, a learning-based classification model is employed to determine the relevance of each statement to the slice criterion. Finally, a variable define-use propagation mechanism for variables is employed to achieve interprocedural slices by recursive analysis. The approach integrates semantic comprehension while preserving the scalability, thereby enhancing the accuracy and practicality of the slicing results. The experimental results demonstrate that P-Slicer achieves 95.95% accuracy, 86.89% precision, and 88.95% recall on slicing task, while maintaining robust performance when handling long methods and interprocedural slices, indicating its promising potential for application in the software engineering.

Key words: program slicing; path extraction; interprocedural analysis
Foundation Item(s): National Natural Science Foundation of China (No.62232003)

收稿日期:2025-09-22;录用日期:2025-11-04;责任编辑:覃怀银
*通讯作者:计卫星

1 引言

程序切片（program slicing）是一种程序分析技术，旨在从源程序中提取与特定变量或语句（称为切片准则，slicing criterion）相关的代码片段[1,2]. 静态程序切片基于程序的控制流图（Control Flow Graph，CFG）和数据依赖图（Data Dependence Graph，DDG），在不执行程序的前提下，确定所有可能影响切片准则的程序点. 程序切片技术通过识别影响切片标准的所有语句，有效简化程序结构，从而降低程序理解和分析的复杂度[3]. 这一特性使其在多个软件工程领域得到了广泛应用，包括但不限于程序调试[4~6]、缺陷检测[7~10]等方向.

目前，面向对象程序切片的主流方法主要基于系统依赖图（System Dependence Graph，SDG）和图可达性分析构建分析框架[11~13]. 典型的静态切片流程通常包含以下几个关键步骤：首先，根据切片准则确定需要关注的程序行为；其次，构建SDG，以形式化方式刻画程序中的数据与控制依赖关系；随后，从切片准则出发，沿SDG中的边进行正向和反向传播，标记所有可能影响该准则或被该准则影响的语句节点；最终，基于识别出的依赖路径提取与切片准则在语义上相关的代码片段. 然而，SDG的构建过程具有较高的复杂度，需对程序的控制流与数据流进行精确分析，尤其对于复杂程序而言，通常带来较大的时间与空间开销[14,15].

近年来，随着深度学习技术在程序理解领域的广泛应用[16~18]，基于神经网络的代码分析方法逐渐兴起[19~21]. Yadavally等人[19]提出了一种基于学习的程序切片方法NS-Slicer. 该方法利用预训练模型对程序进行上下文感知建模，从而捕捉目标变量与其相关语句之间的语义关联. 实验结果显示，该方法在完整代码的后向切片和正向切片任务中分别取得了90.12%和85.37%的F1值. 然而，NS-Slicer性能受限于当前主流代码预训练模型的输入长度限制. 由于其依赖预训练大模型对代码进行嵌入表示，输入序列长度最大token数通常被限制为512，导致该方法在处理长方法或跨过程切片时可能丢失关键上下文信息，进而影响切片精度与完整性.

基于学习的程序切片预测研究主要面临以下三个方面的挑战：（1）难以有效处理长方法：主流代码预训练模型受限于最大输入token数量，无法对长方法中的完整代码进行建模，导致上下文信息丢失，影响切片精度；（2）难以支持跨过程切片：现有方法采用语句级分类，无法在不同方法间扩展切片准则，限制了对跨过程依赖关系的建模能力；（3）程序结构信息利用不足：尽管GraphCodeBERT等模型通过数据流图增强语义表示，但对程序的控制流结构建模较为薄弱.

针对上述挑战，本文提出了一种面向路径表示学习的程序切片方法P-Slicer. 为应对预训练模型在处理长方法和程序结构信息建模方法的局限性，P-Slicer将原始代码按控制流分析生成多条可能的执行路径，并从中提取信息丰富、能组合覆盖更多代码的关键路径子集，分别对每条路径进行建模与分析. 该策略有效缩短了输入序列长度，缓解了输入token数量限制，同时保留了各路径的局部语义完整性. 针对现有方法难以支持跨过程切片的问题，P-Slicer根据切片语句类型，显式识别可能跨过程且影响切片准则的关键变量，并追踪其在其他方法中扩展出的切片准则，从而实现跨过程依赖关系的建模与切片传播.

在本文中，我们提出并实现了一种新型程序切片方法，在保证高精度的前提下，有效应对当前基于学习的切片方法所面临的关键技术挑战. 本文的主要贡献如下：

（1）提出一种面向路径表示学习的程序切片方法P-Slicer. 该方法将程序按执行路径进行划分，并对每条路径进行局部建模，有效缓解了预训练模型在处理长方法时所面临的输入长度限制，同时保留了程序语义的局部完整性，显著提升了模型对长方法的建模能力.

（2）引入控制流感知机制，增强程序结构信息建模能力. 通过显式建模代码片段中的控制流关系，P-Slicer能更准确地区分在不同条件下被执行的语句，从而提升切片结果的语义一致性与预测准确性.

（3）设计变量级语句追溯机制，实现跨过程切片. 通过追踪变量的定义-使用关系，P-Slicer实现了从语句到变量的细粒度语义建模，支持在已有切片基础上实现跨过程切片准则的扩展和切片结果的提取.

（4）全面评估了P-Slicer在等多种典型切片场景下的性能. 在单方法、长方法及跨过程等任务中进行了系统实验，结果验证了该方法在各类切片任务中的有效性与良好的泛化能力.

2 相关工作

2.1 基本程序切片技术

程序切片技术自Weiser等人[4]于1984年提出以来，已经成为软件分析的核心手段之一. 其核心目标是从源程序中提取所有可能影响特定程序点（即切片准则）行为的语句集合，从而为程序理解、调试、维护及缺陷定位等任务提供有效支持[3,4,22].

传统的静态程序切片方法大致可以分为两类. 第一类是基于数据流分析方程的切片方法[23]，其核心思想是在控制流图上迭代求解数据流方程，识别变量定义与使用之间的传播路径，并结合控制依赖关系逐步构建切片结果. 此类方法无需显式构建全局依赖图，空间开销较低，适用于资源受限环境. 然而，在处理并发

程序或涉及跨过程调用等复杂场景时,数据流分析可能难以提供足够精确的信息,导致切片效率显著下降.第二类方法是基于程序图表示与图可达性分析的切片技术.该类方法通过构建程序的结构化中间表示在图结构上进行可达性分析,从而提取切片结果.Ottenstein等人[23]首次将程序依赖图(Program Dependence Graph,PDG)用于程序切片计算.PDG以有向图形式刻画单个子程序内部的控制依赖与数据依赖关系,其中节点代表程序语句或控制点,边表示相应的数据或控制依赖.在此基础上,Graham等人[12]提出了SDG,作为PDG的扩展形式.SDG在保留PDG内部依赖建模能力的基础上,引入函数间调用关系的表示,从而实现对整个程序系统的依赖关系建模.基于图可达性的切片方法通常以PDG或SDG为基础,从给定切片准则出发,沿依赖边进行正向或反向遍历,收集所有可达节点构成切片结果.此类方法在图结构构建完成后,可在近似线性时间内完成切片提取,具有较高的运行效率.然而,PDG和SDG的构建依赖于完整的控制流与数据流分析,涉及大量中间表示和依赖关系推导,因此通常带来较大的时间与空间开销[14,15].

除了上述主流方法,程序切片领域还发展出多种针对特定需求的技术.增量切片根据数据依赖关系的类型逐步合并不同类别的依赖信息,动态扩展切片范围[24],该方法适用于需要逐步细化分析粒度的场景.屏障切片允许程序员指定“屏障”语句,限制切片时仅关注特定区域内的代码路径[25],有助于聚焦关键执行路径,辅助调试与分析.并发切片则扩展了传统控制流图与程序依赖图,以建模线程间的交互与同步关系,为多线程程序的切片分析提供形式化基础[26].此外,基于观察的切片能够有效应对传统依赖图方法在处理复杂语言特性、不完整代码和动态行为时所面临的挑战,并生成语义更精确的切片结果[26~30].其核心思想是通过迭代删除程序中非目标元素,并观察目标行为的变化,从而识别出对其执行具有实质性影响的代码片段.尽管基于观察的切片在切片精度方面表现优异,但其依赖于大量程序变异与执行验证,这会产生显著的性能开销,限制了其在大规模程序或频繁变更场景中的实际应用.

2.2 基于学习的程序切片技术

近年来,随着机器学习技术的进步和大规模代码预训练模型(Pre-trained Language Models,PLMs)在程序理解领域的广泛应用,研究人员开始尝试将这些新兴技术与传统依赖分析方法相结合,构建了基于学习的程序切片技术[19,20].这为克服传统静态切片中高昂的图构建代价提供了新的思路.

为了应对基于观察的切片方法在可扩展性方面的局限性,Lee等人[31]提出了一种基于依赖概率估计的近似依赖分析方法,旨在保证一定切片质量的前提下大幅提升分析效率.该方法摒弃传统的布尔型依赖关系,转而引入概率性依赖建模机制,通过机器学习模型在开发历史数据与词法特征基础上,预测任意两个程序元素之间是否存在潜在的数据或控制依赖.该概率值反映了程序元素之间的语义关联强度.

