



南京大學

研究生畢業論文  
(申請碩士學位)

論文題目 Pyreview: 一个基于抽象语法树差异  
提取的 Python 源代码分析工具

作者姓名 李清言

学科专业名称 计算机科学与技术


研究方向 软件分析

指导教师 徐宝文

2016 年 5 月 27 日

学 号：MG1333028

论文答辩日期：2016 年 5 月 27 日

指导教师： (签字)

# 南京大学硕士学位论文

## Pyreview：一个基于抽象语法树差异提取的Python源代码 分析工具



申 请 人：李清言

学 号：MG1333028

专 业：计算机技术

研究方向：软件分析

指导教师：徐宝文

2016 年 5 月



**Pyreview: A Python Source Code Analysis Tool Based on  
Abstract Syntax Tree Differencing Algorithm**

By

**Li Qingyan**

A thesis

Submitted to the faculty of graduate studies  
in partial fulfillment of the requirements for the degree of

**MASTER OF ENGINEERING**

In

Computer Technology

Supervised by

**Prof. Xu Baowen**

Department of Computer Science and Technology

Nanjing University

May, 2016

Nanjing, P. R. China

## 南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目: Pyreview: 一个基于抽象语法树差异提取的 Python

源代码分析工具

计算机技术 专业 2013 级硕士生姓名: 李清言

指导教师 (姓名、职称): 徐宝文 教授

### 摘要

在软件开发维护过程中, 代码差异分析是一个非常重要的内容。通过代码差异分析, 开发者可以更好地理解版本间代码的变更, 追踪软件的演化, 也可以检测克隆代码块, 进而做好软件的维护和后续的开发工作。Python 语言作为使用最为广泛的动态语言之一, 针对它的这方面的工作却极为匮乏。为此, 本文设计实现了 Pyreview, 一个基于抽象语法树差异提取的 Python 源代码分析工具。

本文的主要工作包括:

- 提出了改进的字符串距离计算和语法树结点列表匹配方法, 提高了已有的基于抽象语法树差异提取的代码变更分析方法的性能。对 GIT 仓库中的 9 个 Python 项目的实验分析表明, Pyreview 的差异代码分析结果的准确率和召回率都在 98% 以上。相比已有的算法, Pyreview 平均可以缩短 20% 的运行时间。
- 通过对 Python 语言中语法规则的分析, 分类总结了 Python 的代码变更类型, 并提出了相应的自动分类方法。Pyreview 中的代码变更类型分析模块可以有效应对 Python 语言中特殊的语法糖和动态特性的使用。
- 提出了一种针对 gapped clone code 的检测方法, 该方法在一定程度上对克隆代码块中语句的插入和删除操作免疫。

Pyreview 作为一个 Python 源代码分析工具, 可以应用在代码比较、代码变更模式提取、软件演化分析、克隆代码块检测等领域, 为相关的研究工作提供一种技术实现和数据获取方式。

**关键词:** 代码结构化比较, 代码克隆检测, 代码变更分类

## 南京大学研究生毕业论文英文摘要首页用纸

THESIS: Pyreview: A Python Source Code Changes Analysis Tool Based on  
Abstract Syntax Tree Differencing algorithm

SPECIALIZATION: Computer Technology

POSTGRADUATE: Li Qingyan

MENTOR: Prof. Xu Baowen

### **ABSTRACT**

A key issue in software analysis is the identification of particular changes that occur across several versions of a program. Information from the code changes can help developers better understand the software function points and the evolution process, thus to improve the maintenance job. Unfortunately, as far as we know, there are no robust tools developed for Python, focused on this area. For that reason, we present Pyreview, a Python source code analysis tool, based on abstract syntax tree differencing algorithm.

In this paper, our contributions are mainly as followed:

- Present a new way to calculate the distance of two strings and reduce the runtime complexity of nodes list permutation. The experiment shows that for 9 GIT repositories, Pyreview can find 1606 code changes and the precision and recall rate of the result are both higher than 98%. Besides that, Pyreview can averagely reduce the running time by 20%.
- Present a new set of rules for code change extraction based on Python syntax details, which can classify changes in Python source code into different change kinds automatically.
- Present a specific way for gapped clone code detection. The process can handle the situation when statements are inserted into or deleted from the code block.

As a Python source code analysis tool, Pyreview can be used in code change extraction, clone code detection, code change evolution analysis and some related areas. In the future, we want Pyreview to be a basic infrastructure in Python source code analysis area.

**Keywords:** code structural comparison, code change extraction, clone code detection

目录

摘要 ..... I

ABSTRACT ..... II

第一章 引言 ..... 1

    1.1 研究动因 ..... 1

    1.2 研究现状 ..... 2

    1.3 本文工作 ..... 3

    1.4 论文结构 ..... 3

第二章 基于抽象语法树的代码差异分析 ..... 1

    2.1 Pyreview 分析流程 ..... 1

    2.2 基础知识 ..... 2

        2.2.1 树的编辑脚本 ..... 2

        2.2.2 抽象语法树上结点的大小 ..... 3

    2.3 代码结构化比较功能的实现原理 ..... 4

        2.3.1 结点的相似度 ..... 5

        2.3.2 树的基本操作的表示 ..... 7

        2.3.2 结点序列的匹配 ..... 7

    2.4 针对 Psydiff 算法的优化 ..... 9

        2.4.1 移动结点的检测 ..... 9

        2.4.2 Bi-gram 字符串相似度算法的使用 ..... 11

        2.4.3 带缓存实现的结点列表匹配 ..... 14

    2.5 实验评估 ..... 16

        2.5.1 实验对象 ..... 16

        2.5.2 实验综述 ..... 18

        2.5.3 代码结构化比较的实验结果 ..... 18

        2.5.4 与 Psydiff 算法的比较 ..... 19

    2.6 本章小结 ..... 21

第三章 Pyreview 代码变更类型分析 ..... 22

    3.1 背景介绍 ..... 22

    3.2 变更类型分析功能的实现 ..... 23

        3.2.1 结点的角色和类型 ..... 23

        3.2.2 结点变更类型规则 ..... 24

        3.2.3 变更类型分析流程 ..... 26

    3.3 实验评估 ..... 27

    3.4 本章小结 ..... 28

第四章 Pyreview 代码克隆检测 ..... 29

    4.1 背景介绍 ..... 29

    4.2 现有工作的不足 ..... 29

    4.3 代码克隆检测功能的实现 ..... 31

        4.3.1 Anti-unification 算法 ..... 31

        4.3.2 Pyreview 中的实现算法 ..... 32

    4.4 实验评估 ..... 34



4.4.1 与 Clonedigger 实验数据的比较 .....36

4.5 本章总结.....37

第五章 总结与展望.....38

5.1 工作总结.....38

5.2 未来工作.....38

参考文献.....40

致谢 .....42

攻读硕士学位期间主要的研究成果.....43

# 第一章 引言

本章主要对本文的研究工作进行整体的介绍，首先介绍研究动因，然后描述相关的研究工作，最后列出本文主要关注的问题并给出文章结构。

## 1.1 研究动因

Python 是一门非常优秀的编程语言，其强大的表达能力，极大地提升了程序员的开发效率，但同时也增加了 Python 语言相关分析工具的开发难度。当前针对 Python 语言在源代码分析方面的工作极为匮乏，因此在本文中我们开发了 Pyreview，为后续的 Python 源代码分析工作提供了一种技术实现。

在软件开发生命周期中，代码变更是一个非常重要的过程。从代码变更中获取到的信息可以帮助开发者更好地理解软件的功能和代码演进过程，进而做好软件的维护和后续的持续开发工作。同时，重复代码的检测，也可以使程序员更早地发现可能存在缺陷的代码，当重复代码需要进行重构时，可以大大降低程序后期维护的成本。

现在的工具如使用版本控制系统我们可以追踪到特定文件源代码变化的完整过程，但版本控制系统只是以文本的形式对待程序源代码，通过代码行的增加或删除来简单表示源代码的具体变化。这种方法的精确度很低，也完全无法展现源代码在结构组织上的变化。

之前软件演进分析领域的很多研究受限于无法获得高质量和高精度的代码变更信息而欠缺了一定的说服力。在本篇论文中，我们首先将程序的源代码转化成等价的抽象语法树形式，通过比较树的差异来获得细粒度级的代码结点的变化。一方面，相对之前的工作，我们优化了抽象语法树结点的差异比较算法，试图在效率和精确度上做到更好的平衡；另一方面除了可以很好的替代 Unix 系统下的 diff 工具或者作为版本控制系统的默认代码比较工具外，Pyreview 还提供了代码变更类型分析和代码克隆检测功能。

本文中的工具 Pyreview 实现了代码结构化比较、代码变更分类、代码克隆检测这三种功能，丰富了 Python 源代码分析工具的生态圈，为后续更多和 Python 源代码分析相关的研究提供了一种技术实现和数据获取方式。

## 1.2 研究现状

从 80 年代起,人们就将代码变更视为软件生命周期的一个重要部分,Lehman 等人的研究表明软件开发需要随着需求和运行环境的改变而不断演进,否则程序会逐渐丧失时效性[1]。近些年人们也研究开发了很多工具和技术来辅助开发者更好地进行大型复杂软件的维护和后续开发工作。如 Ying 和 Zimmermann 的工作着眼于代码的协同变更,可以告诉使用者,程序员在更改程序的一个函数时同时也更改了哪些部分[2][3]。Hipikat 工具通过对项目开发历史的分析,可以对项目的维护工作推荐更合适的候选开发者[4]。Gall 通过分析程序模块间的协同改变来发现潜在的程序维护热点[5]。

上面的工作很有意义,但受限于实验数据中并没有高可靠性高精度的代码变更信息,代码的变更信息只是精确到了文件或函数、类的级别。获取代码的变更信息(此处单指源代码之间的差异信息),现在主要有三种方法:

### (1) 将源代码视为文本数据,运用字符串比对算法

以单个字符为基本单位将源代码视为文本数据,在代码行的级别上进行代码差异的比较,Unix 系统下的 diff 工具便是基于这种原理。主要的比较算法有 Baker 算法[6]、Rabin-Karp 字符串搜索算法[7]。此种方案的优点在于程序的普适性高,无需为特定语言进行适配工作;算法的时间复杂度低,可以应用于大规模的代码比较。但同时缺点也很明显,最显著的两点是基于代码行级别的差异比较粒度太大、基于文本的比较无法应对代码结构上的变化。

### (2) 基于抽象语法树的代码差异比较

在计算机科学中,抽象语法树是源代码的抽象语法结构的树状表现形式,树上的每个结点都表示源代码中的一种结构。在源代码的解析过程中,抽象语法树在语法分析阶段创建,虽然抽象语法树不会表示出真实语法中出现的每一个细节,比如嵌套括号被隐含在树的结构中,并没有以结点的形式呈现,但抽象语法树是与相应源代码等价的表现形式。CHANGEDISTILLER 是一个针对 Java 源代码的变更分类工具,最初发表在论文[8]中,其核心算法为 Chawathe algorithm[9],即是提取语法树上结点的差异。Clonedigger[10]是一个针对 Python 源代码的克隆检测工具,它在实现中使用了 anti-unification 算法[11][12],同样是在抽象语法树的级别进行检测。Psydiff[13]是一个已有的代码结构化比较算法,Pyreview 中代码

结构化比较功能的实现也是基于 Psydiff 算法。

### (3) 矩阵计数检测

矩阵计数检测方法是受到搜索领域的向量空间模型的启发，通过特定规则将源代码映射到一个向量空间模型中，通过比较两个向量空间的差异进而得到源代码间的差异[14][15]。矩阵计数检测方法的出现使大规模的代码比较有了可能，但矩阵计数检测只能得到一个近似的结果，更适用于代码克隆检测和代码功能分类领域。

