

基于图神经网络的相似性代码检测方法

赵 涵, 崔博夕, 张卫丰

(南京邮电大学 计算机学院, 江苏 南京 210023)

摘 要: 针对现有代码相似性检测方法在语义表达能力方面的不足, 提出一种基于图神经网络的代码相似性分析方法。现有技术通常可分为基于文本、基于词法单元、基于语法结构以及基于程序依赖关系的多种类型, 但大多侧重单一层面的特征表示, 难以有效捕获程序执行过程中的深层语义信息。为解决这一问题, 构建一种融合语法结构与程序行为特征的图表示模型。具体而言, 以抽象语法树为基础框架, 通过引入控制依赖边与数据依赖边, 对原有树结构进行扩展, 从而形成包含多类型关系的程序图表示。在此基础上, 利用图神经网络对代码片段进行表示学习, 通过节点间信息传播与聚合获得代码的向量化表示, 并采用余弦相似度对不同代码片段进行相似性度量。该方法能够同时刻画程序的结构特征与语义信息, 从而提升对复杂代码相似性的识别能力。实验结果表明, 与传统方法相比, 所提方法在语义相似性检测任务中具有更好的性能表现。

关键词: 图神经网络; 代码相似性检测; 数据流; 控制流

DOI: 10.11907/rjtk.241789

中图分类号: TP311

文献标识码: A

文章编号: 1672-7800(2026)006-0048-08

扫描二维码阅读全文:



Code Similarity Detection Method Based on Graph Neural Network

ZHAO Han, CUI Boxi, ZHANG Weifeng

(School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China)

Abstract: To address the shortcomings of existing code similarity detection methods in terms of semantic representation capabilities, this paper proposes a code similarity analysis method based on graph neural networks. Existing techniques are generally categorized into text-based, lexical unit-based, syntactic structure-based, and program dependency-based methods, but most focus on single-level feature representation, failing to effectively capture deep semantic information during program execution. To solve this problem, this paper constructs a graph representation model that integrates syntactic structure and program behavior features. Specifically, based on an abstract syntax tree framework, the original tree structure is expanded by introducing control dependency edges and data dependency edges, thus forming a program graph representation containing multiple types of relationships. On this basis, a graph neural network is used to learn the representation of code fragments, obtaining vectorized representations of the code through information propagation and aggregation between nodes, and using cosine similarity to measure the similarity of different code fragments. This method can simultaneously characterize the structural features and semantic information of the program, thereby improving the ability to identify complex code similarities. Experimental results show that the proposed method has better performance in semantic similarity detection tasks compared to traditional methods.

Key Words: graph neural network; similarity code detection; data flow; control flow

0 引言

从实际应用效果出发, 代码相似性分析已成为支撑多类软件工程任务的基础技术, 其核心价值在于能够自动识

别在语法结构或程序行为上具有相似性的代码片段。该能力广泛服务于程序理解、学术不端检测、软件演化研究以及安全分析等场景。例如, 在漏洞检测中, 若某一缺陷代码被多次复制, 并稍作修改后分布于不同模块, 具备语义感知能力的相似性分析技术可以快速定位这些变体, 从

收稿日期: 2025-04-15

基金项目: 国家自然科学基金面上项目(62272214); 南京市国际合作项目(202401006)

作者简介: 赵涵(2002-), 女, 南京邮电大学计算机学院硕士研究生, 研究方向为代码仓库; 崔博夕(1997-), 男, 南京邮电大学计算机学院硕士研究生, 研究方向为代码仓库; 张卫丰(1974-), 男, 博士, CCF高级会员, 南京邮电大学计算机学院教授, 研究方向为代码管理、持续集成。本文通讯作者: 张卫丰。

而提升缺陷修复的覆盖范围与效率。

然而,从技术实现角度来看,现有方法在深层语义建模方面仍面临显著挑战。首先,源代码本身是一种高度结构化但语义隐式的表示形式,单纯依赖文本或浅层结构难以准确反映程序的真实执行行为。其次,即便将代码解析为抽象语法树(Abstract Syntax Tree, AST)能够刻画语法层级关系(如语句嵌套与表达式构成),但并未显式表达程序执行过程中变量值的传播路径(数据流)以及语句执行顺序的依赖关系(控制流)。例如,在一个条件分支语句中,AST能够描述“if-else”的结构,但无法直接体现变量在不同路径上的取值变化及其对后续计算的影响。

针对上述问题,本文从表示学习的角度出发,提出一种面向语义增强的代码相似性建模方法。该方法以AST为基础骨架,通过引入额外的关系边扩展其表达能力,使其不仅包含语法信息,还能够编码程序中的关键语义依赖。具体而言,在原有树结构中补充两类重要连接:一类用于描述控制依赖关系,例如条件判断与其支配语句之间的联系;另一类用于刻画数据依赖关系,例如变量定义与使用之间的信息传递路径。通过这种扩展方式,原本仅反映语法结构的树被转化为一种多关系图结构,从而为后续学习算法提供更加丰富的上下文信息。