NS-Slicer[19]是基于学习的程序切片技术研究方向的代表性工作之一.它将程序语言模型应用于细粒度语句分类任务,用于判断任意语句是否属于给定变量的前向或后向切片.该方法摆脱了构建SDG所带来的高昂计算开销,在不依赖完整控制流与数据流分析的前提下,依然能够取得较高的切片精度.此外,该方法在处理不完整代码时也表现出良好的鲁棒性,进一步拓展了程序切片在实际工程场景中的适用范围.

Shahandashti等人[21]探索了大型语言模型在静态与动态程序切片中的应用潜力.他们评估了不同提示词技术的有效性,并比较了几种大型语言模型的性能表现.该研究特别关注复杂控制流导致切片失败的常见根本原因,试图通过改进模型理解和处理复杂控制逻辑的能力来提升切片准确性.这些发现不仅为未来优化基于PLM的切片技术指明了方向,还强调了结合上下文信息的重要性,以提高切片过程中的准确性和效率.

2.3 代码预训练模型

代码预训练模型是近年来程序理解与软件工程领域的重要研究方向,其核心思想是:在大规模源代码语料库上采用自监督学习方法,预先训练通用的代码表示模型,并在具体任务(如代码摘要生成、代码搜索、缺陷检测等)中进行微调[32~37].该类模型通常基于Transformer架构,通过多层自注意力机制建模代码的语法结构与语义依赖关系,能够有效捕捉上下文信息及长距离控制流特征[35~38].

目前已涌现出多个代表性代码预训练模型.其中,CodeBERT[33]是最早获得广泛关注的双模态预训练模型,其输入同时包含自然语言描述与编程语言代码,采用掩码语言建模和替换标记检测作为预训练任务,有效建模代码语义与文档描述之间的映射关系,在多个下游任务中表现出色.GraphCodeBERT[35]在此基础上引入程序的数据流图,结合传统程序分析技术与深度学习方法,通过建模变量定义-使用路径提升了模型对程序语义的理解能力.

然而,现有代码预训练模型仍存在若干局限.首先,多数模型基于线性代码序列建模,难以充分保留程序的结构化语义信息[39],导致在程序切片、程序修复等需要高精度语义理解的任务中性能受限.其次,当前大

多数基于Transformer架构的代码预训练模型（如CodeBERT[33]、GraphCodeBERT[35]、PLBART[40]和CodeT5[41]）均采用固定长度的输入序列，这一限制在建模长上下文任务中成为关键技术瓶颈．例如，在处理包含复杂控制流与数据依赖的长函数或跨文件代码片段时，模型可能因上下文缺失而无法完整捕获相关信息，从而影响分析效果．

为缓解token长度限制对模型上下文建模的影响，已有研究尝试引入滑动窗口[42,43]、稀疏注意力[44]等机制进行扩展．尽管这些方法在一定程度上缓解了长度限制带来的信息缺失问题，但仍存在计算效率低、信息冗余高及模型结构复杂等缺陷，尚未形成统一有效的解决方案．Zhang等人[45]针对长代码与复杂结构中漏洞特征提取困难的问题，提出一种基于执行路径提取的方法．该方法通过建模多个执行路径中的线性语句序列，简化了神经网络对代码语义的建模过程，为长代码表示学习提供了一种新思路．

综上所述，代码预训练模型在程序理解任务中展现出强大的语义建模能力，并在多项软件工程下游任务中取得显著进展．然而，在程序结构与语义依赖要求更高的程序切片任务中，现有模型仍面临诸多挑战．首先，主流模型受限于固定输入长度，难以有效建模长方法等复杂场景，导致关键上下文信息丢失；其次，尽管已有工作尝试引入数据流图以增强语义表示，但对控制流结构的建模仍较薄弱，难以准确捕捉条件分支、循环结构等影响语句可达性的关键程序特征；此外，当前多数基于学习的方法局限于语句级别的分类任务，缺乏对变量间细粒度依赖关系的建模能力，难以扩展到跨过程切片任务．

上述问题表明，如何在有限输入长度下建模长距离语义依赖，如何融合程序的数据流与控制流信息以提升语义一致性，以及如何实现从语句到变量层面的细粒度追溯机制，已成为基于学习的程序切片技术发展的关键瓶颈．针对上述挑战，本文提出一种融合路径建模与变量追溯机制的程序切片新方法P-Slicer. 该方法通过执行路径划分缓解输入长度限制，结合控制流感知机制增强程序结构建模能力，并引入变量级追溯机制实现跨过程切片，从而在保证精度的前提下有效应对当前技术瓶颈．

3　程序切片方法P-Slicer

如图1所示，P-Slicer的整体工作流程分为两个主要阶段：方法内切片和过程间切片，逐步解决了长代码下的程序切片问题．在方法内切片阶段，P-Slicer首先对目标方法的源代码进行解析，生成抽象语法树（Abstract Syntax Tree，AST）．基于该AST，进一步构建基于语法的CFG以刻画代码的基本控制结构．随后，提取从方法入口到切片准则以及从切片准则到方法出口的执行路径．接下来，P-Slicer利用预训练模型分别对每条执行路径及其对应的切片准则中的变量进行编码表示．在此基础上，针对前向切片与后向切片任务分别训练相应的分类模型，从而实现对当前方法的精确切片．

在获取方法内切片结果的基础上，P-Slicer进一步引入基于语句类型的变量追溯机制，以实现过程间切片．具体而言，首先识别出与切片准则相关的语句，并基于变量定义-使用关系分析涉及的变量集合，从而确定可能影响或受该切片准则影响的变量集合．若某变量存在逃逸行为（如作为参数传递至其他方法），P-Slicer将在其被调用的方法中继续执行切片操作．通过迭代该过程，最终实现对整个程序的增量式跨过程切片．

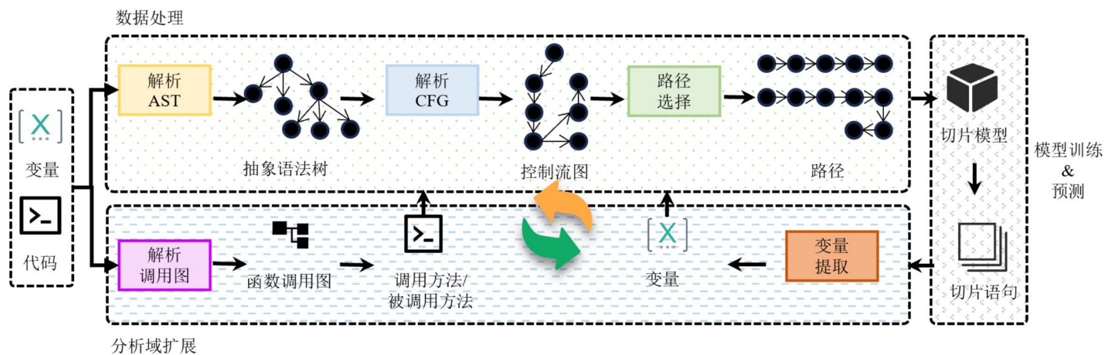


图1　P-Slicer整体框架图

3.1　基于路径表示学习的过程内切片

本阶段的目标是针对给定的Java程序代码及其切片准则，识别出前向切片语句集合（被该准则所影响的所有语句）和后向切片语句集合（影响该准则的所有语句）．整个过程如图2所示，主要包括以下几个步骤：

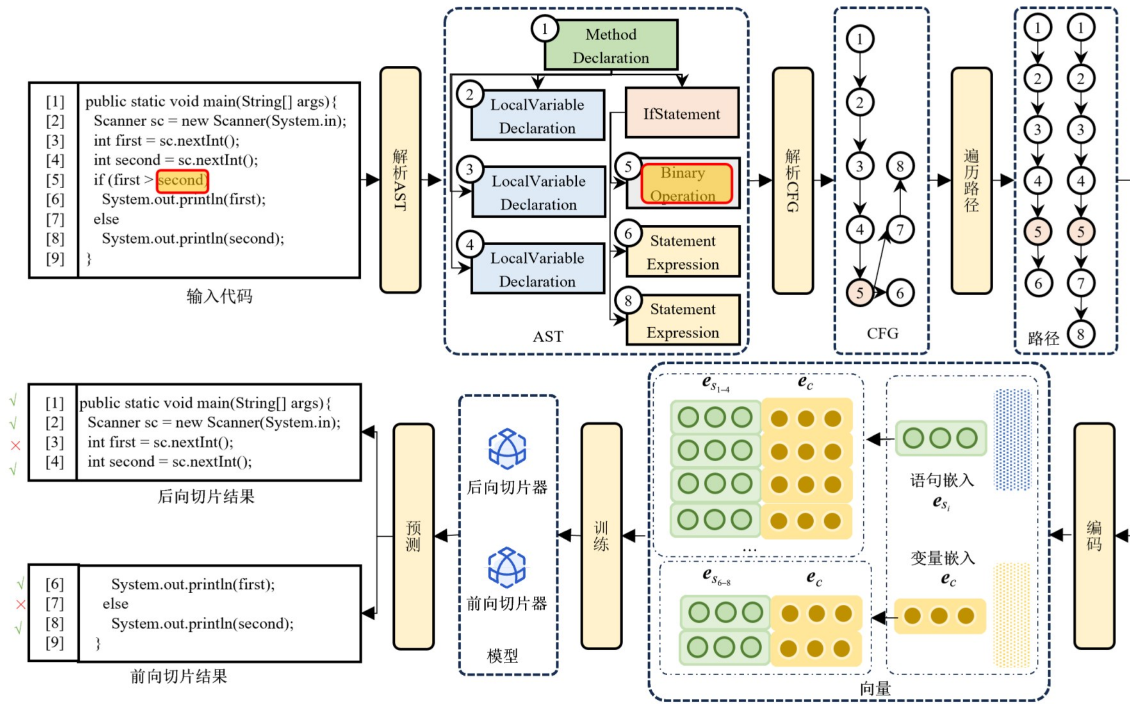
（1）控制流图构建．基于AST生成程序的CFG，用于刻画程序中各语句之间的控制转移关系．

（2）执行路径提取．从方法入口到切片准则，以及

从切片准则到方法出口两个方向上分别提取语义丰富的部分执行路径.