## 1.3 本文工作

源代码的变更分析可以帮助开发者更好的理解代码意图，更有利于维护工作的进展。传统的源代码比较方法，以 Unix 系统下的 diff 工具[16]为例，将源代码视为文本数据，只能以代码行的增加和删除来表示具体的变更。这方面，基于抽象语法树差异提取的算法可以实现词法级别的比较。相对于之前已经存在的工作，Pyreview 提升了 Psydiff 算法在字符串字面量比较和结点序列匹配方面的表现，在程序的运行效率和结果的精确度上做到了更好的平衡。同时，本文中的工具 Pyreview 实现了代码变更分类和代码克隆检测功能。本文的工作是对现有 Python 源代码分析领域的一种补充。代码变更分类功能可以帮助开发者针对特定的 Python 代码变更模式进行研究；在代码克隆检测方面也克服了 Clonedigger 无法应对的 gapped clone[41]问题。本文中的 benchmark 表明 Pyreview 的结果具有很高的精确度，可以应用在工业或学术界的相关研究领域。

## 1.4 论文结构

论文结构安排如下：第一章概述本文的主要工作，并且分析当前的研究现状；第二章主要描述了 Pyreview 的基本功能即代码结构化比较的实现过程；第三章中我们通过对代码结构化比较结果的分析，实现了代码变更分类功能，并进行了相关的实验验证；第四章我们实现了 Pyreview 中的代码克隆检测功能，并与 Clonedigger 的实现进行了比较；第五章是对本文的工作进行总结和展望。

## 第二章 基于抽象语法树的代码差异分析

### 2.1 Pyreview 分析流程

Pyreview 是一个基于抽象语法树差异提取的 Python 源代码分析工具，其核心功能是实现了源代码的结构化比较。在对代码结构化比较结果的分析基础上，Pyreview 进一步提供了代码变更类型分析和代码克隆检测功能。源代码的结构化比较是在抽象语法树级别上抽取代码的异同结点，可以在词法级别显示代码的变化。代码变更类型分析可以将代码中的改变映射到具体的变更类型，进而可以量化地分析代码的演进过程。代码克隆检测功能可以检测出软件中存在的重复代码，辅助开发者进行软件的重构和维护工作。Pyreview 的分析流程如图 2-1 所示。

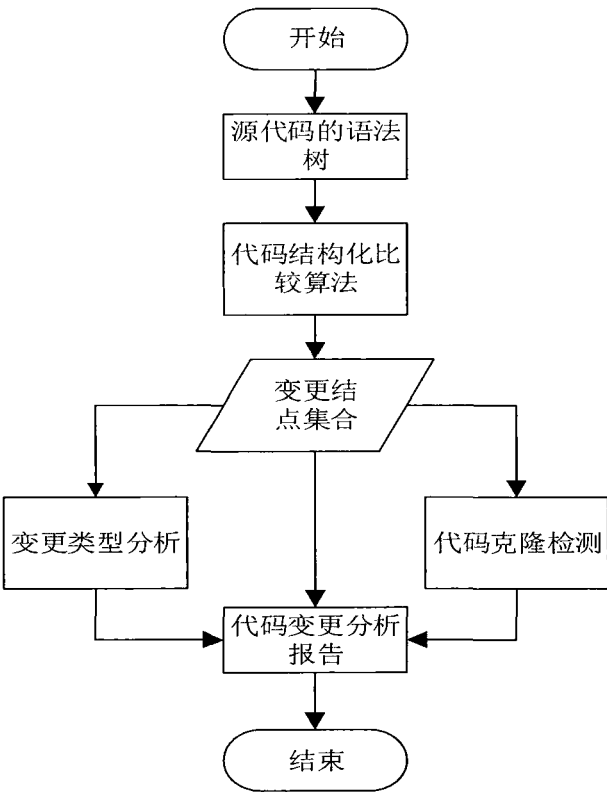


图 2-1 Pyreview 的分析流程

Pyreview 中代码结构化比较功能是基于 Psydifff 算法[13]实现。在本章中我们将重点介绍代码结构化比较功能的实现原理和细节，并通过具体的实验来验证 Preview 分析结果的有效性和正确性。

## 2.2 基础知识

### 2.2.1 树的编辑脚本

代码结构化比较的效果依赖于具体的抽象语法树上的差异提取方法，为了精确得到语法树 T1 和 T2 之间的差异结点，我们需要做到：

- 1) 通过高效精确的算法尽可能多地匹配 T1 和 T2 上的结点
- 2) 根据第一步中得到的 T1 和 T2 上的结点的匹配集合，求接出一个近似的 T1 和 T2 之间的最小距离编辑脚本。

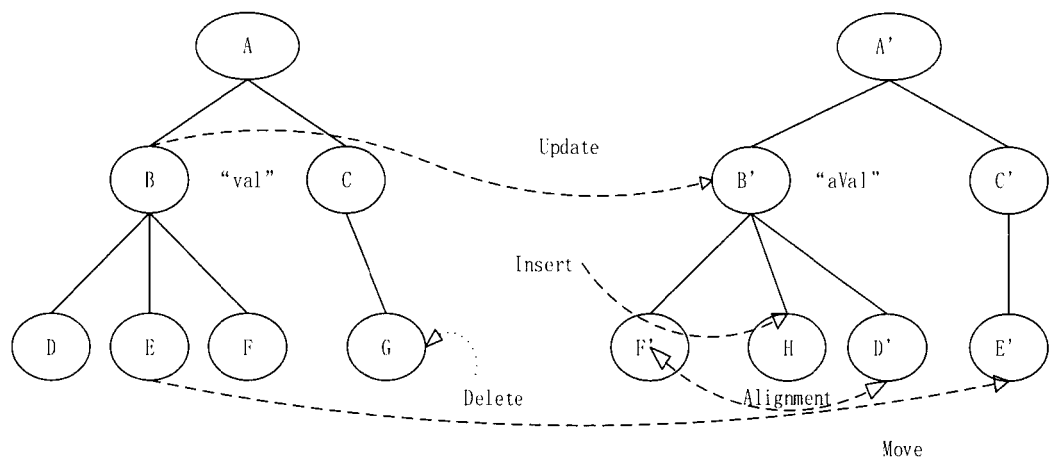


图 2-2 五种树的基本操作示例图

树的最小编辑距离[17]是指由一棵树转换成另一棵树所需的最少的编辑操作次数。如图 2-2 所示，在树的转换过程中被允许的编辑操作包括：

- 1) 插入一个结点： $INS((l,v),y,k)$ ，在  $y$  结点下插入一个标签为  $l$ ，值为  $v$  的结点，作为  $y$  的第  $k$  个孩子结点。在图 2-2 中，结点  $H$  就是作为结点  $B'$  的第二个孩子结点被插入到 T2 中： $INS((H,""),B',2)$ 。

- 2) 删除一个结点： $DEL(x)$ ，在树中删除结点  $x$ 。在图 2-2 中，结点  $G$  就在 T1 中被删除： $DEL(G)$ 。

- 3) 移动一个结点： $MOV(x,y,k),p(x) \neq y$ ，结点  $x$  成为结点  $y$  的第  $k$  个孩子结点，并且不再是  $p(x)$  的孩子结点。在图 2-2 中结点  $E$  从 T1 移到 T2，父结点也由  $B$  变成了  $C'$ ： $MOVE(E,C',1)$ 。

- 4) 更新一个结点： $UPD(x,val)$ ，更新结点  $x$  的值为  $val$ ，其中  $val = v_{new}(x)$  且  $v_{old}(x) \neq v_{new}(x)$ 。在图 2-2 中结点  $B$  的值由 "val" 变成了 "aval"： $UPD(B,"aval")$ 。

5) Alignment:  $MOV(x, p(x), k)$ , 结点  $x$  成为  $x$  的父结点  $p(x)$  的第  $k$  个孩子结点。在图 2-2 中, 结点  $F'$  成为了结点  $B'$  的第一个孩子结点:  $MOV(D, B', 3)$ 。

根据上面关于树的基本操作和最小编辑距离的定义, 我们也可以得出从  $T1$  转换到  $T2$  的最小编辑脚本为:

```
INS((H, " "), B', 2);  
DEL(G);  
MOVE( E, C', 1);  
UPD(B, "aval");  
MOV(D, B', 3);
```

2.2.2 抽象语法树上结点的大小

抽象语法树结点大小的定义为:

一个结点以及在抽象语法树上该结点的子树中所包含的终结符结点的个数。

在抽象语法树的具体表示中, 终结符结点并不是指叶结点。终结符出现在编程语言的 BNF 范式中, 是编程语言语法规则中不能拆分的最小单元。而抽象语法树的叶结点实际上是编程语言中的基本类型, 如布尔型, 整数型变量等。叶结点集合是终结符结点集合的一个子集。结点大小的计算包括下面两步:

- 1) 对抽象语法树中的结点进行分类, 区分出终结符结点和非终结符结点。
- 2) 针对不同类型的结点, 定义具体的计数规则。

在第 1) 步中, 我们将抽象语法树中的结点, 划分成了五种类别, 分别是原子结点、标识符结点、数值结点、封装的字符串结点和其他的继承自 AST 的结点。具体的划分类型和说明在表 2-1 中:

表 2-1 抽象语法树结点划分

atom node	int, string, bool, float
Name node	语法树中的标识符
Number node	语法树中封装的数值对象
String node	语法树中封装的字符串对象
Other AST node	泛指除上述对象外, 在语法树出现的其他类型的结点

在得到结点的划分后, 就可以针对不同的类型, 定义具体的计数规则, 最终

采用的计算公式是：

$$node-size(n)=\begin{cases} 1, n\text{ is atom node} & rule1 \\ 1, n\text{ is Name node} & rule2 \\ 1, n\text{ is Number node} & rule3 \\ 1, n\text{ is String node} & rule4 \\ 1+map(node-size(n.children)), n\text{ is the rest kind of AST nodes} & rule5 \end{cases}$$

在上面的公式中，我们将 atom node、Name node、Number node、String node 都视为最小的语法规则单元处理，计数是 1。针对其他的继承自 AST node 的结点，我们将它们当作复合的结点来处理，其 size 属性的值为该结点的所有子结点的 size 值的和加 1。

以语句result = 2 \* val为例，设表示整个语句的语法树结点为 n，下面来说明结点 n 的 size 值的计算过程。

在抽象语法树中，n 是一个赋值结点（Assign Node），我们首先要根据 rule5 进行计算。结点 n 的 JSON 格式表示为{expression:{type: AssignmentExpression, operator: =, target: {type: Name, id: result}, value:{type: BinaryExpression, operator: \*, left: {type: Number, value: 2}, right:{type: Name, id: val}}},}}。对于结点 n，有：