在此基础上,引入图神经网络对构建的程序图进行表示学习。通过节点间的信息传播与聚合机制,模型能够逐步融合局部结构特征与全局语义信息,最终生成具有判别能力的代码向量表示。该向量可用于后续的相似性度量,从而实现对复杂代码克隆(尤其是语义相似但结构差异较大的情况)的有效识别。

综上,本文方法通过将语法结构与程序行为统一建模,并结合图学习技术,有效缓解了传统方法在语义表达方面的不足,为代码相似性检测提供了一种更具泛化能力的解决思路。

1 相关工作

相似性代码检测已成为一个活跃的研究领域,吸引了大量学者的广泛关注和深入研究。如图1所示,代码克隆被分为4种类型:①两个代码片段除了注释、代码布局和空白字符的差异外,其余部分完全相同;②两个代码片段在语法和结构上相似,但可能存在标识符、变量类型、注释以及代码布局的差异,又称参数化克隆;③在类型1或类型2的基础上,两个代码片段允许语句的增删或修改;④两个代码片段在语法结构上存在较大差异,但实现了相同的功能。例如,一个实现冒泡排序的代码和一个实现快速排序的代码被视为第四种类型的相似代码片段。此外,代码相似性检测技术主要分为以下4种:基于文本的方法、基于Token的方法、基于树的方法和基于程序依赖图的方法。

```

// original code
public int main(int m, int n)
{
    if (m>0)
        n=m+1;
    return n;
}

// Type I
public int main(int m, int n)
{
    if (m>0)
        n=m+1;
    return n;
    // some comment
}

// Type II
public int funcT2(int u, int v)
{
    if (u>0)
        v=u+1;
    return v;
}

// Type III
public int funcT3(int u, int v)
{
    int p=0;
    p=u;
    if (p>0)
        v=p+1;
    return v;
}

// Type IV
public int funcT4(int v, int u)
{
    return v > 0 ? (v + 1) : u;
}

```

Fig. 1 Code cloning types

图1 代码克隆类型

从综合性能角度来看,不同类别的代码相似性检测方法在语义表达能力与计算效率之间呈现出显著差异。因此,近年来研究逐步从浅层表示转向结构化甚至语义驱动的建模方式。然而,在具体技术路径上,各类方法仍沿着不同的信息抽象层次展开。

首先,从结论出发,基于文本表示的方法虽然实现简单,但其检测能力主要局限于表层特征。这类方法将源代码抽象为字符串或行序列,通过匹配公共子串或重复行实现相似性判断。例如,可利用序列对齐算法(如局部对齐)在字符级或行级发现相似片段。为提升该类方法的表达能力,部分研究引入神经网络模型对序列进行建模。Quradad等^[1]对循环神经网络在代码克隆检测中的应用进行了系统分析,指出基于LSTM的模型在捕获上下文依赖关系方面具有优势,能够一定程度上识别语法及语义层面的相似性。此外,针对编译差异带来的干扰问题,Schott等^[2]通过对Java字节码进行规范化处理,并借助Jimple中间表示与优化技术,有效消除了绝大多数由编译环境引起的差异(超过99%),从而为后续相似性分析提供了统一输入。Glani等^[3]结合逐级抽象策略与Smith-Waterman局部序列对齐方法,实现了从简单克隆到跨间隙复杂克隆的检测,在精度与召回率方面均优于传统方法;其后续工作通过引入灵活的签名匹配机制,进一步提升了对Type-II与Type-III克隆的识别能力,并能够定位克隆间的差异区域^[4]。

相较而言,基于词法单元的检测方法在鲁棒性方面有所提升,其核心思想是通过中间表示降低语法噪声的影响。具体流程通常包括两个阶段:首先,对源代码进行词法分析,将其转化为由关键字、标识符和运算符构成的令牌序列;其次,在该序列上执行相似性计算,例如,通过检测重复子序列或构建索引来进行匹配。围绕这一框架,已有大量优化工作。例如,Nishi等^[5]提出自适应预过滤策略,并结合Delta倒排索引提升检索效率;Tekhandani等^[6]利用有向无环图(DAG)结构建模代码演化过程,从版本控

制系统中提取克隆信息;Farhadi等^[7]设计了可扩展的汇编代码克隆搜索系统 ScalClone,并引入局部敏感哈希(LSH)以支持近似匹配;YuKi等^[8]则采用多粒度分析策略,并结合 Smith-Waterman 算法识别相同的哈希序列;Sajnani等^[9]开发的 SourcererCC 工具利用倒排索引实现大规模代码库中的高效查询;此外,Li等^[10]提出 CCLEARNER 方法,将令牌序列输入深度神经网络(DNN)进行表示学习,从而增强模型对潜在语义模式的捕捉能力。

在进一步提升结构表达能力的基础上,基于语法树的方法通过显式建模程序的层次结构来进行相似性分析。该类方法首先将源代码解析为抽象语法树(AST),再通过树匹配或子结构搜索算法来计算相似度。例如,Yang等^[11]在多个开源项目(Jdk、Ant、Tomcat、Antlr 和 DnsJava)上开展实验,其主要通过对 AST 进行结构转换,并结合序列对齐算法计算相似性分数,从而提升检测效果。此外,Pati等^[12]利用 AST 从 ArgoUML 代码库中提取克隆组件,并结合时间序列模型(如 ARIMA 与神经网络模型)预测克隆代码的演化趋势,为软件维护提供决策支持。