(3)代码表征与模型训练.采用GraphCodeBERT模型对每条执行路径进行语义表示学习;针对前向与后向切片任务,分别训练二分类模型以判断路径中各语句是否属于切片语句.

(4)结果合并.汇总所有执行路径上的预测结果,生成最终的程序切片.



注:○内标识行号.

图2 过程内基于路径的切片预测

本阶段方法的核心优势体现在两个方面:一方面,通过将程序切分为多条执行路径分别进行处理,有效缓解了编码器的学习模型在长代码建模中因输入长度限制导致的上下文截断问题;另一方面,结合程序结构信息与预训练模型的语义表示能力,显著提升了切片结果的准确性与鲁棒性.

3.1.1 控制流图构建

控制流图是程序分析中用于描述执行路径的数据结构.本文基于Javalang工具解析Java源码,生成抽象语法树,并在此基础上构建基于语法的控制流图.为精确刻画程序控制流,本文将AST节点划分为四类主要类型:

(1)简单类型节点:表示独立语句,如赋值语句、变量声明等;

(2)跳转类型节点:表示非顺序执行的语句,如break、continue等;

(3)复合类型节点:包含多个子语句,如条件分支(if)、循环结构(for、while)等;

(4)其他类型节点.如try-catch异常处理结构(try-catch-finally)、同步语句(synchronized)、返回语句等.

表1展示了部分不同类型节点对应的CFG构建示例.与传统CFG不同,本文构建的CFG节点以代码行号为标识.此外,本文在构建方式上与Zhang等人[45]所采用的基于语法的控制流图构建方法有所不同.针对循环与分支结构,本文采用了一种简化的展开策略,以降低循环带来的路径重复建模问题.以for循环为例,Zhang等人[45]在构建控制流图时,将循环体内各语句的出口指向循环控制语句本身,从而形成显式循环路径;而本文则将循环体语句的出口指向for循环的结束位置,从而在图结构层面避免该类回边的出现.需要说明的是,该简化策略仅发生在CFG结构层面,旨在降低控制流图的复杂度并为后续执行路径的提取提供基础.尽管本文在CFG中避免了循环回边的显式建模,但依赖关系的捕获并未因此受到影响:在切片过程中,本文结合GraphCodeBERT的语义建模能力捕获变量的定义-使用关系,从而发现潜在的数据依赖,并在迭代场景下仍能保留关键依赖关系.

3.1.2 路径选择方法

为实现过程内程序切片的高覆盖率与准确性,本文提出一种路径选择方法.该方法旨在从构建的CFG中选择从方法入口节点到切片准则点(后向路径)以及从切片准则点到方法出口节点(前向路径)的可能的执行路径,从而为后续代码语义建模提供结构基础.

表 1　不同语句类型的 CFG 构建示例

语句类型	局部代码片段	局部 AST	局部 CFG
简单节点	… [3] int first = sc.nextInt(); [4] int second = sc.nextInt(); …	root → LocalVariable Declaration ③; root → LocalVariable Declaration ④	③→④
IfStatement	… [5] if (first > second) [6] System.out.println(first); [7] else [8] System.out.println(second); [9] …	IfStatement → condition → Binary Operation ⑤; then_ statement → Statement Expression ⑥; else_ statement → Statement Expression ⑧	⑤→⑥, ⑤→⑦, ⑦→⑧, ⑧→⑨, ⑥→⑨
循环节点	… [5] for (int i = 0; i < 10; i++) [6] System.out.println(i); [7] …	ForStatement → control → ForControl ⑤; body → Statement Expression ⑥	⑤→⑥, ⑥→⑦, ⑤→⑦
Switch Statement/ 跳转语句	[3] … [4] switch (s) { [5] case "a": [6] System.out.println("a"); [7] break; [8] case "c": [9] break; [10] } [11] …	Switch Statement → expression; cases → Member Reference ④; Switch StatementCase → case → Literal ⑤, statements → Statement Expression ⑥, Break Statement ⑦; Switch StatementCase → case ⑧, statements → Break Statement ⑨	④, ⑤, ⑥, ⑦, ⑧, ⑨, ⑩

注:○内标识行号.

尽管全覆盖路径提取有助于提升切片精度,但在实际应用中面临"路径爆炸"问题. 具体而言,一方面,CFG 中可能存在循环结构,导致路径数量呈指数级增长. 为此,本文采用有限展开策略,即仅将循环体展开一次,而非无限展开或完全忽略. 另一方面,即使在无循环的情况下,由于控制流图中存在多个分支节点(即出度大于 1 的节点),路径组合仍可能迅速膨胀. 假设每个分支节点产生 m 条路径,且存在 n 个此类节点,则理论上路径总数可达 m^n,带来严重的组合爆炸问题. 通过实证观察可以发现,这些路径中往往包含大量重复或高度相似的节点序列,其对最终切片结果的贡献和代码缩减的贡献有限. 因此,亟需一种兼顾覆盖率与效率的路径选择算法,能够在合理控制路径数量的同时,尽可能保留关键语义信息,避免无效路径带来的性能负担.

在路径提取过程中,本文设计了一种兼顾覆盖率与效率的路径选择算法. 对于任意给定的方法 m 及其对应的切片准则 C,我们定义两类关键路径集合:

(1)后向路径集合 $P_{\text{backward}}=\{p_1,p_2,\cdots,p_n\}$:表示从方法入口到切片准则点 C 的所有简单路径;

(2)前向路径集合 $P_{\text{forward}}=\{q_1,q_2,\cdots,q_m\}$:表示从切片准则点 C 到方法出口的所有简单路径.

路径选择算法的基本思路如下:对路径集合按长度排序后,使用贪心算法优先选择路径较长且涉及 token 数量不超过阈值 θ 的路径. 接下来路径选择的具体步骤为:

第 1 步:按路径长度对 P_{backward} 进行降序排序;

第 2 步:选择其中最长且 token 数量不超过 θ 的路径 p_i;

第 3 步:将路径 p_i 中的所有节点标记为已覆盖;

第 4 步:在所有满足小于 token 数量阈值 θ 且能贡献新节点的路径中,选择新增节点最多的一条路径;

第 5 步:重复上述过程,直到所有候选路径均被处理完毕,或控制流图中所有节点均已实现全覆盖.

已有研究中采用贪心策略进行路径选取[45],例如优先选择覆盖更多代码的路径且长度较短的路径,以在路径数量受限时取得较好的代表性与效率平衡. 此类方法在诸如缺陷检测等任务中具有一定优势. 然而,程序切片是一项面向语句粒度的分析任务,其核心目标是对每条语句是否对切片准则产生影响做出判断. 若仅选取部分路径,则可能导致某些语句未被充分建模,从而影响最终切片结果的完整性与准确性. 若仅关

注于长度更短路径,则可能在切片任务中语句的上下文建模不全面. 本文的做法在考虑更全面建模上下文的同时,实现代码的基本覆盖.

3.1.3 代码表征与模型训练

本文借鉴NS-Slicer[19]的代码表征方法与神经网络架构,用于程序切片任务中的代码语义建模与切片预测. 具体而言,本文采用GraphCodeBERT对每条执行路径所对应的代码段进行编码,并为每个语句和切片准则变量构建聚合表示,以捕捉其语义特征. 设执行路径涉及的代码段$P'=\{s_1,s_2,\cdots,s_N\}$,其中$s_i$表示第$i$行语句,包含若干词法单元token. 我们将整个程序输入GraphCodeBERT,获取每个token的上下文感知嵌入向量. 在此基础上,通过语句级聚合操作,分别获得每条语句的整体语义表示以及切片准则中涉及变量的语义表示. 随后,构建两个独立的多层感知机(MLP),分别用于前向切片与后向切片的预测任务. 对于任意语句s_i与切片准则变量v,将语义嵌入拼接作为输入,输出为该语句是否属于对应切片的概率值.最终,设定阈值为0.5,筛选出概率高于该阈值的语句,构成最终的切片结果集合.