1+node-size(operator)+node-size(target)+node-size(value) #rule5

↓

1+1+ node-size(target) + node-size(value) #rule1

↓

2 + 1 + node-size(value) #rule2

↓

3+1+node-size(operator)+node-size(left)+node-size(right) #rule5

↓

4+1+node-size(left)+node-size(right) #rule1

↓

5+1+node-size(right) #rule3

↓

6+1 = 7 #rule2

最终求得语句result = 2 \* val的 size 值为 7。

2.3 代码结构化比较功能的实现原理

Pyreview 中的代码结构化比较模块可以抽取结点的插入、删除和更新这三种



树的基本操作，并根据基本操作序列得到两颗树之间的近似的具有最小代价的编辑脚本。代码结构化比较功能的分析过程如图 2-3 所示。

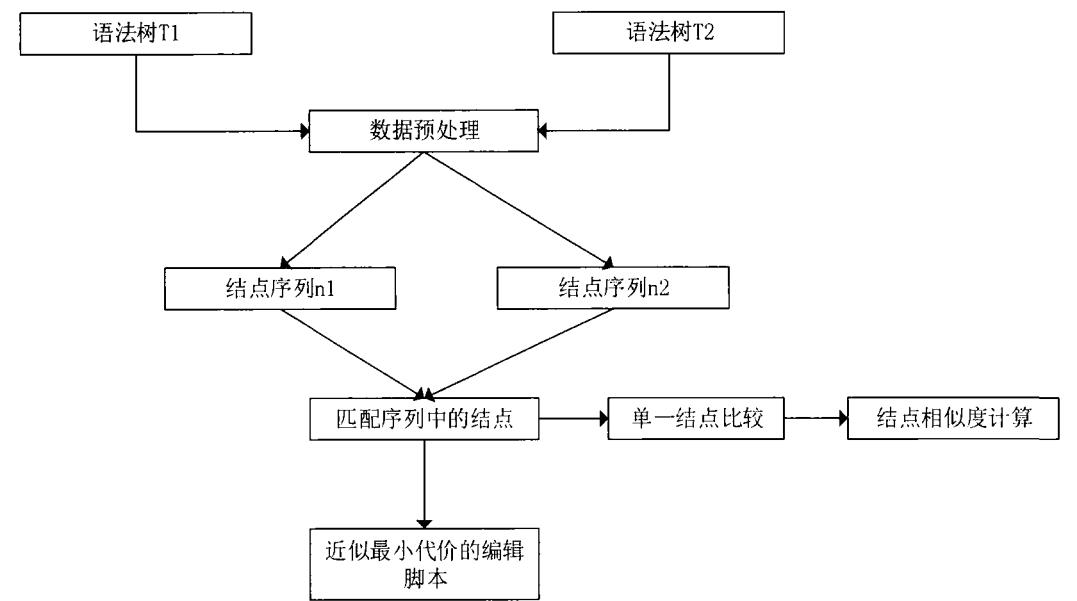


图 2-3 Pyreview 中代码结构化比较功能的分析流程

代码结构化比较模块的输入是两个版本的源代码的抽象语法树形式。在代码结构化比较的实现中，我们将具体的语法树作为嵌套的结点序列处理，在预处理阶段完成语法树到结点序列的转换。代码结构化比较的核心步骤是针对上面的结点序列计算出一个最优的结点间的匹配集合，其中语法树上的结点是根据彼此之间的相似度来决定是否可以匹配。Pyreview 最后的分析结果是两颗抽象语法树之间近似具有最小代价的编辑脚本。在下面的子章节中，我们会介绍 Pyreview 中代码结构化比较功能实现中的原理和细节。

2.3.1 结点的相似度

设比较中的两个抽象语法树结点为 node1 和 node2，Pyreview 中结点相似度计算函数  $\text{sim}(\text{node1}, \text{node2})$  的定义为：

$$\text{sim}(\text{node1}, \text{node2}) = 1 - \frac{\text{cost}(\text{node1}, \text{node2})}{\text{node} - \text{size}(\text{node1}) + \text{node} - \text{size}(\text{node2})}$$

相似度值介于 0 到 1 之间，相似度越大，两个结点越一致，相似度为 1 时，则认为两个结点相同。相似度 $\text{sim}(\text{node1}, \text{node2})$ 的计算一方面依赖于结点之间进行转换的最小代价值，另一方面依赖于比较结点的 size（即大小）值。

2.2.2 节中，我们已经知道了语法树结点大小的计算方法和过程，接下来是对

结点之间进行转换的最小代价值的 `cost(node1, node2)`函数进行介绍。

2.3.1.1 结点转换的代价

假设 `node1` 和 `node2` 是在比较中的两个结点，那么`cost(node1,node2)`函数是计算由结点 `node1` 转换到结点 `node2` 所需要的最小代价值。为了得到结点 `node1` 和 `node2` 之间的 `cost` 值，我们要分别对待下面两种情况：

- 1) `node1` 和 `node2` 是同种类型的结点时；
- 2) `node1` 和 `node2` 的类型不相同时。

第一种情况，当 `node1` 和 `node2` 的类型相同时，我们还要区分 `node1` 和 `node2` 是同为终结符结点还是内部结点。当 `node1` 和 `node2` 同为终结符结点时，图 2-4 给出了 `cost` 函数具体的计算流程。

```
1: Input: node node1, node2
2: Result: cost value changed from node1 to node2
3: if node1 == node2    // node1 和 node2 在内存中是同一结点
4:     cost(node1,node2) = 0
5: else if node1 and node2 are Number node //比较 value 属性，相同为 0，不同为 1
6:     if node1.value == node2.value
7:         cost(node1, node2) = 0
8:     else
9:         cost(node1,node2) = 1
10: else if node1 and node2 are String node    //比较 value 属性，计算字符串的距离
11:     cost(node1, node2) = strdist(node1.value, node2.value)
12: else if node1 is Name node and node2 is Name node //比较 id 属性，计算字符串的距离
13:     cost(node1, node2) = strdist(node1.id, node2.id)
```

图 2-4 `node1` 和 `node2` 同为终结符结点时，`cost(node1, node2)`函数的计算流程

在图 2-4 中，`cost` 函数首先判断 `node1` 和 `node2` 在内存中是不是同一结点，若为同一结点，`cost(node1,node2)`的值为 0；接着 `cost` 函数根据 `node1` 和 `node2` 具体的类型，来求解 `cost` 的值。当 `node1` 和 `node2` 是封装的数值类型时，查看它们实际存储的数字是否相等，若相等，则 `cost(node1,node2)`的值为 0，不相等则为 1；当 `node1` 和 `node2` 是封装的字符串类型时，`cost(node1,node2)`为实际存储的字符串字面量之间的距离；当 `node1` 和 `node2` 是标识符结点时，`cost(node1,node2)`为标识符的字符串字面量形式之间的距离。

当 `node1` 和 `node2` 同为非终结符结点时，`cost` 函数的计算方法如图 2-5 所示：

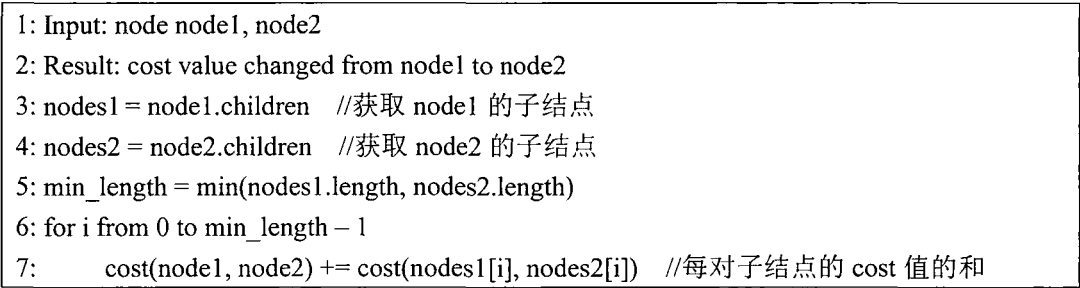


图 2-5 node1 和 node2 同为非终结符结点时，cost(node1, node2)函数的计算流程

当 node1 和 node2 类型相同且为非终结符结点时，cost 函数将 node1 和 node2 当作复合结点处理，cost(node1,node2)为 node1 和 node2 共有的孩子结点的 cost 值的求和。

第二种情况，当 node1 和 node2 的类型不相同时，cost 函数会返回 node-size(node1) + node-size(node2)的值（即 node1 结点和 node2 结点大小的和）作为 cost(node1,node2)的结果。

2.3.2 树的基本操作的表示

Pyreview 中的代码结构化比较模块可以抽取树的三种基本操作即结点的插入、删除和更新，当获得结点间进行转换的 cost 值后，我们就可以使用抽象的形式来表示这三种树的基本操作。假设比较的两个语法树结点为 node1 和 node2：

当 node1 和 node2 是相同类型的结点时，结点的更新操作可表示为：

$$\text{modNode}(\text{node1}, \text{node2}, \text{cost}) \rightarrow (\text{node1}, \text{node2}, \text{cost}(\text{node1}, \text{node2}))$$

当 node1 和 node2 不是同类型的结点时，结点的删除操作可表示为：

$$\text{delNode}(\text{node1}) \rightarrow (\text{node1}, \text{NULL}, \text{node} - \text{size}(\text{node1}))$$

结点的插入操作可表示为：

$$\text{insNode}(\text{node2}) \rightarrow (\text{NULL}, \text{node2}, \text{node} - \text{size}(\text{node2}))$$

2.3.3 结点序列的匹配

代码结构化比较功能的核心在于获得最优的具有最小转换价值的结点序列匹配，进而计算出近似具有最小代价的编辑脚本。下文中使用 diff-node(node1, node2)函数作为结点序列进行比较的入口函数。

diff-node 函数返回两个待比较语法树结点彼此进行转换需要的树的基本操作

序列。设比较中的两个结点是 node1 和 node2，当 node1 和 node2 类型相同，且都是终结符结点时，diff-node(node1, node2) 返回 modNode(node1,node2, cost(node1,node2)); 当 node1 和 node2 类型不同时，diff-node(node1, node2)返回 [delNode(node1), insNode(node2)]; 当 node1 和 node2 类型相同，且是非终结符结点时，基本操作序列集合的计算方式如图 2-6 所示：

```
nodes1 = node1.children
nodes2 = node2.children
changes = []
length = min(nodes1.length, nodes2.length)
for i in range(length)
    c = diff-node(nodes1[i], nodes2[i])
    changes += c
return changes
```

图 2-6 node1 和 node2 类型相同，且都是非终结符结点时，结点序列的匹配

即当 node1 和 node2 都是非终结符结点时，diff-node(node1,node2)的返回值 为 node1 和 node2 共有的孩子结点之间需要进行的树的基本操作序列的并集。

```
if node1 is empty and node2 is empty    //node1 和 node2 都为空，返回值也是空
    return []
else if node1 is empty                  //node1 为空时，node2 中的每个结点都是插入结点
    change = []
    for n in node2
        change += [insNode(n)]
    return change
else if node2 is empty                  //node2 为空时，node1 中的每个结点都是删除结点
    change = []
    for n in node1
        change += [delNode(n)]
else                                    //递归比较 node1 和 node2 中的结点，取代价最小的组合方式
    c0 = diff-node(node1[0], node2[0])
    c1 = diff-list(node1[1:], node2[1:])
    change1 = c0 + c1
    c2 = diff-list(node1[1:], node2)
    c3 = diff-list(node1, node2[1:])
    change2 = c2 + delNode(node1[0])
    change3 = c3 + insNode(node2[0])
    change = minCost(change1, change2, change3)
    return change
```

图 2-7 diff-list(node1,node2)函数实现伪代码

当在比较的两个结点中出现列表类型时，算法首先确保比较中的两个对象都是列表类型，如有必要可以将结点放入列表中，强制转换成列表类型。然后在 `diff-node` 函数中会调用 `diff-list(node1,node2)`函数。`diff-list(node1,node2)`函数的功能是进行列表之间结点的比较，其伪代码在图 2-7 中。

当 `node1` 和 `node2` 都是空列表时，函数自然会返回空的基本操作序列。当 `node1` 为空时，`node2` 则是现在版本中新增的代码块。我们遍历 `node2` 中的结点，为每个结点生成一个用 `insNode(node)`来表示的结点的插入操作。当 `node2` 为空时，表明 `node1` 只存在于原始版本的源代码中，即在现在的版本中，这是一次代码块的删除操作。我们遍历 `node1` 中的结点，为每个结点生成一个用 `delNode(node)`来表示的结点的删除操作。而当 `node1` 和 `node2` 都不是空列表时，我们会对 `node1` 和 `node2` 中的结点按不同的组合情况进行比较，取其中 `cost` 值最小的结点匹配组合作为最终的基本操作序列集合的返回值。