在语义层面建模最为充分的是基于程序依赖关系的分析方法。这类方法通过构建程序依赖图(PDG),同时刻画控制依赖与数据依赖,从而更接近程序的实际执行语义。例如,在变量定义与使用之间建立数据依赖边,在条件语句与受控语句之间建立控制依赖边。基于此表示,可以采用图同构或相似性度量方法进行比较。Sargsyan等^[13]提出了一种基于 LLVM 基础设施的克隆检测方法,通过构建 PDG 并应用同构算法实现高精度检测;Raghithwet-sagul等^[14]则提出将代码转换为图像(PNG 格式)进行处理,并利用 Jaccard 相似度进行匹配。值得注意的是,尽管该类方法在语义表达方面具有明显优势,但由于图构建过程复杂、计算成本较高,其在大规模场景中的应用仍面临挑战。

国内研究同样在探索更高效的结构化表示方法。曾杰等^[15]提出基于程序向量树的检测方法,通过将语法结构映射为向量形式以降低匹配复杂度,但由于未显式建模控制流与数据流,其在检测语义差异较大的克隆(如 Type-IV)时表现受限。随后,曾杰等^[16]又提出基于序列模式匹配的 API 误用检测方法,通过分析 API 调用序列中的模式特征识别异常行为。尽管该方法在特定场景中效果显著,但其适用范围主要局限于接口级分析。

从应用视角来看,代码相似性检测技术在软件工程实践中具有多方面价值:①在学术场景中,可用于识别不同开发者提交代码之间的潜在抄袭行为;②在代码审查过程中,可自动发现重复代码块,从而降低维护成本;③在安全分析中,可通过检测漏洞代码的克隆变体来追踪缺陷传播路径;④在代码重构过程中,可辅助发现可合并的重复逻辑。例如,在漏洞检测场景中,当某段存在缓冲区溢出问题的代码被多次复制时,通过相似性分析可以快速定位所

有相关变体,从而显著提升漏洞修复的效率与覆盖范围。

2 基于图神经网络的相似代码检测

为提升代码相似性分析在语义层面的判别能力,本文首先给出结论性设计:采用一种融合多源程序信息的图表示,并结合图神经网络进行表示学习,以实现代码片段的高质量向量化建模。在此基础上,通过向量空间中的相似度度量完成最终判断。

从建模角度来看,单一结构难以全面刻画程序特性。一方面,抽象语法树(Abstract Syntax Tree, AST)主要反映语法层级关系,即程序由哪些语法构件组成以及其嵌套结构,但其本质上忽略了程序执行过程中涉及的控制路径(control flow)和数据传播(data flow)。另一方面,尽管可以通过编译或中间表示将程序转换为控制流图(CFG),以描述语句执行顺序,但该类结构在图学习场景中同样存在不足。具体而言:①控制流图的连接关系相对稀疏,边数量较少,这会限制图神经网络中信息在节点间的传播范围,进而影响表示学习效果;②控制流图中的节点通常对应完整语句或表达式,而非细粒度的词法单元,若采用简单嵌入方式(如基于词袋模型),容易丢失关键语义信息。例如,一条包含函数调用的语句,其内部参数关系与调用语义难以通过粗粒度进行完整表达。

针对上述问题,本文提出一种改进的程序图构建方式:以 AST 为基础结构,在其节点之间引入多类型边以补充语义信息。这些新增连接用于显式编码程序中的控制依赖与数据依赖关系。例如,将条件判断节点与其支配语句通过控制边连接,将变量定义与使用之间通过数据边关联。通过这种方式,原本仅包含语法信息的树结构被扩展为一种“增强型程序图”,能够同时表达程序的结构特征与执行语义。

在表示学习阶段,首先,将每个代码片段转换为上述增强图结构;其次,将该图输入图神经网络模型,通过多层邻域聚合机制不断更新节点表示,使其融合来自局部及全局上下文的信息;再次,经过迭代传播,可获得整个代码片段的向量表示;最后,利用基于角度的相似度度量(如余弦相似性)对两个代码向量进行比较,从而量化其相似程度。

在模型训练过程中,采用均方误差(Mean Squared Error, MSE)作为优化目标,使模型学习到的向量表示,在相似样本与不相似样本之间具有良好的区分能力。整体训练流程可概括为4个阶段:①进行代码解析与图结构构建;②完成图表示的特征初始化;③通过图神经网络进行表示学习;④基于相似度计算与损失函数进行参数优化。通过上述流程,模型能够逐步学习到兼具语法与语义信息的代码表示,从而提升相似性检测的准确性。代码相似性检测方法的具体训练过程可分为4个步骤,如图2所示。

首先,在程序表示层面,为弥补传统抽象语法树(AST)