3.2 基于语句类型的过程间切片

在程序分析中,基于学习的跨过程切片面临的核心挑战是如何跨越方法边界,追踪变量传播路径. 为解决这一问题,本文提出一种基于语句类型的过程间切片方法,其核心思想是:通过静态分析Java语句的语法结构,系统性地识别与切片准则相关的变量,并判断其是否具有逃逸行为.

设m为当前分析的方法,$v\in V$为其中的一个变量. 若存在调用上下文C,使得v的值被传递至C的某个变量(前向逃逸);或v的值来源于C的某个变量(后向逃逸),则称变量v在方法m中发生逃逸. 变量逃逸的判断有助于区分局部依赖与跨方法依赖,是实现精确过程间切片的前提.

根据切片方向的不同,逃逸变量的处理方式也有所区别:对于前向切片,若变量逃逸至被调用方法(callee),则将该变量在callee中首次使用的程序点作为切片准则;对于后向切片(backward slicing),若变量逃逸至调用者(caller),则将该变量在caller中的调用本方法的程序点作为切片准则,若变量逃逸至被调用方法(callee),则将callee的返回语句作为切片准则. 在获得callee和caller的方法代码及其对应的切片准则后,可分别对二者执行局部切片. 随后,将调用链上所有方法的局部切片结果进行合并,最终形成完整的过程间切片结果.

3.2.1 相关变量识别

本阶段基于语句类型细化前向和后向切片中相关变量提取策略,明确各类语句在依赖传播中的角色. Java语句按照语义结构可以分为:表达式语句、声明语句、调用语句、控制流语句和其他语句. 本文针对每一类语句分别定义了适用于前向切片(即识别受切片准则影响的变量)和后向切片(即识别影响切片准则的变量)的变量提取规则,具体示例如表2所示. 表达式语句和声明语句在前向切片中提取可能受影响的左值变量,在后向切片中则关注提供数据的右值变量,二者建立了变量定义与使用的依赖关系;方法调用语句根据是否存在返回值来决定前向或后向切片中的相关变量提取,强调了跨方法的数据依赖,而控制流语句无论前向后向均需提取影响执行路径的变量,跳转语句与其他类型通常不直接参与数据依赖传播.

表2 不同语句类型的相关变量提取

语句类型	示例	前向切片相关变量	后向切片相关变量
赋值语句	$x = y + z$;	x	y, z
自增/自减语句	i++;	i	i
声明语句	int $m = n + p$;	m	n, p
方法调用	int a = func(c, d);	func(), c, d	a
控制流语句	if (x > y)	x, y	x, y
循环结构	for(int i = 0; i < n; i++)	i, n	i, n

3.2.2 跨过程切片

在本文第3.1节中,已经实现了基于学习的过程内切片机制. 第3.2.1节中简单识别了变量之间的依赖关系. 这一过程构成了后续实现过程间切片分析的基础. 在进行跨方法切片时,关键挑战在于如何跨越方法边界传播切片准则,并准确识别变量之间的跨方法依赖关系. 具体而言,存在两个核心问题:其一,需要明确哪些变量接收了来自调用者的影响(后向切片),或向被调用方法传递了影响(前向切片);其二,如何在调用链上下文中合理定义每个方法的切片准则,以确保切片结果的完整性与一致性. 为此,本文提出了一种基于相关变量集与参数交集的变量影响识别机制.

在前向切片过程中,若当前方法包含对其他方法的调用,则需识别哪些变量通过该调用将影响传递至被调用方法(callee). 为此,本文首先提取当前调用语句所涉及的实参集合,并结合当前方法中所有前向切

片语句的相关变量集合以及当前方法的切片准则变量集，计算出可能传入 callee 的变量集合．具体而言，设当前方法的切片准则变量集为 V_s，调用语句的实参集合为 P_{actual}，前向切片语句的相关变量集合为 $R_{forward}$，则可能传入被调用方法的变量集合为

$$V_{out}=P_{actual}\cap(V_s\cup R_{forward}) \tag{1}$$

以图 3 中的 compare 方法为例，在其第 3 行中，first 变量为切片准则，因此 $V_s=\{first\}$；该行调用语句的实参集合为 $P_{actual}=\{first\}$，且无其他前向切片语句，因此 $R_{forward}=\varnothing$．我们可以获得：

$$V_{out}=\{first\}\cap\left(\{first\}\cup\varnothing\right)=\{first\} \tag{2}$$

即变量 first 是从前向切片中传入被调用方法 isOddorEven 的变量．

在后向切片过程中，若当前方法接收了其他方法的返回值，则将该方法作为相关变量．例如后向切片语句 c = func()；中 func() 为相关变量．在后向切片过程中，若当前方法被调用，则需识别哪些变量从调用者(caller)接收了影响．为此，我们提取当前方法的形式参数集合，并结合当前方法中所有后向切片语句的相关变量集合以及当前方法的切片准则变量集，计算出可能从调用者接收影响的变量集合．设当前方法的切片准则变量集为 V_s，后向切片语句的相关变量集合为 $R_{backward}$，形参集合为 P_{formal}，则可能从调用方法接收影响的变量集合为

$$V_{in}=P_{formal}\cap(V_s\cup R_{backward}) \tag{3}$$

仍以图 3 为例，在 compare 方法中，后向切片提取的相关变量集合为 $R_{backward}=\{first, second\}$，切片准则变量集为 $V_s=\{first\}$，方法形参集合为 $P_{formal}=\{first, second\}$．我们可以获得：

$$\begin{aligned}V_{in}&=\{first, second\}\cap\left(\{first\}\cup\{first, second\}\right)\\&=\{first, second\}\end{aligned} \tag{4}$$

即变量 first 和 second 均可能受到调用方法 main 的影响．

在获取了影响传入与传出的变量集合之后，下一步是为调用链上的每个方法定义合适的切片准则．若相关变量为方法，则切片准则为该方法的返回值．对于调用者(caller)中的切片准则，处理较为直接，即将调用语句本身及相关的实参变量集合设定为切片准则．而对于被调用方法(callee)中的切片准则，本文并未直接使用 callee 方法头中的形参作为切片起点，而是选取相应参数首次被使用的程序点及其相关变量作为切片准则．这是因为在构建训练数据集的过程中，为了保持前向切片样本与后向切片样本数量的比例在 0.3 至 0.7 之间，减少了以方法头为切片准则的样本．如果在实际应用中仍将切片准则固定于方法头位置，模型可能无法有效识别此类模式，从而导致切片失败．

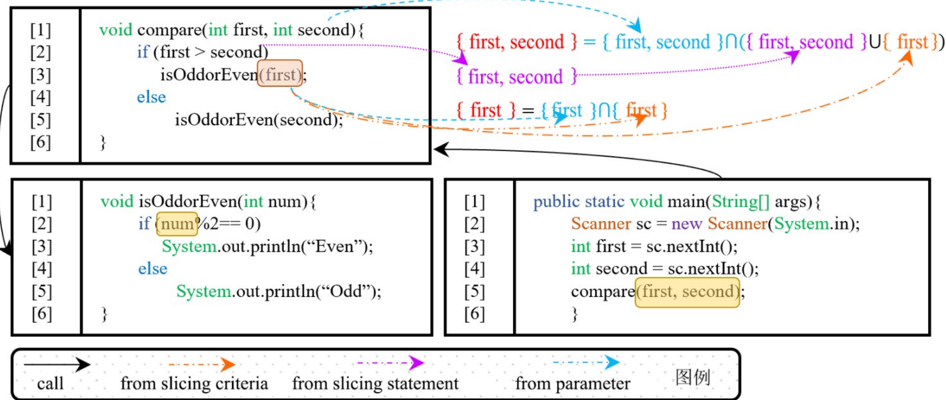


图 3　跨过程切片示例

4　实验设计

4.1　研究问题

为了全面评估 P-Slicer 在程序切片任务中的性能表现，并深入分析其技术优势与适用场景，本文设计了一系列实验，以验证以下五个关键研究问题(Research Questions，RQs)，为方法改进与实际应用提供理论依据与实证支持．

RQ1：相较于现有先进基线方法，P-Slicer 在过程内切片的有效性如何？

RQ2：路径选择算法的有效性如何？

RQ3：P-Slicer 在长方法切片场景下的表现如何？

RQ4：P-Slicer 在跨过程切片中的有效性如何？

RQ5：P-Slicer 在程序切片任务中的效率如何？

RQ1 旨在评估 P-Slicer 在单方法或单过程内部识别与切片准则相关的语句的能力，并与当前主流的基于学习的程序切片工具进行对比．通过引入精确率(Precision)与召回率(Recall)作为核心评估指标，验证 P-Slicer 在过程内切片的有效性，以说明路径简约策略对

精度无显著影响. RQ2考察不同执行路径数量、不同代码覆盖率对切片性能的影响. RQ3聚焦于方法体较长、控制流结构复杂的方法场景,考察P-Slicer是否能够保持较高的切片精度与稳定性,验证其在处理大规模代码时的鲁棒性. RQ4关注跨方法的切片任务,检验P-Slicer在跨过程依赖建模方面的有效性. RQ5从运行效率角度出发,衡量P-Slicer训练和预测时长,进而评估其可扩展性与工程实用性.