## 2.4 针对 Psydiff 算法的优化

Pyreview 中代码结构化比较功能的实现是基于 `psydiff` 算法。在本节中，我们将介绍 Pyreview 使用的代码结构化比较算法相对原始 `Psydiff` 算法所做的优化和改进。

### 2.4.1 移动结点的检测

`Psydiff` 算法只能抽取语法树上结点的插入、删除、和更改操作，对结点的移动操作没有实现检测。`Psydiff` 算法中结点列表的匹配是一个自上而下的递归比较过程，在某些情况下，因为 `Psydiff` 算法不能检测出结点的位置移动，进而会错误得将结点的变更归类为树的插入和删除操作。

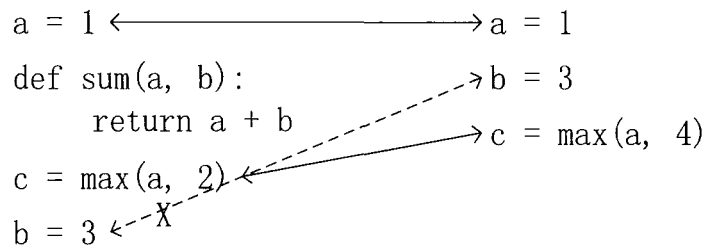


图 2-8 Psydiff 算法在有些情况下无法检测结点的移动

针对图 2-8 中进行比较的代码片段，可以看到 Psydiff 算法的比较结果中，将在两处代码中都出现过的  $b = 3$  语句视作一次语法树上的删除操作和一次插入操作。而实际上，这更应该是语法树上结点的一次移动操作。

在 Pyreview 中，我们扩展了 Psydiff 算法实现了对结点的移动操作的检测。在 2.3.3 小节对结点序列的匹配的介绍中，我们知道当 `node1` 和 `node2` 类型相同，且都是非终结符结点时，我们的算法会返回一个树的基本操作的集合，即为 `node1` 和 `node2` 共有的孩子结点之间相互转换需要进行的树的基本操作集合的并集。在 Pyreview 中我们对 `node1` 和 `node2` 作用域范围内的子结点中代码块顺序的变化作了特殊处理，可以匹配源代码中相同或相似但出现顺序不同的语法树结点，具体的算法如下：

```
1: Input: original changes set as old_changes
2: Result: new changes set as new_changes
3: deletions ← ∅, insertions ← ∅
4: matched ← ∅, new_changes ← ∅

5: for all change object c ∈ old_changes do
6:   if is_delete_node(c)
7:     deletions ← c
8:   if is_insert_node(c)
9:     insertions ← c

10: for node d0 ∈ deletions
11:   for node i0 ∈ insertions
12:     if same_type(d0,i0)
13:       changes, cost = diff-node(node1, node2)
14:       if cost < p
15:         matched ← d0, i0
16:         deletions.remove(d0)
17:         insertions.remove(i0)
18:         new_changes ← changes

19: old_changes = {c, c ∈ old_changes and c not in matched}
20: new_changes ← old_changes
21: return new_changes
```

图 2-9 移动结点匹配算法

根据 2.3.2 小节的描述, 在 `diff-node` 函数的实现中, 假设在比较中的两个结点为 `n1, n2`, 当两者类型不同时, 我们就将`[delNode(node1), insNode(node2)]`作为返回结果。同时, 在 `diff-list` 函数中, 我们会调用 `diff-node` 函数, 因此 `diff-list` 函数的返回结果中会有一部分被认定为删除或增加的结点。这些结点都在同一个父结点的作用域内, `move detect` 算法的作用就是发现在这些结点中是否存在相同或相似 (即转换代价值小于阈值的) 的结点匹配组合。

`move detect` 算法将原始的树的基本操作集合 (`old_changes`) 作为输入。在第 5 行到第 9 行, 我们首先获取 `change` 集合中, 删除的结点和增加的结点对应的集合。设在一次树的基本操作中比较的源结点为 `origin`, 当前结点为 `current`, 若 (1) `current` 为空, `origin` 不为空, 这是一个结点的删除操作, 我们将 `origin` 放入 `deletions` 集合; (2) `origin` 为空, `current` 不为空, 这是一个结点的增加操作, 我们将 `current` 放入 `insertions` 集合。在第 10 行到 17 行, 我们用一个双层循环遍历 `deletions` 和 `insertions`。若结点 `d0, i0` 是相同定义的结点, 则先将它们都放入 `matched` 列表中, 接着从 `deletions` 和 `insertions` 集合中删除。计算出的 `d0` 和 `i0` 之间的变更结点也要放入 `new_changs` 集合中。接着在第 18、19 行, 我们只保留 `old_changes` 集合中没有在 `matched` 中出现的基本操作。将结果过滤后的 `old_changes` 集合与新生成的 `new_changes` 集合作并运算, 则得到了最终的结果。

## 2.4.2 Bi-gram 字符串相似度算法的使用

为了获取两个结点之间进行转换的代价, 当结点的值表现为字符串字面量时, 我们需要计算出两个字符串之间的距离。

在 `Psydiff` 算法中使用的是 `Levenshtein String Similarity Measure`[19]的变种方法。`Levenshtein Distance`[19], 又称编辑距离 (`Edit Distance`), 是指两个字符串之间, 由一个转换成另一个所需的最少编辑操作次数。被允许的编辑操作包括 1) 插入一个字符, 2) 删除一个字符, 3) 将一个字符替换成另一个字符。编辑距离衍生自最长公共子串问题。一般来说, 编辑距离越小, 两个字符串的相似度越大, 距离为 0 时, 则指明两个字符串相等。编辑距离算法的时间复杂度是  $O(n \cdot m)$ 。若比较的两个字符串是  $S_a$  和  $S_b$ ,  $n$  指  $S_a$  中字符的个数,  $m$  是  $S_b$  中字符的个数。

在 `Psydiff` 算法中为了计算两个字符串之间进行转换的代价值, 首先要求

出字符串的距离。设在比较的两个字符串为  $s_1$  和  $s_2$ ，具体的距离计算公式为：

$$dist(s_1,s_2)=\begin{cases} s_2.length,s_1=="" \\ s_1.length,s_2=="" \\ rule3 \end{cases}$$

当  $s_1$  和  $s_2$  均不为空时算法使用上文的 rule3，针对比较中的两个字符串  $s_1$  和  $s_2$  的首个字符，具体的计算规则如下：

$$\begin{cases} d_0=0,s_1[0]==s_2[0] \\ d_0=1,s_1[0].lower==s_2[0].lower \\ d_0=2,s_1[0]\neq s_2[0] \end{cases}$$

将同样的规则应用于字符串中的所有字符，算法会得到三个距离候选值：

$$\begin{cases} d_0=d_0+dist(s_1[1:],s_2[1:]) \\ d_1=1+dist(s_1[1:],s_2) \\ d_2=1+dist(s_1,s_2[1:]) \end{cases}$$

那么 Rule3 得到的距离值  $d$  为  $\text{Min}(d_0,d_1,d_2)$ ，根据上文的计算，我们最终得到的距离  $d$  的值介于 0 到  $s_1.length + s_2.length$  之间。

算法的目的是计算两个字符串之间转换的代价值，距离值本身并不是一个有效的度量指标。除了编辑距离值的大小以外，我们还需要将比较的字符串的长度计算在内。为此 Psydiff 算法中使用下面的公式来完成距离  $d$  到代价  $\text{cost}$  的转换：

$$\text{cost}(s_1,s_2)=\frac{2*d}{s_1.length+s_2.length} \quad 0\leq \text{cost}\leq 2$$

Levenshtein Distance 或者 Psydiff 算法中使用的基于它的变种算法[20]都不是线性复杂度的算法，当比较的字符串的长度超过一定的数值时，程序会有明显的性能瓶颈。同时，编辑距离算法衍生自最长公共子串问题[21]，在设计之初，就不能对字符串中字符顺序的变化免疫。例如两个字符串  $s_1 = \text{verticalDrawAction}$ ， $s_2 = \text{drawVerticalAction}$ ，在相邻版本的源代码同一结点中发现这种变化时，我们有充分的理由认为这是一次在不影响程序功能下进行的代码重构。Levenshtein Distance 并不能很好处理这种变化。在下面的计算过程中，字符串  $s_1$  和  $s_2$  的最长公共子串为“verticalAction”，剩余的字符在转换过程中需要四次插入操作和四次删除操作，所以 Levenshtein Distance 得到的编辑距离是 8。

上文示例中的字符串变化在代码重构的过程中经常出现，因此在本文 Pyreview 的实现中我们希望可以寻找到一个字符串比较算法，在降低算法复杂度



的同时，能更好地处理重构过程中单词序列的变化。

在计算两个字符串的相似度时，大部分的算法都是基于 Dice Coefficient（相似系数）理论[22]。Dice Coefficient 是 Jaccard Coefficient[23]的一个变种。Adamson 和 Boreham 的工作通过分析待比较字符串中 n-grams 组合的异同来评估字符串之间的相似性[24]。N-grams 是一个集合，里面包括了一个字符串中所有的相邻且长度为 n 的子串，如 vertical 的 3-grams 为：

$$3 - \text{grams}(\text{vertical}) = \{ \text{"ver"}, \text{"ert"}, \text{"rti"}, \text{"tic"}, \text{"ica"}, \text{"cal"} \}$$

Adamson 和 Boreham 提出的根据 n-gram 相似度计算字符串相似度的公式为：

$$\text{sim}_{ng}(s_a, s_b) = \frac{2 * |n - \text{grams}(s_a) \cap n - \text{grams}(s_b)|}{|n - \text{grams}(s_a) \cup n - \text{grams}(s_b)|}$$

在实际使用中，开发者倾向于使用 bi-grams 和 tri-grams。先前的研究中，Xing 和 Stroulia 在论文[25]中使用了 bi-grams，Weidl 和 Gall 在论文[26]中使用了 tri-grams。

本文发现 n-gram 相似度可以更好的用于程序源代码变更分析。同样针对前文中的字符串  $s_1 = \text{verticalDrawAction}$  和  $s_2 = \text{drawVerticalAction}$ . 计算出的 bi, tri, four-grams 的相似度分别为：

$$\begin{aligned} \text{sim}_{2g}(s_1, s_2) &= \frac{2 * 13}{34} \approx 0.76 \\ \text{sim}_{3g}(s_1, s_2) &= \frac{2 * 11}{32} \approx 0.65 \\ \text{sim}_{4g}(s_1, s_2) &= \frac{2 * 9}{30} \approx 0.53 \end{aligned}$$

可以看出 n-gram 相似度在上面的测试案例中可以得到更好的结果。此外，重要的一点是 n-gram 相似度算法在实现时可以用 set 集合存储一个字符串产生的所有 n-grams 的子字符串，而另一个字符串在遍历产生子字符串时，为了求得交集只需要查看 set 集合中是否已经存在相同的字符串。综上，n-gram 相似度的算法是  $O(n + m)$ ，只需一次遍历就可以求出结果，性能上相比 Levenshtein Distance 有极大提升。

n-gram 相似度算法可以更好地应对字符串中序列的变化。n-gram 相似度算法

在实现中并不依赖字符串的最长公共子串，两个字符串中共有的字符会对结果产生更大的影响。不过在比较的字符串长度超过一定的限制时，n-gram 的准确度会有下降。因为随着字符串长度的增大，子字符串的分布会覆盖更多的情况，子字符串的交集个数也倾向于变大，由此导致结果的精确度降低。不过当仅针对源代码中变量标示符和字符串字面量的使用场景时，Beat Fluri 等在论文[27]中的实验表明，只在极少数的情况下，n-gram 的结果会差于 Levenshtein Distance。因此在 Pyreview 的实现中，我们使用了 n-gram 算法，我们使用的是 2-gram 相似度算法。我们的目的是得到两个字符串之间进行转化的代价 cost 值，在具体实现中我们使用下面的公式将得到的  $\text{sim}2g(s_1,s_2)$  转换成  $\text{cost}(s_1,s_2)$ ：

$$\text{cost}(s_1,s_2) = 2 * (1 - \text{sim}2g(s_1,s_2)), 0 \leq \text{cost} \leq 2$$