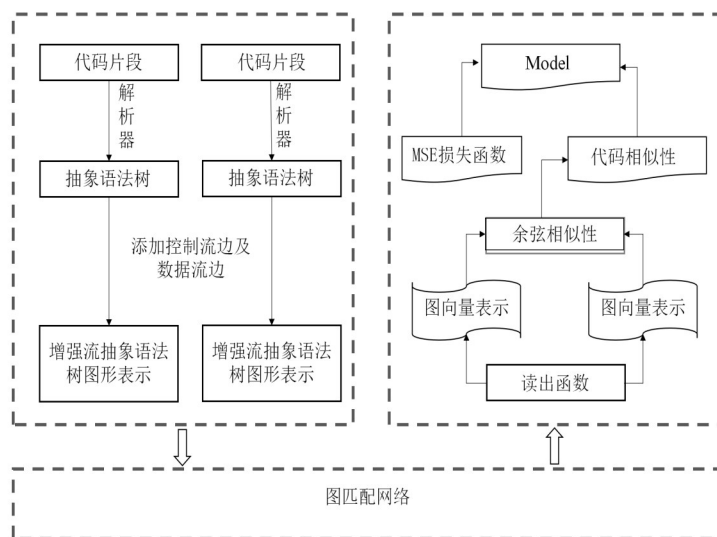


Fig. 2 Code similarity detection training process based on graph neural network

图2 基于图神经网络的代码相似性检测训练过程

仅刻画语法层级而忽略执行语义的不足,本文对其进行结构扩展。具体而言,在原有树结构的基础上,引入两类关键关系:一类用于描述程序执行路径约束的控制依赖,例如条件语句与其分支体之间的支配关系;另一类用于反映变量值传播过程的数据依赖,例如变量定义与后续使用之间的关联。通过增加上述多类型边,原本的树结构被转化为一种包含丰富语义信息的程序图,从而为后续分析提供更充分的上下文信息。

其次,在表示学习阶段,考虑到直接对图结构进行匹配计算复杂度较高且难以扩展至大规模数据,本文引入图神经网络对程序图进行编码。通过图级读出函数(readout function)将节点表示进行聚合,生成整个代码片段的低维向量表示。该过程可以理解为由将复杂的图结构压缩为固定长度的语义向量,从而便于后续高效比较。例如,不同实现方式但功能相同的排序算法,在该表示空间中应具有较高的相似性。

随后,在相似性度量阶段,将获得的代码向量映射到同一特征空间,并采用基于夹角的相似性度量方法(即余弦相似度)计算两个向量之间的接近程度。该方法能够有效衡量向量方向的一致性,从而避免因尺度差异带来的干扰,适用于表示学习场景中的相似性评估。

最后,在模型优化过程中,将相似性预测任务转化为回归问题进行处理。具体而言,当计算得到的相似度高于预设阈值时,可将对应代码片段视为克隆样本对;反之则视为非克隆对。在此基础上,采用公式(1)中的均方误差(Mean Squared Error, MSE)作为损失函数,对模型输出的相似度与真实标签之间的差异进行最小化。由于该任务的预测结果为单一实值(即两个代码片段之间的相似度),因此模型输出维度设定为一维($d=1$)。通过上述训练机制,模型能够逐步学习到区分相似与不相似代码的判别

特征。

$$MSE = \frac{1}{d} \sum_{i=1}^d (y_i - \hat{y}_i)^2 \quad (1)$$

其中, d 是 y_i 和 $\hat{y}_i$ 的维数, $y_i - \hat{y}_i$ 是真实值与预测值之间的差异。

2.1 增强流边添加

为弥补传统程序结构表示在语义建模方面的不足,本文首先对抽象语法树进行扩展,构建一种包含多类型关系的程序图表示,从而增强对代码执行行为的刻画能力。与仅反映语法层级关系的标准抽象语法树(AST)不同,该扩展结构不仅保留原有的语法信息,还通过引入多种连接关系显式编码程序中的局部上下文与潜在依赖。

从设计角度来看,标准AST的局限在于其仅描述“组成关系”,即程序由哪些语法单元构成以及它们的嵌套方式,而无法体现语句之间的顺序联系或变量的使用轨迹。例如,在一段连续赋值与使用变量的代码中,AST能够表示赋值语句的结构,但无法直接反映变量在不同语句之间的传播路径。为此,本文在原有树结构基础上引入多种边类型,以增强信息表达能力。

具体而言,所构建的程序图包含以下五类核心关系:①Child边,用于连接非叶节点与其子节点,保留语法树的自上而下层次结构;②Parent边,作为Child关系的逆向连接,用于从子节点回溯至其父节点,从而支持双向信息传播;③NextSib边,用于连接同一层级中的相邻节点,使模型能够感知同级语句或表达式之间的顺序关系,如同一段代码块中的多个语句;④NextToken边,用于连接词法层面的相邻终端节点,即将源代码中按顺序出现的标识符或关键字依次连接,这有助于保留原始代码的线性上下文信息;⑤NextUse边,用于关联同一变量在不同位置的定义与使用节点,从而近似刻画数据流关系。例如,在变量被

多次读取或更新的场景中,该类边能够显式表示变量的使用链路。

通过上述多关系建模方式,原本单一的树结构被扩展为一个具有丰富语义连接的图结构。这种表示不仅保留了语法信息,还引入了顺序信息与数据依赖特征,为后续基于图神经网络的表示学习提供了更具表达力的输入,从而有助于提升代码相似性分析的准确性。其中,加入的控制流类型主要有以下4种:If语句、While循环、For循环和顺序执行语句块Block。