4.2 实验数据

为全面评估本文所提出的程序切片方法P-Slicer的有效性,本文选用了NS-Slicer[19]所采用的CodeNet数据集作为基础数据源. 该数据集由大量使用Java编写的编程问题构成,并已按照问题类型划分为训练集(包含200个问题)、验证集(25个问题)和测试集(25个问题). 在样本分布方面,NS-Slicer[19]数据集中对正例(即应被包含在切片中的语句)与负例(不应被包含的语句)进行了平衡处理,有利于后续模型训练过程中的类别均衡学习以及性能评估的客观性.

在上述数据集基础上,本文进一步对其进行了扩展. 具体而言,对于每段代码,依据第3.1.2节所述的路径提取方法,生成多条组合执行路径,并将每条路径及其切片结果作为独立数据实例进行建模. 该策略使模型能够在更细粒度的控制流上学习切片规则,从而提升处理复杂程序结构的能力与泛化性能,并增强对不同路径的训练建模能力. 在实际测评时,我们不仅在单路径上做了性能评估,还在完整代码级别的数据集(即NS-Slice所使用的原始基准数据集)上进行了验证.

具体实现时,本文对每条执行路径独立调用JavaSlicer[46]生成切片标签. 由于JavaSlicer仅能针对完整代码进行切片,因此本文将前向路径和后向路径组合. 为了合理配对向路径,首先按路径涉及的代码token数量升序对P_{backward}和P_{forward}排序,依次将后向路径与组合次数最少的前向路径配对;遍历完成后,将剩余前向路径与组合次数最少的后向路径继续匹配,直至全部完成. 该策略既提高整体覆盖率、避免路径组合数量指数增长,又能防止模型重复学习相同的前向或后向代码片段.

4.3 基准模型及评价指标

为系统评估本文提出的程序切片方法P-Slicer的有效性,本文将其与当前具有代表性的基于学习的切片技术NS-Slicer[19]进行对比实验. NS-Slicer是一种近年来提出的新型程序切片预测工具,能够适用于完整代码与不完整代码场景下的切片生成任务. 该方法的核心思想在于利用预训练的源代码语言模型,捕捉变量与语句之间的细粒度语义依赖关系,并分别构建前向与后向切片预测模型,以实现对程序中影响目标语句或被目标语句影响的代码片段的识别.

在评估指标方面,本文借鉴了NS-Slicer中所采用的标准评价体系. 首先,在语句粒度上,采用了分类任务中广泛使用的四类基本指标:准确率$\text{A-S}=\dfrac{\text{TP}+\text{TN}}{\text{TP}+\text{TN}+\text{FP}+\text{FN}}$、精确率$P=\dfrac{\text{TP}}{\text{TP}+\text{FP}}$、召回率$R=\dfrac{\text{TP}}{\text{TP}+\text{FN}}$以及F1分数$\text{F1}=2\times\dfrac{P\times R}{P+R}$. 其中,TP表示的是模型正确识别的切片语句;FP表示模型错误识别为切片语句的非切片语句;TN为模型正确识别为非切片语句的语句;FN为模型漏检的真实切片语句. 此外,为了更全面地衡量模型在切片结构层面的整体预测能力,本文同时引入了NS-Slicer中切片级匹配指标A-EM(Accuracy of Exact Match). 该指标用于统计模型在每段代码中预测的前向切片与后向切片均完全等同于基准切片的比例. A-EM能够有效反映模型在实际应用场景中对完整切片结构的还原能力,弥补了语句粒度指标在结构一致性方面的不足. 此外,本文在程序级别上分别计算了各个评价指标的数值,并对所有程序结果取算数平均,记为Overall. 上述评价指标共同构成了一个从局部到整体、从语句级别到切片结构级别的多层次评估体系,可用于支撑本文后续研究问题的验证与分析.

4.4 实验方法及设置

为了有效捕捉代码中的数据依赖关系,本文选用GraphCodeBERT[35]作为基础预训练语言模型. GraphCodeBERT是一种面向代码理解的图增强型预训练模型,能够融合程序的结构信息与语义特征,从而提升对代码上下文的理解能力. 该模型支持的最大输入token数量为512,在本实验中,本文提取其最后一层隐藏层输出作为语句的嵌入表示,该嵌入向量的维度为768. 所有实验均在配备两块NVIDIA RTX 3090 GPU的环境下进行. 为兼顾训练效率与模型稳定性,训练过程中采用的批次大小为64,初始学习率设为1×10^{-4},优化器参数中AdamW的ε值设置为1×10^{-8},以防止数值不稳定问题.

5 实验结果与分析

5.1 过程内切片有效性(RQ1)

本节旨在评估P-Slicer在单方法Java程序切片任务中的性能表现,并将其与当前最先进的基于学习的切片工具NS-Slicer进行对比. NS-Slicer所采用的编码器与本文一致,均为GraphCodeBERT. 实验所用的数据集包含两类,一类为基准测试数据集,由完整的单方法Java代码组成;另一类为单路径数据集,即将基准数据集中的每段代码按照本文提出的路径提取方案进行拆分后得到的路径级别的代码片段,表3展示了在上述两

类数据集上P-Slicer和NS-Slicer的评估结果.

表3　在单方法上的切片结果对比

数据集	方法	切片准则	评价指标/%				
			A-EM	A-S	*P*	*R*	F1
基准数据集	NS-Slicer	Backward	42.95	84.66	90.10	86.77	88.41
		Forward	40.83	86.26	83.24	89.10	86.07
		Overall	41.89	85.44	87.19	87.70	87.44
	P-Slicer	Backward	43.60	85.32	91.75	86.22	88.90
		Forward	39.61	86.65	80.14	93.67	86.38
		Overall	41.61	85.94	86.89	88.95	87.91
单路径数据集	NS-Slicer	Backward	43.60	84.32	90.42	86.44	88.39
		Forward	54.16	89.61	88.00	92.73	90.30
		Overall	48.88	86.85	89.37	89.01	89.19
	P-Slicer	Backward	46.46	85.20	91.19	86.95	89.02
		Forward	53.13	89.48	87.95	92.51	90.17
		Overall	49.79	87.24	89.79	89.22	89.50

从表3可以看出，在单路径数据集上，P-Slicer在除A-S外的各项指标上均略优于NS-Slicer. 这一结果主要归因于NS-Slicer的训练数据中加入了额外的单路径样本，使其在该类数据上的表现相对更优. 而在基准测试数据集上，尽管P-Slicer采用将完整方法划分为多条路径并分别进行切片、最终合并结果的策略，其整体性能仍与NS-Slicer相当，或略有提升. 具体而言，P-Slicer在准确率、召回率和F1值上的表现分别为85.94%、88.95%和87.91%. 这一结果表明，本文提出的路径简约程序切片机制不仅能够有效支持对程序的建模与分析，同时在不牺牲模型准确性的前提下，实现了对复杂控制流结构的处理.

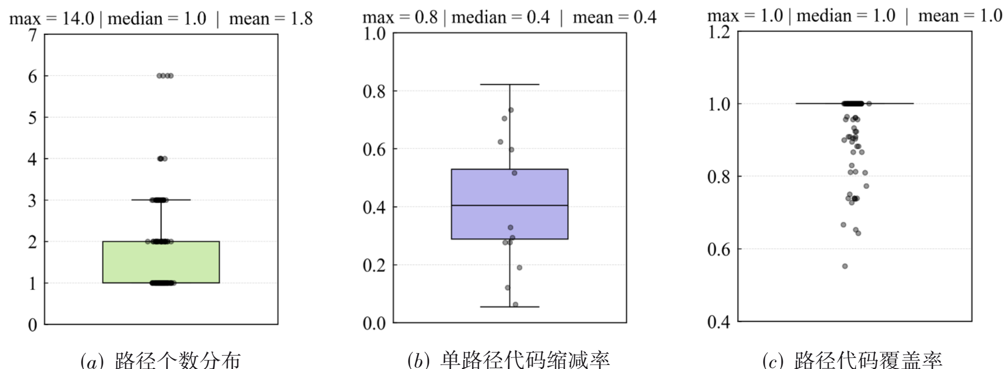
总结1：实验结果表明，P-Slicer在保持与现有先进方法相当性能的基础上，通过路径级别的建模方式，提升了对完整程序结构的切片能力.

5.2　路径选择算法的有效性(RQ2)

在5.1节，本文对P-Slicer的整体性能进行了评估，并与现有的先进方法NS-Slicer进行了对比，验证了其在程序切片任务中的有效性. 在此基础上，本节进一步聚焦于路径选择算法的设计与实现，旨在深入分析路径数量、代码覆盖率等关键因素对切片效果的影响，从而验证本文路径划分策略在建模过程中的合理性与实用性. 如第3.1.2节所述，本文在路径划分过程中采用了特定的路径选取策略与组合方式，尽可能保留更多上下文信息.