2.4.3 带缓存实现的结点列表匹配

当 node1 和 node2 都是结点列表时，我们需要寻找出 node1 和 node2 中结点之间的最优匹配组合，使得 node1 转换到 node2 的代价最小，并返回此匹配组合下对应的树的基本操作的集合。若用 diff-list 函数实现这一过程，Psydiff 算法中的伪代码如图 2-10 所示：

```
c0 = diff-node(node1[0], node2[0])
c1 = diff-list(node1[1:], node2[1:])
change1 = c0 + c1
c2 = diff-list(node1[1:], node2)
c3 = diff-list(node1, node2[1:])
change2 = c2 + delNode(node1[0])
change3 = c3 + insNode(node2[0])
change = minCost(change1,change2,change3)
return change
```

图 2-10 Psydiff 算法中 diff-list 函数的伪代码

在 Psydiff 算法的实现中，diff-list 函数是采用了上图中的直接的自顶向下递归实现的方式。图中 node1[0]表示 node1 列表中的第一个元素（下标从 0 开始）、node1[n:m]表示 node1 中从 n 开始到 m-1 的子列表，当 m 省略时表示从 n 开始直到列表结束。设 node1 的长度为 p，node2 的长度为 q， $n = \min(p, q)$ 。

在对 Psydiff 算法的性能测试中我们发现当 n 的值超过一定的规格时，即使

轻微的变化程序的运行时间也会急剧地增加。程序运行效率变差的根本原因在于 `diff-list` 函数会反复地用相同的参数值对自身进行递归调用，即它会反复求解相同的子问题。图 2-11 显示了当  $n=4$  时，`diff-list` 调用过程中问题规模的变化

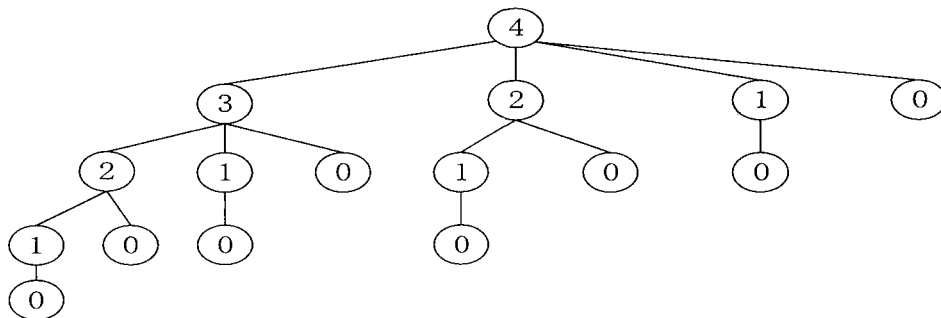


图 2-11 diff-list 函数中问题规模的变化

可以看出当问题规模为  $n$  时, 递归调用树共有  $2n$  个结点, 其中有  $2n-1$  个叶结点。另  $T(n)$  表示 node1 和 node2 的最小长度为  $n$  时 diff-list 的调用次数。此值等于递归调用树中根为  $n$  的子树中的结点总数。其中  $T(0) = 1$ , 且

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

可以证明 $T(n) = 2^n$ ，diff-list 的运行时间是  $n$  的指数函数。

为了解决这个问题,在 Pyreview 的实现中针对这段代码使用了动态规划方法,重新安排了问题的求解顺序,对每个子问题只求解一次,并将结果保存下来。这样在之后的求解过程中,如果再次需要此子问题的解,只需查找保存的结果,而不必重新计算。因此动态规划方法是付出额外的内存空间来节省计算时间,试图在程序的运行过程中做到时空权衡 (time-memory trade-off) [28]。

在对 **Psydiff** 算法的优化过程中，本文实现了带备忘的自顶向下实现方法。在问题求解的过程中，算法的基本思路仍是递归求解，但过程中会保存每个子问题的解（此处存放在二维数组中）。当需要一个子问题的解时，我们首先检查是否已经保存过此解。如果是，则直接返回保存的值；否则，按通常的方式计算这个子问题。因为在过程中，我们将中间结果进行了缓存，因此是一种带备忘的实现方式。加入备忘机制后的 **diff-list** 函数的伪代码如图 2-12 所示：

`table` 是一个二维数组，行和列的长度分别为 `node1.length` 和 `node2.length`。在改进的算法中，我们首先会检查所需值是否已知（2、3 行），如果是，则第四行直接返回保存的值；否则，第 5~12 行用原来的方法计算 `change` 集合的值，第

13 行将 `change` 的值存入到 `table` 中，第 14 行返回函数结果。

```
1. MEMORIZED-DIFF-LIST(table, node1, node2)
2. cached = table_lookup(table, len(node1), len(node2))
3. if cached is not None:
4.     return cached
5. c0 = diff-node(node1[0], node2[0])
6. c1 = diff-list(table, node1[1:], node2[1:])
7. changel = c0 + c1
8. c2 = diff-list(table, node1[1:], node2)
9. c3 = diff-list(table, node1, node2[1:])
10. change2 = c2 + delNode(node1[0])
11. change3 = c3 + insNode(node2[0])
12. change = minCost(changel, change2, change3)
13. table_put(table, node1.length, node2.length, change)
14. return change
```

图 2-12 缓存机制的 diff-list 函数实现伪代码

自顶向下的 `MEMORIZED-DIFF-LIST` 函数当求解一个之前已经计算过结果的子问题时，递归调用会立即返回，即 `MEMORIZED-DIFF-LIST` 函数对每个子问题只求解一次，而它求解了规模为  $0,1,\dots,n$  的子问题；为求解规模为  $n$  的子问题，第 6、8、9 行都会执行  $n$  次的 `diff-node` 函数；因此，`MEMORIZED-DIFF-LIST` 进行的所有递归调用的迭代次数是一个等差数列，其和是  $\theta(n^2)$ ，那么带缓存机制的 `diff-list` 函数的运行复杂度则为  $O(n^2)$ ，较之原来  $O(n^3)$  的实现在性能上有了很大的提升。

## 2.5 实验评估

### 2.5.1 实验对象

`Pyreview` 是用 Python 语言编写，最初也只适用于 Python 语言源代码的比较。选择 Python 语言的原因是其标准库中的 `ast` 模块[29]内置有完整的语法树结点定义；`ast` 模块生成的抽象语法树在内存中也是直接以树的形式存储。在这方面，Ruby 语言的 `ripper` 模块[30]会生成 `S-expression`[31]形式的结果、JavaScript 语言的 `esprima`[32]模块会生成 `JSON`[33]格式的结果，对于这些中间结果，我们需要在程序中再作一次到语法树的转换。而使用 Python 的 `ast` 模块，我们可以直接对语法树进行操作。

Python 是一门语法简洁，功能强大的动态编程语言。在工业界深受程序员喜爱，在教学、科学计算、机器学习、Web 开发等领域都有广泛的应用。在本章中我们会使用 Pyreview 工具去挖掘一些 Python 项目中的源代码变更信息，试图找到一些有用的结论。首先我们希望作为实验对象的 Python 项目满足下面的 3 点要求：

- (1) 源代码开源且使用 GIT[18]作为版本控制工具。
- (2) 项目在其应用领域内有很大的市场份额或技术影响力。
- (3) 我们希望研究的项目有一个较长的开发周期或项目有一个很大的规模。

针对第(1)点，我们从 Github 社区上寻找潜在的满足要求的项目；针对第(2)点，我们使用 Github 上一个项目获得的 star 的数量来判断一个项目的影响力；针对第(3)点，我们使用一个项目中提交(commit)的数量来评定一个项目的规模。

表 2-2 实验对象说明

项目	提交数	版本数	Star 数
django	22479	133	19115
flask	2427	16	19934
fabric	3603	95	7010
ansible	19119	97	16457
matplotlib	17750	49	3557
nupic	5614	55	4062
scrapy	5607	62	13741
mailpile	5043	12	6453
ipython	21410	46	9041

综合上述要求，本文最终选择 django、flask、ipython、fabric、ansible、matplotlib、mailpile、nupic、scrapy 作为我们的实验对象。这 9 个项目都是知名的 Python 开源工程。其中，django 和 flask 应用在 Web 开发领域；fabric、ansible 应用在系统运维领域；matplotlib、nupic 应用在科学计算领域；scrapy 是广泛使用的 Python 编写的爬虫工具；mailpile 是一个 Python 编写的邮件客户端；ipython 是一个增强的 python 交互式 shell。可见这 9 个项目基本涵盖了 Python 语言主要的应用场景。项目的具体信息见表 2-2。

2.5.2 实验综述

本次实验使用的计算机硬件条件如下：CPU 为 Intel Core i5-4570，主频 3.2GHZ；DDR3 16G 内存。操作系统为 Windows 10，用来开发的 Python 版本是 2.7.11。本次实验中，我们选取了表 2-2 中 9 个项目当前 master 分支的前 50 次提交，对其中进行过更改的 Python 源文件做代码结构化比较。一方面我们人工审核了代码结构化比较的结果，将每一个变更结点作为一个审查单元，统计得到了 Pyreview 在代码结构化比较方面结果的精确度。另一方面，针对 Pyreview 对原始 Psydiff 算法的优化，我们综合比对了 Pyreview 与 Psydiff 在代码结构化比较方面的性能表现。

2.5.3 代码结构化比较的实验结果

表 2-3 代码结构化比较功能的实验结果

项目	Python 源文件	检测出的变更实例数	误报的变更实例数	遗漏的变更实例数	准确率	召回率
django	60	151	2	0	98.7%	100%
ansible	46	209	3	0	98.6%	100%
fabric	26	141	0	1	100%	99.3%
matplotlib	38	211	2	0	99.0%	100%
flask	15	52	0	0	100%	100%
ipython	34	119	1	0	99.2%	100%
mailpile	82	450	7	0	98.4%	100%
scrapy	29	164	2	1	98.8%	99.4%
nupic	14	109	0	0	100%	100%
总计	344	1606	17	2	98.9%	99.9%

代码结构化比较的结果输出在一个 HTML 文件中，我们需要人工审核这些文件来判定结果的真伪。在本次实验中，我们选取了 9 个项目当前 master 分支的前 50 次提交，在这 450 次提交中，涉及到修改的 Python 源文件有 344 个，共有 1606 处变更。我们得到的统计结果在表 2-3 中。

实验结果表明 Pyreview 在代码的结构化比较方面表现良好，准确率维持在 98% 以上，召回率维持在 99% 以上。之所以有这种成绩，很大部分也得益于 GIT 的微量提交原则。我们分析了 9 个项目中，涉及到更改的 Python 源文件中变更实例的数量分布情况。