(1) If控制流结构图,如图3所示。If控制流节点包含2个或者3个子节点。第1个子节点是条件节点,当条件为True时,添加Condition True边到Then子节点,当条件为False时,添加Condition False边到Else节点。

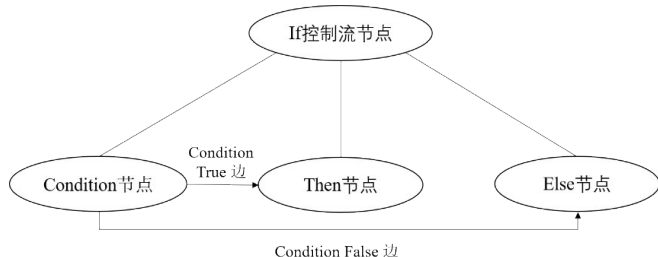


Fig. 3 If control flow structure diagram

图3 If控制流结构图

(2) While控制流结构,如图4所示。While控制流包含2个子节点:条件节点和主体节点。其中,WhileExec边从条件节点连接到主体节点,WhileNext节点从主体节点连接到条件节点,达到循环的执行过程。

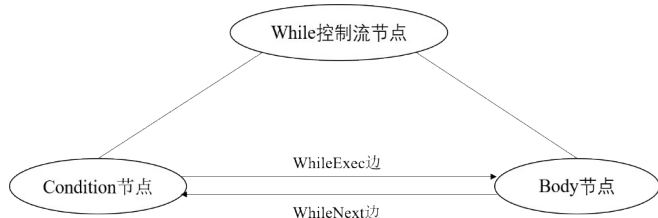


Fig. 4 While control flow structure

图4 While控制流结构

(3) For控制流结构,如图5所示。For控制流包含2个子节点:ForControl节点和body节点。其中,ForExec边从ForControl节点连接到主体节点,ForNext节点从主体节点连接到ForControl节点,达到循环的执行过程。

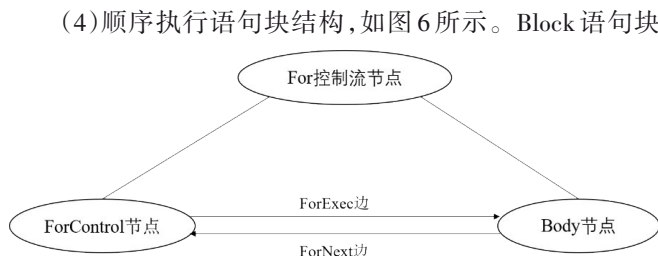


Fig. 5 For control flow structure

图5 For控制流结构

控制流节点是顺序执行语句块结构的根节点。Java中按顺序执行的语句存在于方法体或者循环体内。Block语句块控制流节点可以包含任意数量的子节点,在每个节点之间添加NextStmt边进行连接。

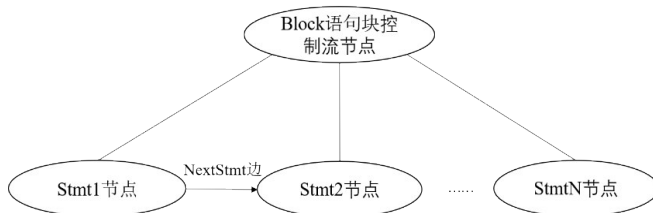


Fig. 6 Sequential execution block structure

图6 顺序执行语句块结构

在构建增强流抽象语法树的过程中,首先,将代码片段解析成抽象语法树,并通过重构语法树算法对抽象语法树进行重构;随后,得出代码片段的词典表和边属性字典edges;最终,结合增强流边添加算法表示出代码片段节点的词典索引集合、边id集合和边属性集合。具体流程如图7所示。

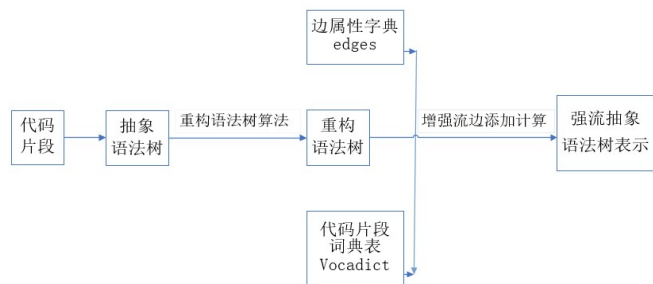


Fig. 7 Enhanced stream abstract syntax tree construction process

图7 增强流抽象语法树构建流程

2.2 抽象语法树重构算法

从代码片段到增强流语法树的图表示,以图8中的代码片段作为示例。

```
public class t1 {
    public int run(int m, int n) {
        if (m > 0) {
            n = m + 1;
            return n;
        }
    }
}
```

Fig. 8 Example code for graph neural network input

图8 图神经网络输入示例代码

通过Python的ast库对代码片段进行抽象语法树的解析,抽象语法树如图9所示。

将token化后的词建立对应的下标词典,如表1所示。

抽象语法树的重构,即在代码片段表示为抽象语法树后,利用抽象语法树重构算法添加id、token和data属性值。其中,算法输入是代码片段抽象语法树,算法输出是重构语法树newtree。