图4展示了测试集中路径划分结果的统计分布，包括路径数量的分布、单条执行路径的代码缩减率分布，以及各代码片段拆分后的整体代码覆盖率. 其中，路径数量是指根据前文所述方法将一段完整方法拆分为若干条执行路径的数量. 实验结果表明，大多数代码片段被划分为1至2条路径，仅少数情况下被拆分为最多14条路径，说明本文路径划分策略在保持路径数量可控的同时，也兼顾了程序结构的复杂性. "路径代码缩减率"定义为每条执行路径相对于原始代码所减少的代码量比例. 从图中统计结果可见，平均每条路径可缩减约10%的代码内容，部分路径的缩减比例甚至高达60%. 这一结果表明，通过路径划分能够有效去除与当前执行无关的代码分支，从而显著减少模型输入的token数量. 这对于缓解当前主流基于Transformer的模型在处理长序列时所面临的token数量限制问题具有重要意义，尤其有助于提升对大型代码片段的切片能力，避免因输入长度限制而导致的切片失败. "路径代码覆盖率"则表示将所有路径合并后所能覆盖的原始代码比例. 理论上，覆盖率越高，越有助于获得完整且准确的切片结果. 实验数据显示，在绝大多数情况下，多条路径合并后的覆盖率达到了100%，说明本文提出的路径划分方法能够全面覆盖原始代码的执行行为，为后续切片提供了坚实的基础保障.

总结2：本文提出的路径划分方法在控制路径数量的同时，实现了较高的代码缩减率与覆盖率. 该策略为解决模型输入长度限制所带来的实际问题提供了可行路径.



(*a*) 路径个数分布　(*b*) 单路径代码缩减率　(*c*) 路径代码覆盖率

图4　路径个数分布、代码缩减率及覆盖率等指标

本节进一步探究了执行路径数量和代码覆盖率对P-Slicer切片性能的具体影响，相关实验结果分别如图5与图6所示．图5展示了不同路径数量下P-Slicer在各项评估指标上的表现．整体来看，路径数量对模型性能的影响相对有限．当路径数量小于7时，准确率、精确度、召回率与F1值等关键指标均较为稳定，其中准确率A-S始终维持在80%以上，表明模型在处理中低复杂度代码结构时具有良好的鲁棒性与一致性．然而，当路径数量达到或超过7时，各项指标开始出现较为明显的波动．例如，当路径数量为8时，召回率下降至60%；在路径数量为12时，精确度也降至约60%．这一现象表明，随着路径数量的增加，代码的控制流结构趋于复杂，路径之间的语义差异增大，模型在路径级建模过程中面临更大的挑战，从而影响了整体的切片效果．尽管如此，P-Slicer在路径数量较多的复杂场景下，仍能保持准确率在70%以上，体现出较好的稳定性与鲁棒性．

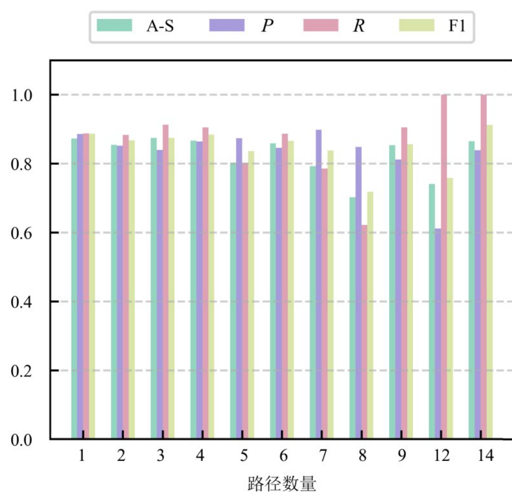


图5 P-Slicer在不同路径数量下的评估结果

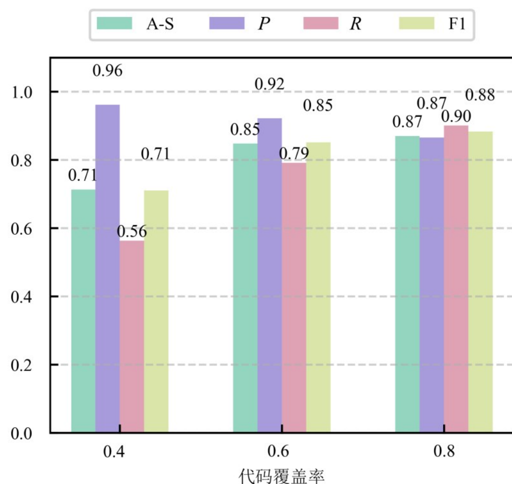


图6 P-Slicer在不同代码覆盖率下的评估结果

图6展示了在不同代码覆盖率条件下，P-Slicer在准确率（A-S）、F1值、精确率等关键指标上的表现趋势．从实验结果可以看出，随着代码覆盖率的提升，切片结果的准确率和F1值得到明显改善．这一现象具有直观合理性：未被路径覆盖的代码语句在切片过程中默认被归类为非切片语句，因此更高的覆盖率意味着模型能够对更多代码语句进行有效地分类决策，从而提升整体预测的完整性与一致性．然而，令人意外的是，在覆盖率低于40%的情况下，精确率仍能达到96%以上．这表明，在覆盖率较低的场景下，P-Slicer仍展现出更强的判别能力，能够更为精准地识别出关键的切片语句．造成这一现象的原因在于，当代码覆盖率较低时，所提取的执行路径通常较为简单，控制流结构相对清晰，路径之间的语义干扰较少，从而有助于模型更准确地判断语句是否属于切片结果．这种"路径简化效应"一定程度上提升了模型的性能，也反映出路径划分策略在降低建模复杂度方面的潜在优势．

总结3：P-Slicer在路径数量较少时表现出良好的稳定性与一致性，而在路径数量较多的复杂结构中虽存在性能波动，但仍具备较强的切片能力；代码覆盖率对P-Slicer的切片性能具有一定影响，尤其在准确率和F1值方面呈现出正相关趋势，然而，模型在低覆盖率场景下仍表现出优异的精确性，反映出路径划分机制在简化程序结构、提升模型判别能力方面的有效性．

5.3 长方法的有效性（RQ3）

本节进一步评估P-Slicer处理长方法的有效性．主流基于Transformer的预训练模型（如GraphCodeBERT）在输入长度上存在token数量限制，导致现有基于该模型的切片工具（如NS-Slicer）在处理长方法时面临输入截断或直接失效的问题．为验证P-Slicer在该类场景下的适应能力，本文在CodeNet数据集中随机选取了75个未包含在基准数据集中的长方法样本，这些方法的token数量均超过512．

P-Slicer通过路径划分机制，从完整方法中提取若干执行路径进行分段建模，从而在一定程度上规避了模型输入长度限制带来的影响．最终，我们在长方法样本上进行了评估，结果如表4所示．实验表明，尽管P-Slicer在处理长方法时仍表现出一定的切片能力，但其性能相较于完整方法场景有所下降．具体而言，准确率A-S由86.89%下降至61.58%，召回率由88.95%下降至48.72%．这一下降趋势主要源于路径划分过程中部分代码信息的丢失，以及模型在片段化路径上建模能力的削弱．然而值得注意的是，P-Slicer在长方法场景下仍保持了较高的精确度，整体维持在80%以上，说明路径级建模机制在一定程度上仍能保证对切片语句的判别准确性．为深入分析其性能变化的原因，本文进一步统计了这些长方法在路径划分过程中的路径数量、代

码缩减率与覆盖率等关键指标,如图7所示.结果显示,长方法的平均路径数量由1.8条增加至3.2条,表明其控制流结构更为复杂;同时,代码覆盖率由接近100%下降至70%左右,说明路径划分虽能缓解输入限制问题,但也带来了信息覆盖不全的挑战.

总结4:P-Slicer在处理长方法程序时,虽因路径划分导致整体性能有所下降,但仍展现出一定的切片能力与较高的精确性.路径级建模策略在缓解模型输入限制问题方面具有可行性,但其在长方法场景下的性能优化仍需进一步探索.