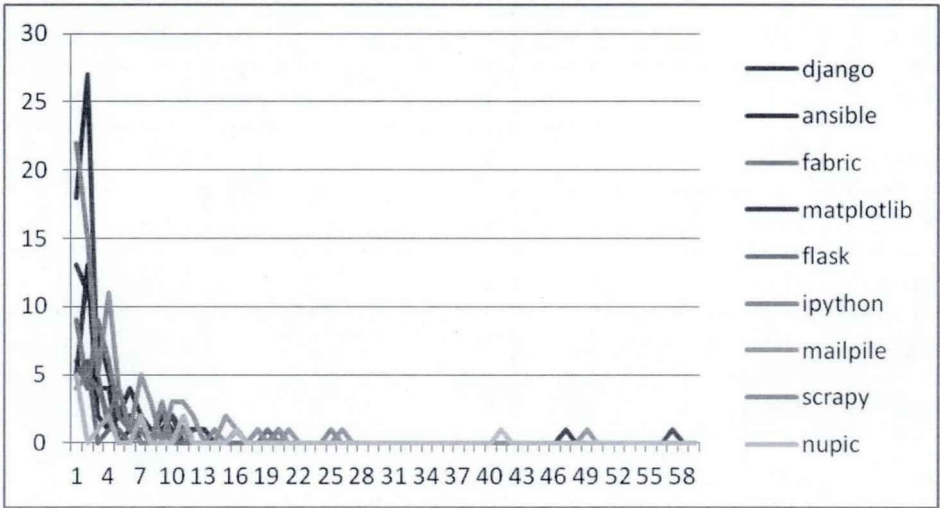


图 2-13 变更实例数量分布图

可以看出变更实例的数量分布是非常集中的，绝大多数的文件都只包含少量的变更实例。但一方面随着总的变更实例数量的增加，Pyreview 的准确率开始有一定下降。另一方面离异点文件的存在也导致 Pyreview 发现了一些错误的结果。具体到源代码级别，我们发现当对一个包含大量元素（且元素之间在字面量上特别相似）的列表或字典进行树的基本操作时，Pyreview 有更大的概率会得到错误的结点匹配集合。

2.5.4 与 Psydiff 算法的比较

在本小节中，一方面针对 Pyreview 中实现了对结点移动操作的检测，我们分析了表 2-3 中移动结点变更的分布情况；另一方面针对 Pyreview 在字符串比较算法上对 Psydiff 算法的改进，我们分析了二者在运行效率上的差异。

Pyreview 检测出的移动结点变更实例的分布如表 2-4 所示。根据表 2-4，我们可以看到移动变更结点的比例在 1.9% 到 12.0% 之间，移动变更结点在代码的变更中占有相当数量的比例。与原始 Psydiff 算法相比，在 1606 处代码变更中，Pyreview 检测出了 65 次移动结点变更，这是对代码结构化比较结果的一个有益补充，对原始 Psydiff 算法的一个重要的功能实现。

表 2-4 移动结点变更实例实验结果

项目	检测出的变更实例数	移动结点变更实例数	移动结点变更实例比例
django	151	7	4.6%
ansible	209	5	2.3%
fabric	141	4	2.8%
matplotlib	211	5	2.3%
flask	52	1	1.9%
ipython	119	4	3.4%
mailpile	450	21	4.7%
scrapy	164	5	3.1%
nupic	109	13	12.0%
总计	1606	65	4.0%

在 Pyreview 的实现中，我们优化了 Psydiff 算法在字符串字面量比较和结点序列匹配方面的运行效率。在本节中，对于同样的实验数据，Pyreview 和 Psydiff 的运行时间对比如图 2-14 所示：

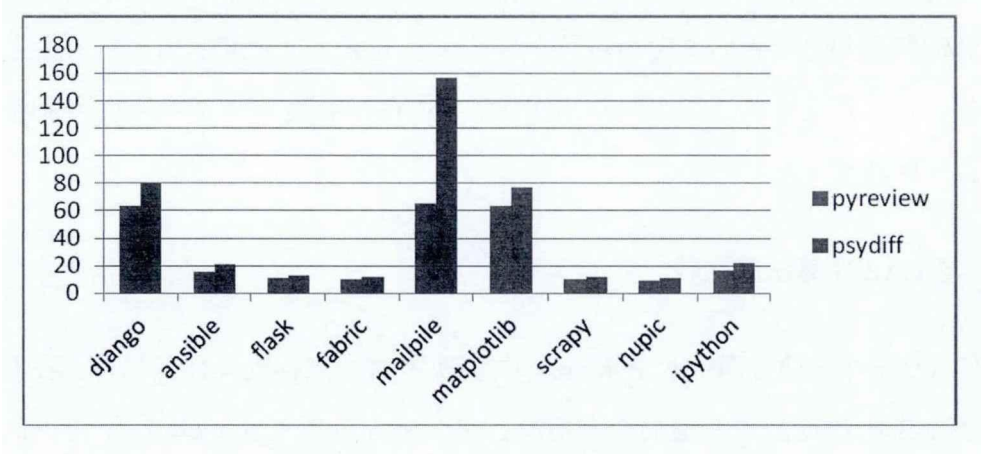


图 2-14 Pyreview 和 Psydiff 的运行时间对比

在本实验中，我们分别使用 Pyreview 和 Psydiff 的实现重复三次进行 2.5.3 小节中的代码结构化比较实验，取三次的平均时间作为最终的运行时间。可以看到在 9 个项目上，Pyreview 的运行时间都要少于 Psydiff，Pyreview 平均可以缩短 20%的运行时间。进一步的分析发现，当比较的源文件中含有大量的变更实例或者当变更实例中存在复杂变换时，Pyreview 在性能上的提升会更加明显，例如针



对 mailpile 项目甚至缩短了 59% 的运行时间。

## 2.6 本章小结

本章重点描述了代码结构化比较的实现原理和过程。我们首先引出了 Psydiff 算法，详细说明了结点相似度的求解过程和结点列表的匹配过程。在 Pyreview 的实现中，相比原始的 Psydiff 算法，我们对代码的结构性移动作了特殊的优化处理，改进后的字符串相似度计算方法和结点列表匹配算法也大大提升了 Pyreview 的运行效率。综上，Pyreview 中的代码结构化比较功能有以下四个优点：

- 1) 可以选择忽略源代码中空白字符、注释和文档字符串的影响。
- 2) 对语法规则中结点的定义敏感，可以感知结点的类型。如 Pyreview 可以正确识别字符“1000”与数字 1000 的区别，不会认为两者是相似的语法树结点。
- 3) Pyreview 可以抽取结点的插入、删除、更新和移动这四种树的基本操作，可以处理复杂的代码变更情景。
- 4) Pyreview 的运行效率优于 Psydiff 算法，在运行时间和结果的精确度上做到了更好的平衡。

## 第三章 Pyreview 代码变更类型分析

### 3.1 背景介绍

变更类型分析是指针对源代码中代码的改变提取出具体的模式，使我们可以对代码的变更作量化的分析[34]。变更类型分析一般是与代码版本控制系统整合，针对前后两个版本的代码，除了可以对代码变更进行分类分析外，我们还可以提取出变更的具体结点，在历史提交信息中，获取到结点完整的变化过程，即变更类型分析可以为源代码的演化分析提供丰富的信息。

变更类型分析功能的实现依赖于结点的结构化比较功能。在对代码结构化比较结果的分析后，我们可以提取出变更结点之间的联系，然后映射到具体的变更规则。

CHANGEDISTILLER[8]是现有的一个 Eclipse 插件，针对 Java 源代码实现了代码变更类型分析。CHANGEDISTILLER 中的核心算法 `change distilling` 是对 Chawathe 算法进行修正和优化后提出的新算法。对于输入的两颗抽象语法树，`change distilling` 首先会计算结点间的相似度，根据相似度的大小求解出结点间的最优匹配组合，进而根据匹配集合得到两颗树之间的编辑脚本。`Change distilling` 算法中自定义了结点的变更类型分类规则，通过对编辑脚本的分析，将具体结点的变化映射到特定变更规则，即可得到代码的变更类型。

`Change distilling` 算法可以对 Java 源代码进行变更类型分析，论文[8]中的 `bench mark` 也表明 `change distilling` 算法相比 Chawathe 算法具有更好的运行效率、实现结果具有更高的可靠性。经过实际的使用和分析后，我们认为 CHANGEDISTILLER 工具在以下 3 点仍然是存在提升空间的：

1) Chawathe[9]算法是一个自底向上的比较方法，优先进行子结点间的最优匹配。在算法的运行时会发生子树差异传播问题，子树上微小差异的累积可能会导致父结点匹配的失败。`change distilling` 算法通过后向的结果再优化在一定程度上去除了子树差异传播问题的影响，但本质上并没有更改 Chawathe 算法中的实现方式。我们倾向于认为后向遍历算法（即从叶结点开始进行结点间的匹配）并不能很好地抑制子树上差异的传播。

2) CHANGEDISTILLER 工具在实现中同样重新构建了源代码的抽象语法树。但在生成的新的语法树结点中，并不包含完整的位置信息（如结点出现的行号，在源文件字符串中的索引）。同时，CHANGEDISTILLER 中对语法树结点的分类粒度太大，如将赋值语句的字面量表示作为一个语法树上的叶结点，而不再更细致地区分出 `target` 结点和 `value` 结点。这种实现方法除更容易导致结点匹配的偏差外，也无法高效的鉴别结果的真伪。

3) CHANGEDISTILLER 工具最初是针对 Java 语言开发，可以检测出的变更类型更适用于静态语言。对于动态编程语言，如 Python 而言，change distill 算法受限于结点信息的不完整无法高效地处理动态语言中特殊语法糖的使用情景。

通过对以上三点的分析，我们认为需要一个新的专门针对 Python 语言（或更多动态语言）的变更类型分析工具，可以检测出 Python 中特有的语法糖的使用，可以在更精细的粒度上显示源代码的变化。

在本章中，我们通过对第二章代码结构化比较结果的分析，实现了 Pyreview 的代码变更类型分析功能。

## 3.2 变更类型分析功能的实现

第二章中代码结构化比较的输出结果是语法树中针对变更结点进行操作的集合，通过对这个中间结果的分析，我们发现可以求解出变更的类型，而且我们只需要做到以下两点：

1. 确定具体变更结点的角色和它本身以及父结点的类型。
2. 自定义具体的代码变更分类规则，即是一个由结点角色和相关结点的类型到具体变更类型的映射。

### 3.2.1 结点的角色和类型

Python 标准库中的 `ast` 模块只提供了由父结点到子结点列表的单向引用。若设父结点为 `p`，则 `p` 具有 `children` 属性，`p.children` 即可得到包含所有子结点的列表。为了实现变更分类的功能我们需要建立子结点到其父结点的引用。若设子结点为 `x`，父结点是 `p`，则我们给 `x` 新增 `parent` 属性，`x.parent` 即指向 `p`。一个结点可以通过 `parent` 属性逐步向上回溯到根结点，在 `ast` 模块的实现中，根结点始终

是一个 `Module` 类结点。`Python` 是一门面向对象的语言，语法树上的结点都与源代码中具体类的定义相对应。于是，一个结点可以看作一个类的对象，而对象是有类型的：如函数定义结点的类型是 `FunctionDef`，类定义结点的类型是 `ClassDef`。一个结点的角色是指这个结点作为其父结点的一个属性而存在，这个属性值的名称即是它的角色。子结点的角色与语法规则中结点类的定义相关。如在 `Python` 的语法规则中，赋值语句 `Assignment` 类结点的定义为 `Assignment(expr* targets, expr value)`。因此，设结点 `n` 是 `Assignment` 结点的子结点，则 `n` 的角色可能是 `targets` 或则 `value`。我们在抽象语法树的构建过程中，给 `n` 新增了 `role` 属性，`n.role` 返回一个字符串表示结点 `n` 的角色值。

综上在用 `ast` 模块得到 `Python` 源代码的抽象语法树后，我们会为树中的结点添加更多的元信息。以赋值语句 `a = 2` 为例，我们最终得到的扩展语法树为图 3-1 所示：