算法1 抽象语法树重构算法

输入:抽象语法树 ast。

输出:重构语法树 newtree。

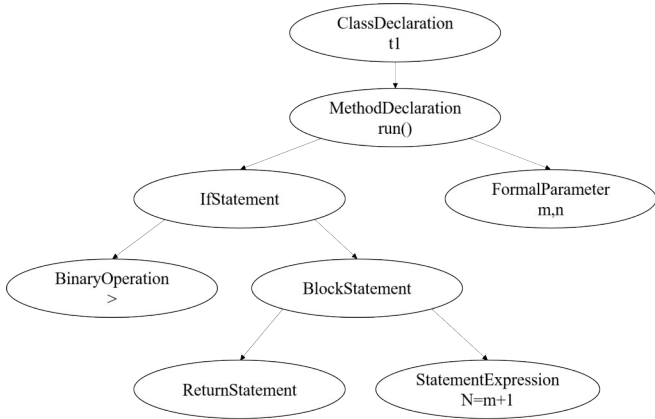


Fig. 9 Code snippet abstract syntax tree

图9 代码片段抽象语法树

Table 1 Code snippet tokenization word subscript table

表1 代码片段 token 化词下标表

词	下标
BinaryOperation	0
l	15
m	11
FormalParameter	21
=	6
BlockStatement	8
n	4
CompilationUnit	12
t1	22
StatementExpression	18
IfStatement	17
BasicType	5
MemberReference	19
Modifier	24
Literal	1
Assignment	2
int	23
+	10
ClassDeclaration	3
public	7
>	20
0	9
MethodDeclaration	14
run	13
ReturnStatement	16

1. 初始化 nodelist 集合为空
2. 初始化 newtree 为 AnyNode (id=0, token=None, data=None)
3. 定义 createTree 方法,参数为 newtree, ast, nodelist, parent=None
4. $id \leftarrow \text{len}(\text{nodelist})$ // 将 id 赋值为 nodelist 的长度
5. $\text{token}, \text{children} \leftarrow \text{get_token}(\text{ast}), \text{get_children}(\text{ast})$ // 获取 ast 的 token 和 children
6. 如果 $id = 0$:
7. $\text{newtree.token} \leftarrow \text{token}$ // 赋值 token
8. $\text{newtree.data} \leftarrow \text{ast}$ // 赋值 data
9. 否则:

10. 构建新节点 $\text{newnode} \leftarrow \text{AnyNode}(\text{id}=\text{id}, \text{token}=\text{token}, \text{data}=\text{ast}, \text{parent}=\text{parent})$
 11. $\text{nodelist.append}(\text{ast})$ // 将 ast 添加到 nodelist
 12. 对于 children 集合中的每个 child:
 13. 如果 $id = 0$:
 14. 递归调用 $\text{createTree}(\text{newtree}, \text{child}, \text{nodelist}, \text{parent}=\text{newtree})$
 15. 否则:
 16. 递归调用 $\text{createTree}(\text{newtree}, \text{child}, \text{nodelist}, \text{parent}=\text{newnode})$
 17. 结束循环
 18. 返回 newtree
- 结束 createTree 方法

2.3 增强流边添加算法

在完成语法结构重构之后,本文进一步对程序表示进行语义增强,其核心目标是构建一个包含节点特征与多类型关系信息的统一图结构。最终得到的表示不仅包括节点的词汇索引集合,还包含边标识集合及其对应的属性集合,从而为后续图模型提供结构与语义兼备的输入。

Table 2 Edge attribute table

表2 边属性表

边属性	索引
Nextsib	1
NextToken	2
PrevToken	3
NextUse	4
PrevUse	5
If	6
Ifelse	7
While	8
For	9
Nextstmt	10
Prevstmt	11
Prevsib	12

随后,对重构语法树执行递归遍历。在遍历过程中,逐步完成节点与边信息的填充:一方面,根据词汇映射表为每个节点分配对应的词典索引,并将节点标识加入节点集合;另一方面,对已有的语法边赋予默认属性值(如用0表示基础语法关系),以便与后续引入的增强关系进行区分。该阶段的关键在于将原始树结构转化为包含初始特征的图表示。

在此基础上,进一步引入多种语义相关的连接关系,以增强图结构的表达能力。具体包括:用于刻画同层节点顺序关系的 Nextsib 连接、反映程序执行或依赖路径的 edgeflow 与 blockededge 关系,以及用于保留词法顺序信息和变量使用链的 Nexttoken 与 Nextuse 关系。在每一种新边引入时,均同步更新节点标识集合与边属性集合,确保图结构与特征表示的一致性。例如,在变量多次使用的场景中,通过 Nextuse 关系可将不同位置的变量节点连接起来,从而显式表达其潜在的数据流联系。