表4 P-Slicer针对长方法的评估结果

切片准则	评价指标/%			
	A-S	*P*	*R*	F1
Backward	60.47	85.74	48.63	62.06
Forward	63.39	77.97	48.87	60.08
Overall	61.58	82.92	48.72	61.37

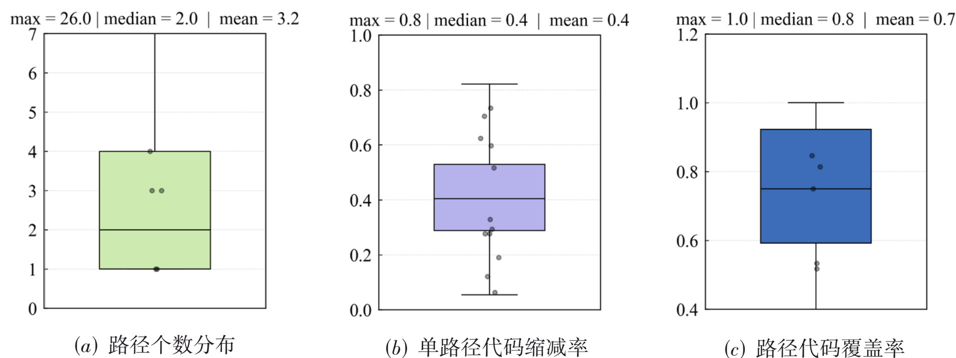


图7 长代码下路径个数分布、代码缩减率及覆盖率等指标

5.4 跨过程切片的有效性(RQ4)

在验证了P-Slicer在单方法程序及长方法场景下的切片能力之后,本节进一步评估其在跨过程切片任务中的有效性.跨过程切片涉及多个方法之间的数据流与控制流依赖,是程序切片中的关键挑战之一.为了构建具有代表性的测试集,本文从CodeNet数据集中选取了包含多个方法的Java文件,并基于这些文件随机选择代码行和变量作为切片标准,共生成397个测试实例.其中,154个实例包含变量在方法间传播的真实路径,属于典型的跨过程切片场景;其余243个实例虽未涉及跨过程传播,但仍被保留用于评估P-Slicer是否能够准确识别非跨过程切片,从而全面测试其对跨过程行为的判别能力.为了获得准确的切片标签,本文首先使用JavaSlicer[46]对397个实例进行切片处理.然而,由于JavaSlicer不支持跨过程切片功能,因此对于存在跨过程传播的154个实例,本文采用人工标注的方式,明确变量在不同方法间的传播路径,并在相应方法中分别调用JavaSlicer进行局部切片,最终整合生成完整的跨过程切片结果作为参考标准.此外,在统计各项评估指标时,本文排除了仅包含右花括号的代码行,以避免语法结构对评估结果产生干扰.

表5展示了P-Slicer在全部397个测试实例上的整体评估结果,表6则聚焦于其中154个存在实际跨过程传播的实例.通过对比分析,可以进一步评估P-Slicer在跨过程依赖识别与建模方面的能力.根据表5中的实验结果可以看出,P-Slicer在整体表现上具有较高性能,准确率高达97.69%,表明其能够有效识别与切片准则中目标变量相关的代码片段.在后向切片方向上,P-Slicer表现尤为稳定,准确率为96.63%,精确度高于92.03%,召回率为88.49%,F1值达到90.22%.而在前向切片方向上,尽管精确度有所下降,准确率仍保持在99.11%,召回率为95.30%,F1值为74.94%.精确度下降主要源于正向切片中路径分支更为复杂,导致部分无关路径被纳入切片范围,从而影响了模型的判别精度,另外一个原因是在测试数据中前向方向传播相对较少,可能存在一定误差.从表6来看,在实际存在跨过程传播的154个测试实例中,P-Slicer仍然表现出良好的切片性能,说明其在处理跨过程依赖关系方面具有较强的建模能力.这一结果进一步验证了本文方法在依赖图构建过程中引入的上下文敏感机制和过程间传播路径识别策略的有效性.

表5 P-Slicer针对多方法的评估结果

切片准则	评价指标/%				
	A-EM	A-S	*P*	*R*	F1
Backward	48.08	96.63	92.03	88.49	90.22
Forward	71.79	99.11	61.77	95.30	74.96
Overall	57.78	97.69	89.40	88.87	57.78

总结5:P-Slicer在跨过程切片任务中展现出较高的准确性和良好的判别能力,尤其在反向切片上表现稳定.

5.5 切片效率(RQ5)

表7展示了在CodeNet基准数据集上,N-Slicer与P-

表6 P-Slicer针对跨过程切片的评估结果

切片准则	评价指标/%				
	A-EM	A-S	*P*	*R*	F1
Backward	49.01	96.91	92.30	90.89	49.01
Forward	60.13	98.53	50.97	94.59	66.25
Overall	52.75	97.52	88.75	91.07	89.90

Slicer在训练与预测阶段的时间开销对比,同时给出了传统切片工具Joern在测试集中切片深度为5时的时间开销. 从数据可以看出,P-Slicer在训练和预测阶段所耗费的时间均略高于N-Slicer. 这一现象是可预期的,主要原因在于P-Slicer在训练过程中引入了单条执行路径上的切片样本,从而增加了模型训练的数据复杂度与计算负担,N-Slicer的训练数据为30 806条,P-Slicer的训练数据有43 568条. 在预测阶段,P-Slicer不仅需要对程序中的执行路径进行提取,还需对每条路径分别进行预测,并将各路径的切片结果进行合并处理,进一步导致了时间成本的上升. 尽管如此,考虑到训练过程通常是一次性完成的,且可以在离线环境下进行,因此其额外开销在整体应用中并不构成显著负担. 而在预测阶段,虽然P-Slicer的单次推理时间有所增加,但其带来的切片精度提升以及对复杂程序结构的更好建模能力,使得这种时间代价在多数实际应用场景中是可接受的. 此外,路径提取与合并过程的耗时仍处于合理范围内,未对整体效率造成明显影响. 此外,相较于传统切片工具,基于学习的程序切片方法在切片效率方面表现更优. 这主要可能因为Joern在执行切片时需要进行完整的语义解析和图构建,而基于学习的方法通过模型预测切片结果,避免了繁琐的图计算与数据流追踪,从而显著降低了时间开销.

表7 切片训练及预测效率

方法	训练	预测
NS-Slicer	385.27 s/epoch	35.08 s
P-Slicer	557.55 s/epoch	56.60 s
Joern	—	> 24 h

总结6:P-Slicer在训练与预测阶段的时间开销均有所增加,但是其增加幅度在可接受范围内,能够在时间成本和性能之间实现良好的平衡.

6 有效性威胁

本文方法在设计过程中充分考虑了当前大模型输入长度限制对程序分析任务的影响,并基于路径简约策略划分代码片段以适配模型输入容量. 然而,在面对复杂且规模较大的源码时,仍可能存在某些执行路径所包含的token数量超出预训练语言模型所能接受的最大输入长度的情况. 此时,该路径将被舍弃,从而可能导致部分代码上下文无法被有效覆盖. 在最坏情况下,若某段代码的所有可行执行路径均超过模型输入限制,则系统将无法为该代码生成任何有效的切片结果,进而影响整体方法的有效性. 未来,我们将继续结合其他手段,例如稀疏注意力模型等在不显著增加计算量的前提下扩大上下文窗口.

此外,本文通过抽象语法树构建控制流图,并在此基础上选取若干具有代表性的执行路径用于后续分析. 受限于计算资源和效率考虑,本文并未穷举所有可能的执行路径,而是优先选择路径长度由长至短排列的若干条路径,以提高代码覆盖率并保留更多上下文信息. 尽管这种策略在一定程度上缓解了因路径过短而导致的语义缺失问题,但较长路径通常伴随更高的token消耗,也可能造成模型推理资源的浪费,尤其是在多路径组合分析场景下,资源开销将显著上升.

再者,本文的方法不依赖于编译过程,仅基于源码层面进行分析. 这种方式虽然提高了方法的适用性和灵活性,但也带来了潜在的精度问题. 例如,控制流图的构造可能受到未解析类型信息等非结构化因素的影响,导致路径分析不够精确;同时,跨过程切片准则的提取也可能因缺少运行时语义支持而产生偏差,从而影响最终切片结果的准确性.

尽管本文方法在多数场景下能够有效应对大模型输入限制所带来的挑战,但仍存在因路径丢弃、路径选取策略及语义信息缺失等因素带来的有效性风险. 未来工作中将进一步探索更高效的路径筛选机制以及结合轻量级语义分析手段,例如在路径选择中引入基于重要性平分的覆盖有限策略,确保关键路径优先保留,以提升方法的鲁棒性与适用范围.

7 结论

本文针对当前基于预训练模型的程序切片方法在处理长方法和跨过程依赖方面存在的局限,提出了一种面向路径表示学习的程序切片方法P-Slicer. 该方法通过路径划分,缓解模型输入长度限制,结合控制流感知机制增强程序结构建模能力,并利用变量级依赖实现跨过程切片. 与已有方法相比,P-Slicer在保持分析可扩展性的同时,有效融合了语义理解能力,提升了切片结果的准确性与实用性,为基于学习的程序切片方法提供了新的建模视角与实现路径.

参考文献

[1] BINKLEY D W, GALLAGHER K B. Program slicing[M]// Advances in Computers. Amsterdam: Elsevier, 1996: 1-50.

[2] WEISER M. Program slicing[J]. IEEE Transactions on Software Engineering, 1984, SE-10(4): 352-357.

[3] XU B W, QIAN J, ZHANG X F, et al. A brief survey of program slicing[J]. ACM SIGSOFT Software Engineering

Notes, 2005, 30(2): 1-36.

[4] WEISER M. Programmers use slices when debugging[J]. Communications of the ACM, 1982, 25(7): 446-452.

[5] WOTAWA F. On the relationship between model-based debugging and program slicing[J]. Artificial Intelligence, 2002, 135(1/2): 125-143.

[6] BADIHI S, NOURJI S, RUBIN J. Slicer4D: A slicing-based debugger for Java[C]//Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. New York: ACM, 2024: 2407-2410.