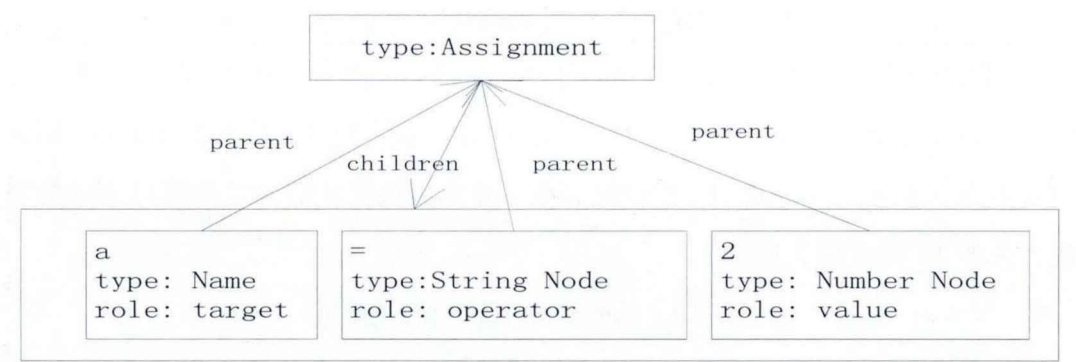


图 3-1 新的赋值结点

3.2.2 结点变更类型规则

在建立了父子结点之间的双向关联和确定了结点的角色和类型后，我们就可以构建变更结点到变更类型的映射。下面以简单赋值语句 `a = 1` 和 `a = 2` 为例来进行说明。

在经过结构化的比较后，针对上面两条语句我们得到的变更结点集合为 `[(Number(1),Number(2),1,0.5)]`，这是一个结点对的更新操作。分析 `(Number(1),Number(2),1,0.5)` 中的原始结点即 `Number(1)` 我们可以知道它的角色是 `value`，而它的父结点是一个 `Assignment` 类型的结点，因此我们可以建立图 3-2 中的映射关系推断出这次变更的类型是 `value in Assignment Statement change`。

当得到的变更结点表示结点的删除或增加操作时，我们可以直接由相应结点的类型来做推断。如当语句`a = 1`变为空时，删除的是一个 `Assignment` 类型的结点，我们可以直接得到变更类型为 `Assignment statement delete`。

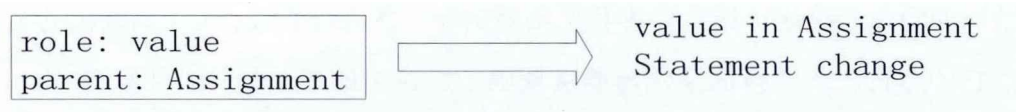


图 3-2 变更类型映射

表 3-1CHANGEDISTILLER 工具中的变更类型

Alternative Part Delete(1)	Alternative Part Delete(2)	Condition Expression Change(3)	Method Renaming(4)
Parameter Delete(5)	Parameter Insert(6)	Parameter Ordering Change(7)	Parameter Renaming(8)
Parameter Type Change(9)	Return Type Change(10)	Return Type Insert(11)	Statement Delete(12)
Statement Insert(13)	Statement Ordering Change(14)	Statement Parent Change(15)	Statement Update(16)

CHANGEDISTILLER 的作者设定了表 3-1 中的 16 种具体的变更类型。Python 作为一门动态语言在使用时不需要显式地指明变量的类型，因此在 Python 语言中并不存在编号为 9、10、11 的变更类型，除此之外，我们的工具可以高效的发现剩余的 13 种变更类型。

表 3-2 Python 语言特有的变更类型

变更类型	说明
Return Variables Number Change	Python 中的 return 语句可以返回多个值
Change in Yield Statement	Python 中的 yield 语句
Key Word Change	Python 支持关键字参数指定
Change in Lambda Statement	Python 支持 Lambda 表达式和闭包
Change in List Comprehension Statement	Python 中独有的列表推导表达式
Change in Decorators	Python 中扩展函数或类功能的修饰器

同时，我们认为 CHAGEDISTILLER 中一部分的分类规则粒度太大，遗漏了代码中的很多细节信息。Pyreview 相比 CHANGEDISTILLER，针对变更的结点可以获得更多的元信息，因此也可以实现更细致精确的变更类型划分。一方面对于编号为 12、13、14、15、16 中与 `Statement` 的变更相关的划分，我们可以进一步的得到 `Statement` 结点的类型来做更细致的划分。在我们的实现中，开发者可以得到 `FunctionDef`、`ClassDef`、`If Statement`、`While Statement`、`For Statement` 等更具体的 `Statement` 类结点的变化。另一方面，为了适配 Python 语言，我们也

定义了相当数量的与 Python 特有语法糖相关的变更类型，部分的新变更类型列举在表 3-2 中。

在确定了结点的变更类型分类后，我们就要自定义具体的映射规则，如 3.3.1 小节描述，Pyreview 的实现中是根据变更结点的角色和类型而确定。在表 3-3 中，我们列举了部分具有代表性的映射规则。

表 3-3 Pyreview 中具体的映射规则示例

变更类型	原结点类型	现结点类型	变更结点的角色	变更结点的父结点类型
类定义语句的删除	ClassDef	None	---	---
函数定义语句的插入	None	FunctionDef	---	---
函数重命名	Not None	Not None	name	FunctionDef
if 语句条件变化	Not None	Not None	test	IfStatement

可以看到针对结点的插入和删除操作，我们只需要知道相关结点的类型即可。对于函数重命名或 If 语句条件变化这些变更类型，在语法树上对应的是结点的更新操作，在这种情况下，首先我们可以得到原结点和现结点都不为空（Python 语言中，空类用 None 表示），然后我们需要根据变更结点的角色和父结点的类型来作出映射。需要注意的一点是在结点的更新操作时，变更结点可能处于一个深层次的嵌套中，为了得到具体的映射规则，我们必须沿着父结点引用向上遍历多次，才可得到正确的结果。

3.2.3 变更类型分析流程

图 3-3 描述了 Pyreview 中变更类型分析功能的运行流程。在具体的实现中，我们选择与 GIT 版本控制系统进行整合。首先我们从 GIT 中检索出我们希望进行比较的文件。在源代码的演化分析中，我们只关心特定的类或函数有没有发生更改。假设一个文件 f 有 1000 行，我们只希望研究其中类 Test 的变化过程，在一次提交中，f 更改了 300 行，但其中并没有包含类 Test 的更改。在这种情况下，我们依然进行了两个版本的文件 f 的代码结构化比较，毫无疑问，这属于一次无意义的操作。为了解决这个问题，我们在 Pyreview 的实现中使用 GIT 的 diff 命令来过滤掉不需要的输入。GIT 中的 diff 命令是 Unix 下 diff 工具的一个扩展实

现。通过 GIT 的 diff 命令我们可以得到变更代码行的集合。针对需要分析的特定类或函数，我们事先设定了一个关键字 token 集合，在进行代码的结构化比较之前，我们首先遍历变更的代码行中是否出现了具体的 token，若没有出现的话，我们会忽略这次文件的变更。

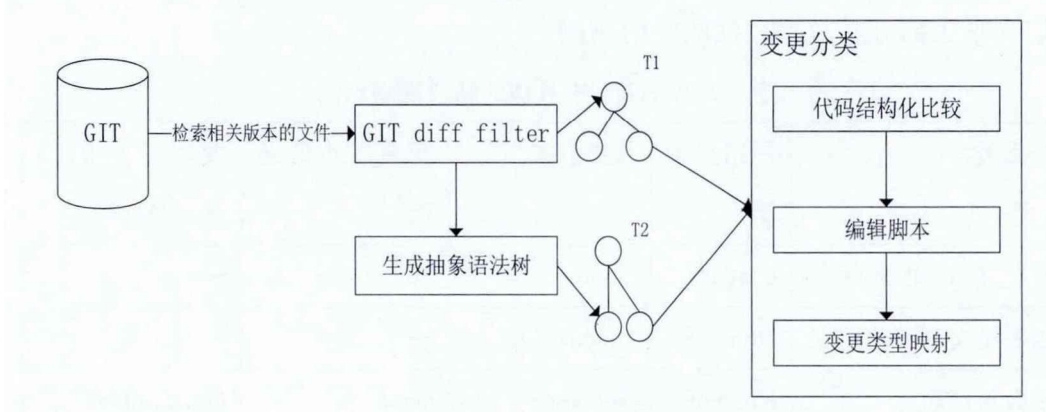


图 3-3 代码变更类型分析流程

我们使用 Python 标准库中的 ast 模块来构建源代码的抽象语法树，ast 模块的实现中完整保留了语法树中结点的位置信息。代码结构化比较的实现中，核心算法是一个自上而下的比较算法，逐步发现抽象语法树中不同的结点，并计算出一个近似具有最小转换代价的树的编辑脚本。最后根据编辑脚本中变更结点的角色和类型，我们会映射到具体的变更类型。

3.3 实验评估

变更类型分类实验的数据集和 2.5 小节中代码结构化比较实验相同。通过 Pyreview 的代码结构化比较发现在 9 个 Python 项目最近的 450 次提交中，涉及到修改的源文件有 344 个，共有 1606 处变更。在本节中，我们对这些变更进行分类，并验证了结果的有效性。

Pyreview 初步显示对 1606 处变更可以划分得到 1238 个变更类型实例。受限于没有充足的人力资源，我们只对 django、ansible、matplotlib、scrapy 这四个项目的结果做了人工审核。在 CHANGEDISTILLER 的 benchmark 中，作者是对表 3-1 中的 16 种变更类型，分别统计了实例的出现次数。在本文中，我们更倾向于根据变更结点的类型，将结果聚类后再展现。以此为规则的变更类型结果展现在表 3-4 中。

表 3-4 代码变更分类实验结果

变更类型/实例数	django	ansible	matplotlib	scrapy
Condition Statement Change	5	17	17	4
Import Statement Change	12	6	2	58
Call Statement Change	9	25	46	11
Parameters Change	28	11	49	11
Return Statement Change	5	0	4	0
Function Define Statement Change	10	8	12	1
Class Define Statement Change	8	1	1	0
Condition Expression Change	3	4	0	1
Decorator Expression Change	4	0	0	1
List Comprehension Statement Change	2	0	1	0
Yield Expression Change	0	1	0	0
Lambda Expression Change	4	1	0	1
Keyword Change	6	0	1	0

表 3-4 中的数据只是我们摘取的 Pyreview 分析结果中有代表性的部分数据。表 3-4 中的前 7 个变更类型，涵盖了大多数的变更实例，是编程语言中的基本组织结构。而后 5 种变更类型则为 Python 语言所特有,涉及到 Python 语言中修饰符、列表推导、yield 语句、lambda 表达式，关键字参数的使用。不过我们认为单纯地获得变更实例数目并不一定可以得到具有启发性的结论。变更实例功能的意义在于为研究特定模式的代码变更提供了可靠的实现，在这方面我们关于 Python 语言动态特性演进的研究正在进行中。

3.4 本章小结

本章以代码结构化比较的结果为基础，通过进一步地分析提供了代码变更分类功能。与前人的工作相比（如 CHANGEDISTILLER），除了实现 Python 语言特有语法糖的适配外，我们的分类规则更加详细，可以提供更细粒度的结果。我们巧妙地建立了语法树上父子结点间的双向关联、为结点赋予了自己的角色和类型。利用这个基础设施，开发者也可以自定义代码变更模式，实现更复杂的代码演进变化检测。我们最新的研究工作就是借助 Pyreview 中提供的基本的变更类型结果，去分析 Python 语言中动态特性使用的规律。代码变更分类功能可使开发者更好地理解源代码的变化，帮助开发者更好地完成后期的维护和开发工作。