算法2 增强流边添加算法

输入: 重构语法树 newtree, 词典表 vocadict, 边属性字典 edges。

输出: 节点词典索引集合 nodeindexlist, 边源 id 集合 edge_src, 边目标 id 集合 edge_tgt, 边属性集合 edge_attr。

```

1. 初始化 nodeindexlist ← [], edge_src ← [], edge_tgt ← [], edge_attr ← []
2. 定义 getNodeAndEdge 方法, 参数为 newtree, nodeindexlist, vocadict, edge_src, edge_tgt, edge_attr
3. nodeindexlist.append([vocadict[newtree.token]]) // 根据 vocadict 填充节点词典索引
4. 对于 newtree 的子树集合中的每个 child:
5. edge_src.append(newtree.id) // 添加节点 id 到边源 id 集合
6. edge_tgt.append(child.id) // 添加子节点 id 到边目标 id 集合
7. edge_attr.append([0]) // 非增强流边属性设为 0
8. edge_src.append(child.id) // 添加子节点 id 到边源 id 集合
9. edge_tgt.append(newtree.id) // 添加节点 id 到边目标 id 集合
10. edge_attr.append([0]) // 非增强流边属性设为 0
11. 递归调用 getNodeAndEdge(child, nodeindexlist, vocadict, edge_src, edge_tgt, edge_attr)
12. 结束循环
13. 结束 getNodeAndEdge 方法
14. 如果 next_sib = True: // 添加 next_sib 边
15. 定义 getEdgeNextSib 方法, 参数为 newtree, vocadict, edge_src, edge_tgt, edge_attr
16. 对于 i 在 range(len(newtree.children)-1) 中:
17. edge_src.append(newtree.children[i].id) // 添加当前子节点 id
18. edge_tgt.append(newtree.children[i+1].id) // 添加下一个子节点 id
19. edge_attr.append([edges['Nextsib']]) // 根据边属性字典添加 Nextsib 边属性
20. edge_src.append(newtree.children[i+1].id) // 添加下一个子节点 id
21. edge_tgt.append(newtree.children[i].id) // 添加当前子节点 id
22. edge_attr.append([edges['Prevsib']]) // 添加 Prevsib 边属性
23. 结束循环
24. 对于 newtree 的子树集合中的每个 child:
25. 递归调用 getEdgeNextSib(child, vocadict, edge_src, edge_tgt, edge_attr)
26. 结束循环
27. 结束 getEdgeNextSib 方法
28. 如果 blockedge = True 或 nexttoken = True 或 nextuse = True 或 edgeflownext = True: // 添加其他类型边
29. 按照 getEdgeNextSib 方法类似方式添加其他类型边
30. 结束条件
返回 edge_src, edge_tgt, nodeindexlist, edge_attr 在算法 2 后, 分别得出节点的词典索引集合 nodeindexlist、边的

```

id 集合 edge_src 和 edgetgt, 以及边的属性集合 edge_attr。随后, 根据深度学习框架 Pytorch 中的 tensor() 方法进行张量化。其中, tensor() 方法的参数设置如下: 第 1 个参数接收集合数据作为输入, 第 2 个参数 dtype 用于指定张量的数据类型, 第 3 个参数 device 用于指定计算设备 (CPU 或 GPU), 最终将这些配置好的张量作为图神经网络模型的输入。

3 实验**3.1 实验数据**

本文选取了 Google Code Jam (GCJ) 数据集, 该数据集源于谷歌每年举办的在线编程大赛 (<https://codingcompetitions.withgoogle.com/codejam>), 由 2016 年 12 个不同竞赛问题的 1 669 个 Java 文件组成, 每个文件均为一个 Java 类。表 3 展示了数据集的基本信息, 其中, 平均节点数表示抽象语法树重构后的 AnyNode 节点数量。

Table 3 Basic information of the dataset**表3 数据集基本信息**

数据集名称	Google Code Jam
代码片段	1 669
平均代码行数	58.79
平均节点数	396.98
词汇量	8 033
重复的代码片段对	275 570
不重复的代码片段对	1 116 376

3.2 评价指标

实验中采用精确率 (precision)、召回率 (recall) 和 F-score 作为评价指标, 这 3 个指标是衡量性能的常用标准。其中, 精确率反映的是预测实例的精确程度, 即预测为正的样本中有多少是真正的正样本, 具体如公式 (2) 所示:

$$precision = \frac{TP}{TP + FP} \quad (2)$$

召回率反映了模型的敏感程度, 即正确的样本中有多少被预测为正确, 具体如公式 (3) 所示:

$$recall = \frac{TP}{TP + FN} \quad (3)$$

F-score 是精确率和召回率两者的调和平均数, 具体如公式 (4) 所示:

$$G-F-score = \frac{2 * precision * recall}{precision + recall} \quad (4)$$

3.3 实验结果与分析

在实验中, 数据集按 8:1:1 的比例划分为训练集、验证集和测试集, 具体信息如表 4 所示。

Table 4 Dataset statistics table**表4 数据集统计信息表**

数据集类型	Java 代码对
训练数据集	726 376
验证数据集	13 695
测试数据集	13 695

图神经网络的相似性代码模型在得到输入集合表示后,通过 `torch.tensor()` 方法进行张量化处理,并作为模型的输入。图神经网络模型参数 `num_layers` 设置为 4, `embedding_dim` 设置为 100。由于计算机性能限制,实验选用 CPU,损失函数选用均方误差损失 MSE 来训练模型,实验中通过 `trange()` 函数设置 10 个 epoch 来进行训练。实验结果如表 5 所示。