[7] 段旭, 吴敬征, 罗天悦, 等. 基于代码属性图及注意力双向LSTM的漏洞挖掘方法[J]. 软件学报, 2020, 31(11): 3404-3420.
DUAN X, WU J Z, LUO T Y, et al. Vulnerability mining method based on code property graph and attention BiLSTM[J]. Journal of Software, 2020, 31(11): 3404-3420. (in Chinese)

[8] LI Z, ZOU D Q, XU S H, et al. VulDeePecker: A deep learning-based system for vulnerability detection[EB/OL]. (2018-01-05)[2025-09-30]. https://arXiv.org/abs/1801.01681.

[9] ZOU D Q, WANG S J, XU S H, et al. μ VulDeePecker: A deep learning-based system for multiclass vulnerability detection[J]. IEEE Transactions on Dependable and Secure Computing, 2021, 18(5): 2224-2236.

[10] TANG C H, SHUAI H L, YANG M M. Program defect detection using sensitive slice semantics with control flow variable dependency[J]. Recent Advances in Computer Science and Communications, 2025, 18(4): 18

[11] AZIM T, ALAVI A, NEAMTIU I, et al. Dynamic slicing for Android[C]//2019 IEEE/ACM 41st International Conference on Software Engineering. Piscataway: IEEE, 2019: 1154-1164.

[12] GRAHAM S L, HORWITZ S, REPS T, et al. Interprocedural slicing using dependence graphs[J]. ACM Transactions on Programming Languages and Systems, 1990, 12(1): 26-60.

[13] ZHANG Y Z, XU B W, GAYO J E L. A formal method for program slicing[C]//Proceedings of the 2005 Australian conference on Software Engineering. New York: ACM, 2005: 140-148.

[14] GALLAGHER K B, KOZAITIS S J. Program slicing: A brief retrospective[J]. IEEE Transactions on Software Engineering, 2025, 51(3): 720-724.

[15] CHEN J B, XIANG H J, ZHAO Z H, et al. Utilizing precise and complete code context to guide LLM in automatic false positive mitigation[EB/OL]. (2025-05-31) [2025-09-30]. https://arXiv.org/abs/2411.03079.

[16] HE J D, TREUDE C, LO D. LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead[J]. ACM Transactions on Software Engineering and Methodology, 2025, 34(5): 1-30.

[17] MICHELUTTI C, ECKERT J, MONECKE M, et al. A systematic study on the potentials and limitations of LLM-assisted software development[C]//2024 2nd International Conference on Foundation and Large Language Models. Piscataway: IEEE, 2025: 330-338.

[18] NAM D, MACVEAN A, HELLENDOORN V, et al. Using an LLM to help with code understanding[C]//Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. New York: ACM, 2024: 1-13.

[19] YADAVALLY A, LI Y, WANG S H, et al. A learning-based approach to static program slicing[J]. Proceedings of the ACM on Programming Languages, 2024, 8(OOPSLA1): 83-109.

[20] YADAVALLY A, LI Y, NGUYEN T N. Predictive program slicing via execution knowledge-guided dynamic dependence learning[J]. Proceedings of the ACM on Software Engineering, 2024, 1(FSE): 271-292.

[21] SHAHANDASHTI K K, MOHAJER M M, BELLE A B, et al. Program slicing in the era of large language models[EB/OL]. (2024-09-19)[2025-09-30]. https://arXiv.org/abs/2409.12369.

[22] CHENG X, WANG H Y, HUA J Y, et al. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network[J]. ACM Transactions on Software Engineering and Methodology, 2021, 30(3): 1-33.

[23] OTTENSTEIN K J, OTTENSTEIN L M. The program dependence graph in a software development environment[J]. ACM SIGSOFT Software Engineering Notes, 1984, 9(3): 177-184.

[24] ORSO A, SINHA S, HARROLD M J. Incremental slicing based on data-dependences types[C]//Proceedings IEEE International Conference on Software Maintenance. ICSM 2001. Piscataway: IEEE, 2002: 158-167.

[25] KRINKE J. Barrier slicing and chopping[C]//Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation. Piscataway: IEEE, 2003: 81-87.

[26] GIFFHORN D, HAMMER C. An evaluation of slicing algorithms for concurrent programs[C]//Seventh IEEE International Working Conference on Source Code Analysis and Manipulation. Piscataway: IEEE, 2007: 17-26.

[27] BINKLEY D, GOLD N, HARMAN M, et al. ORBS: Language-independent program slicing[C]//Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2014: 109-120.

[28] BINKLEY D, GOLD N, HARMAN M, et al. ORBS and the limits of static slicing[C]//2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation. Piscataway: IEEE, 2015: 1-10.

[29] GOLD N E, BINKLEY D, HARMAN M, et al. Generalized observational slicing for tree-represented modelling languages[C]//Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. New York: ACM, 2017: 547-558.

[30] BINKLEY D, GOLD N, ISLAM S, et al. Tree-oriented vs. line-oriented observation-based slicing[C]//2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation. Piscataway: IEEE, 2017: 21-30.

[31] LEE S. Scalable and approximate program dependence analysis[C]//Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings. New York: ACM, 2020: 162-165.

[32] LIU P F, YUAN W Z, FU J L, et al. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing[J]. ACM Computing Surveys, 2023, 55(9): 1-35.

[33] FENG Z Y, GUO D Y, TANG D Y, et al. CodeBERT: A pre-trained model for programming and natural languages[EB/OL]. (2020-09-18)[2025-09-30]. https://arXiv.org/abs/2002.08155.

[34] NIU C G, LI C Y, NG V, et al. SPT-code: Sequence-to-sequence pre-training for learning source code representations[C]//2022 IEEE/ACM 44th International Conference on Software Engineering. Piscataway: IEEE, 2022: 1-13.

[35] GUO D Y, REN S, LU S, et al. GraphCodeBERT: Pre-training code representations with data flow[EB/OL]. (2021-09-13)[2025-09-30]. https://arXiv.org/abs/2009.08366.

[36] LI Z Y, LU S, GUO D Y, et al. Automating code review activities by large-scale pre-training[C]//Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York: ACM, 2022: 1035-1047.

[37] GUO D Y, LU S, DUAN N, et al. UniXcoder: Unified cross-modal pre-training for code representation[EB/OL]. (2022-03-08)[2025-09-29]. https://arXiv.org/abs/2203.03850.

[38] 曹鹤玲, 刘昱, 韩栋. 基于自注意力机制神经机器翻译的软件缺陷自动修复方法[J]. 电子学报, 2024, 52(3): 945-956.
CAO H L, LIU Y, HAN D. Self-attention neural machine translation for automatic software repair[J]. Acta Electronica Sinica, 2024, 52(3): 945-956. (in Chinese)

[39] WANG Z L, LI G, LI J, et al. Line-level semantic structure learning for code vulnerability detection[EB/OL]. (2024-11-08)[2025-09-27]. https://arXiv.org/abs/2407.18877.

[40] AHMAD W U, CHAKRABORTY S, RAY B, et al. Unified pre-training for program understanding and generation[EB/OL]. (2021-04-10) [2025-09-29]. https://arXiv. org/abs/2103.06333.

[41] WANG Y, WANG W S, JOTY S, et al. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation[EB/OL]. (2021-09-02)[2025-09-28]. https://arXiv.org/abs/2109.00859.

[42] ZHANG Q L, CHEN Q, LI Y L, et al. Sequence model with self-adaptive sliding window for efficient spoken document segmentation[C]//2021 IEEE Automatic Speech Recognition and Understanding Workshop. Piscataway: IEEE, 2022: 411-418.

[43] FU Z C, SONG W T, WANG Y J, et al. Sliding window attention training for efficient large language models[EB/OL]. (2025-06-04)[2025-09-30]. https://arXiv.org/abs/2502.18845.

[44] JASZCZUR S, CHOWDHERY A, MOHIUDDIN A, et al. Sparse is enough in scaling transformers[EB/OL]. (2021-11-24)[2025-09-30]. https://arXiv.org/abs/2111.12763.

[45] ZHANG J W, LIU Z X, HU X, et al. Vulnerability detection by learning from syntax-based execution paths of code[J]. IEEE Transactions on Software Engineering, 2023, 49(8): 4196-4212.

[46] GALINDO C, PEREZ S, SILVA J. A program slicer forJava (tool paper) [C]//Software Engineering and Formal Methods. Cham: Springer, 2022: 146-151.

作者简介

刘天阳　女,1995年7月出生于河北省保定市.现为北京理工大学体系结构与高性能计算研究所博士研究生.主要研究领域为程序分析与优化.

E-mail: lty@bit.edu.cn

叶嘉威　男,2001年5月出生于浙江省临海市.现为北京理工大学体系结构与高性能计算研究所硕士研究生.主要研究领域为程序分析与优化.

E-mail: yejiawei@bit.edu.cn

石剑君　女,1991年2月出生于河南省邓州市.现为北京师范大学人工智能学院助理研究员.主要研究领域为程序分析与优化、系统软件安全.

E-mail: shijianjun@bnu.edu.cn

计卫星　男,1980年2月出生于陕西省咸阳市.现为北京师范大学人工智能学院教授.主要研究领域为计算机体系结构、程序分析与优化、并行与高性能计算.中国电子学会会员编号:E190197518M.

E-mail: jwx@bnu.edu.cn