Table 5 Experimental results of similarity code detection based on graph neural network

表 5 基于图神经网络的相似性代码检测实验结果

模型	损失函数	精度	召回率	F1-Score
GMN	MSE	0.993 9	0.979 0	0.986 4
GGNN	MSE	0.786 3	0.998 1	0.965 2

由表 5 可知,在相同 MSE 损失函数条件下,图神经网络模型 GMN 与 GGNN 在召回率和 F1-Score 指标上表现相当,但 GMN 在精度指标上实现了 21% 的提升,整体性能表现更优。

4 结语

克隆代码是软件质量保证中不可或缺的一个环节,尽早发现克隆代码能够为开发人员的后期项目维护提供极大的便利,从而大幅提升工作效率。本文提出了一种基于图神经网络的代码相似性检测方法,具体采用图神经网络(GNN)进行代码克隆检测。该方法的实现主要包含以下 3 个步骤:首先,为程序创建图形表示;其次,使用图神经网络计算代码片段的向量表示;最后,通过度量代码向量表示的相似性来度量代码相似性。为了充分利用程序中的控制流和数据流信息,本文构建了一种基于抽象语法树(AST)的程序图表示。通过在 AST 中添加多种类型的边,以表示不同类型的控制和数据流,从而完成构建。在实验部分,本文采用门控图神经网络(GGNN)和图匹配网络(GMN)来学习代码片的特征向量,并在精度、召回率和 F1-Score 等指标上进行对比分析。

参考文献:

[1] QURADAA F H, SHAHZAD S, ALMOQ B R S. A systematic literature review on the applications of recurrent neural networks in code clone research[J]. PLOS ONE, 2024, 19(2): e0296858.

[2] SCHOTT S, PONTA S E, FISCHER W, et al. Java bytecode normalization for code similarity analysis[C]// ECOOP 2024: 38th European Conference on Object-Oriented Programming, 2024: 1-37, 29.

[3] GLANI Y, LUO P. CCDive: a deep dive into code clone detection using

local sequence alignment[J]. Tsinghua Science and Technology, 2025, 30(4): 1435-1456.

[4] GLANI Y, LUO P. CodeGuard: enhancing accuracy in detecting clones within Java source code[J]. Frontiers in Computer Science, 2024, 6: 1455860.

[5] NISHI M A, DAMEVSKI K. Scalable code clone detection and search based on adaptive prefix filtering[J]. Journal of Systems and Software, 2018, 137: 130-142.

[6] TEKCHANDANI R, BHATIA R, SINGH M. Code clone genealogy detection on e-health system using Hadoop[J]. Computers & Electrical Engineering, 2017, 61: 15-30.

[7] FARHADI M R, FUNG B C M, FUNG Y B, et al. Scalable code clone search for malware analysis[J]. Digital Investigation, 2015, 15: 46-60.

[8] YUKI Y, HIGO Y, KUSUMOTO S. A technique to detect multi-grained code clones [C]// 2017 IEEE 11th International Workshop on Software Clones (IWSC), 2017: 1-7.

[9] SAJNANI H, SAINI V, SVAJLENKO J, et al. SourcererCC: scaling code clone detection to big-code [C]// Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016: 1157-1168.

[10] LI C L, FENG H, ZHUANG W, et al. CCleaner: a deep learning-based clone detection approach [C]// 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017: 249-260.

[11] YANG Y, REN Z, CHEN X, et al. Structural function based code clone detection using a new hybrid technique [C]// 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), 2018: 286-291.

[12] PATI J, KUMAR B, MANJHI D, et al. A comparison among ARIMA, BP-NN, and MOGA-NN for software clone evolution prediction [J]. IEEE Access, 2017, 5: 11841-11851.

[13] SARGSYAN S, KURMANGALEEV S, BELEVANTSEV A, et al. Scalable and accurate detection of code clones[J]. Programming and Computer Software, 2016, 42(1): 27-33.

[14] RACKHITWETSAGUL C, KRINKE J, MARNETTE B. A picture is worth a thousand words: code clone detection based on image similarity [C]// 2018 IEEE 12th International Workshop on Software Clones (IWSC), 2018: 44-50.

[15] ZENG J, BEN K R, ZHANG X, et al. Code clone detection based on program vector tree [J]. Journal of Computer Science and Technology, 2020, 14(10): 1656-1669.

曾杰, 贵可荣, 张献, 等. 基于程序向量树的代码克隆检测[J]. 计算机科学与探索, 2020, 14(10): 1656-1669.

[16] ZENG J, BEN K R, ZHANG X, et al. API misuse defect detection based on sequence pattern matching[J]. Journal of Huazhong University of Science and Technology (Natural Science Edition), 2021, 49(2): 108-114.

曾杰, 贵可荣, 张献, 等. 基于序列模式匹配的 API 误用缺陷检测[J]. 华中科技大学学报(自然科学版), 2021, 49(2): 108-114.

(责任编辑:陈维捷)