## 第四章 Pyreview 代码克隆检测

### 4.1 背景介绍

之前的研究报告表明一个成熟的软件系统中会包含大量的重复代码, 这个比例维持在 6.4%至 20%之间[35]。开发者的本意是通过重复代码的使用提高软件的开发效率, 但如果重复代码本身存在功能缺陷, 则在后期的维护工作中, 开发者需要修复所有的克隆版本, 这将大大增加软件的开发成本[36]。至少在这种意义上, 代码克隆检测是一个非常有价值的工作, 可以辅助开发者进行软件质量评估和重构工作。重复代码的鉴定存在多种标准: 在语法级别上, 可以根据语句的相似度判断; 在语义级别上, 则可以根据代码段的运行效果判断。语义级别的比较, 属于程序的动态分析范畴, 具有很大的实现难度且受限于程序运行效率的限制。在本文的工作中, 我们是在语法级别上进行代码克隆的检测。

两个语句序列如果被认为是重复代码, 则它们之间的相似度要超过程序设定的阈值, 在 Pyreview 中阈值的设定是根据第二章中语法树结点的相似度而确定。

当前的工作大多是使用 anti-unification 算法来判断两个代码片段在语法树的结构上是否相似, Pyreview 的实现中也借鉴了 anti-unification 算法的思想。

### 4.2 现有工作的不足

Clonedigger 是我们找到的唯一一个针对 Python 语言的代码克隆检测同类型工具。Clonedigger 是由 Peter Bulychev 在 2008 年开发。根据论文[10]的描述, Clonedigger 也是在抽象语法树级别上进行代码克隆的检测, 使用 anti-unification 算法计算源代码对应的语法树在结构上的相似度。Clonedigger 在实现中使用了聚类的思想, 大致的检测流程如下:

#### 1) 构建源代码的抽象语法树

首先每一个源文件都会转化为等价的抽象语法树形式存储在内存中。针对 Python 语言, 使用标准 CPython 实现中的 compiler 模块来获取抽象语法树, 并将它存储在一个 XML 文件中。

2) 语句聚类

在 Clonedigger 的实现中，使用 anti-unification 算法计算语句间的距离，以此来定义两个待比较语句的相似度。相似度高的语句会被并入同一个聚类集中。

3) 由单一相似语句进而找到相似语句序列

针对一个语句，根据源代码中的位置信息，可以与相邻语句结合称为一个代码序列，在这个过程中，Clonedigger 会找到待定的重复代码集合。

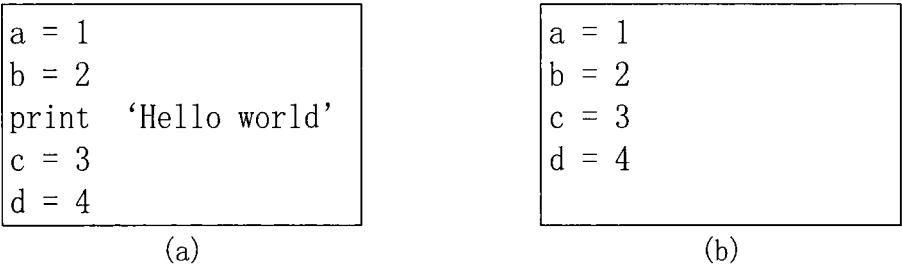
4) 重新使用 anti-unification 算法对第（3）步中得到的待定克隆集合进行结果的再提纯过程。

Clonedigger 的贡献点在于 1) 可以忽略源代码中注释和空白符的影响; 2) 可以处理源代码中常量和变量以及函数名称的改变; 3) 支持一些复杂的代码变更场景。不过在对 Clonedigger 的具体使用后，我们发现 Clonedigger 在以下三点上存在功能缺陷：

1) 不能处理克隆代码块中语句序列的插入和删除

Clonedigger 在实现中只能检测连续出现的语句序列，若在序列中出现其他的语法结点，则 Clonedigger 检测出的结果会出现一定的偏差。

针对图 4-1 中待比较的两个代码片段，因为（a）中 print 语句的出现，Clonedigger 最终的检测结果会认定存在两处重复代码，即 a = 1; b = 2 和 c = 3; d = 4。在这种情况下，我们认为如果将整个语句序列作为一次重复代码的出现会是一个更符合开发者常规思维的结果。



问题，Clonedigger 并没有对这种情况做特殊处理。在我们对 Clonedigger 进行测试的过程中，发现它并不能处理图 4-3 中的代码段。原因是 4-3(a)中第 3、4、5 行代码与(b)中第 3 行代码存在显著差异,这段子树上差异的传播使得 Clonedigger 的相似度算法出现偏差，最终错误认为两个文件中 hello 函数和 hi 函数的定义不是一次代码克隆。

```
def hello():
    a = 1
    b = 2
    c = 3
    d = 4
```

(a)

```
def hello():
    c = 3
    a = 1
    d = 4
    b = 2
```

(b)

图 4-2 代码中的语句顺序发生改变

```
def hello():
    a = 1
    a += 3
    if a > 3:
        print 'OK'
    try:
        b = a/0
    except Exception:
        pass
```

```
def hi():
    a = 1
    a = a + 3
    try:
        c = a/0
    except Exception:
        pass
```

图4-3 gapped clone code 示例

3) Clonedigger 的检测结果显示函数与类这些组织结构的变化。

Clonedigger 在检测过程中以语句作为代码比较的基本单位，在单条语句的基础上，扩展实现语句序列的比较。这种实现会丢失语句的上下文信息，使开发者不能在函数或类的层次观察代码的变化，我们认为这也是功能上的一种缺失。

针对上述问题，我们尝试在 Pyreview 中实现了代码克隆检测功能。

4.3 代码克隆检测功能的实现

4.3.1 Anti-unification 算法

Anti-unification 算法最初是由 Plotkin[11]和 Reynolds[12]提出，可以用来计算

两个抽象语法树之间的距离，并且通过距离值来完成相似语法树的聚类操作。设  $E_1$  和  $E_2$  是两个语法树结点，如果存在两个变换  $\sigma_1$  和  $\sigma_2$ ，使得  $\sigma_1(E) = E_1$  和  $\sigma_2(E) = E_2$ ，则称  $E$  是  $E_1$  和  $E_2$  共同的规约形式[37]。Anti-unification 算法的目的就是针对给定的结点  $E_1$  和  $E_2$ ，找到所需变换次数最少的  $E$ 。

将 anti-unification 算法应用于树  $T_1$  和  $T_2$  则是指替换  $T_1$  和  $T_2$  子树中的特殊结点 [38][39]。若占位符用  $?n$  表示，那么算法对函数调用语句  $Add(Name(i),Name(j))$  和  $Add(Name(n),Const(1))$  的输出结果是  $Add(Name(?_1),?_2)$ 。特殊地，对  $Add(Name(i),Name(j))$  和  $Add(Name(j),Name(j))$ 的输出结果是 $Add(Name(?_1),Name(?_1))$ 。

通过 anti-unification 算法可以计算出两颗树之间的 anti-unification distance[40]。我们假设树  $T_1$  和  $T_2$  可以通过变换  $\sigma_1$  和  $\sigma_2$  得到它们的规约形式  $U$ 。 $n$  为  $U$  中占位符的数量，那么  $\sigma_1$  和  $\sigma_2$  的映射集合为  $\{?_1, ?_2, \dots, ?_n\}$ 。我们同时规定一颗树的大小为它所包含的叶结点的数目。那么 anti-unification distance 的定义为  $\sigma_1$  和  $\sigma_2$  中所有替换子树的大小的和。

以树 $Add(Name(i),Name(j))$ 和 $Add(Name(n),Const(1))$ 为例，我们可以得到  $\sigma_1$  为  $\{i/?_1, Name(j)/?_2\}$ ， $\sigma_2$  为  $\{n/?_1, Const(1)/?_2\}$ 。 $\sigma_1$  和  $\sigma_2$  中包含的子树为 $|i| = |n| = |Name(j)| = |Const(1)| = 1$ ,则它们之间的 anti-unification distance 为 4。

anti-unification distance 说明了两颗语法树之间的差距，可以看作是树的编辑距离的等价表现，只是在 anti-unification distance 的计算中，只允许出现特定的树的基本操作。

### 4.3.2 Pyreview 中的实现算法

源代码经过第二章的结构化比较算法后，会生成一个树的基本操作序列集合。其中可以表示三种基本的语法树操作，即结点的插入、删除和更新。我们的代码克隆检测算法依然是一个后向处理方法，我们会遍历基本操作序列集合，重新计算其中结点进行转换的 cost 值。

受 anti-unification 算法的启发，我们在遍历的过程中会对不同类型的对象采取不同的处理方法。首先，对于表示插入或删除的对象，我们会计算其中变更结点的大小，然后根据不同的 size 值设定不同的权重。例如，在两个版本的源代码

中发生一次函数定义结点删除操作，得到的基本操作为(FunctionDef, None, 8,0)。我们定义变更结点的 size 值位于 0~5 之间时其权重为 0.5,5~10 之间时为 0.8,超过 10 时为 1。对于删除或插入操作，cost 值即是变更结点的 size 值。FunctionDef Node 的 size 值为 8，对应的权重为 0.8，那么转换后新的 cost 值是  $8 * 0.8 = 6.4$ 。

对于表示结点更新的对象，我们的方法较之 anti-unification 算法，将结点的角色也考虑在内。我们会得到变更结点的角色，为不同的角色设定不同的计算权重。采用这种方法的原因是我们认为不同的角色在源代码中的作用和影响力是不同的。举例来说，对于图 4-4 中的两组代码变更，(a)和(b)都只有一处更改，原始的 cost 值甚至是(b)小于(a)。但如果从代码克隆的角度分析两组代码，(a)的相似度必然是高于(b)的，也就是我们认定在函数调用语句中，函数名称的变化较之参数值的变化有着更加大的作用和影响力。

xxmethod(1,2)	xxmethod(1,3)
(a)	
xxmethod(1,2)	xymethod(1,2)
(b)	

图 4-4 Pyreview 中代码克隆检测示例代码

在第 3 章变更类型的分析中，我们就已经可以得到结点的角色，在上例中，我们可以将参数结点的权重设为 0.3，而函数名称结点的权重设为 0.8，这样我们新计算得到的 cost 值就可以反映真实的代码复用情况。

Pyreview 中的代码克隆检测方法包含下面几个步骤：

- 1) 程序的输入是待比较的源文件。针对每个文件，得到它所有基本组织结构的抽象语法树。在 Pyreview 的实现中，我们将类与函数的定义以及最顶层的面向过程的代码集合作为一个文件的基本组织结构。
- 2) 两两比较集合中的抽象语法树，得到原始的变更结点集合。
- 3) 使用我们定义的新的代码克隆检测算法，重新计算结点进行转换的 cost 值，并进而求出新的语法树相似度，如果相似度值超过我们设定的阈值则认为这是一次代码克隆现象。

在步骤 1) 中，我们要得到所有基本结点的抽象语法树，这些基本结点是可以嵌套定义的。为了提高程序运行的效率，我们使用哈希表来存储每个文件对应的语法树集合。

在步骤 2) 中，我们要在一个双层循环中比较所有的语法树匹配。为了提高

运行效率，一方面我们可以限定比较的语法树结点的条件，如要求嵌套的层次相同、内部包含的语句序列的长度要超过一定的值，避免大量零碎结点间进行比较。另一方面，我们可以并行化进行结点间的比较，这样可以使程序的运行效率有 3~4 倍的提升。

在步骤 3) 中，我们初始设定的新的相似度计算公式为： $\text{similarity}(n1, n2) = 1 - \frac{\text{newcost}(n1, n2)}{\text{oldcost}(n1, n2)}$ ，阈值为 0.5。后文中的 benchmark 表明我们设定的参数可以得到很好的检测结果。

综上，我们认为 Pyreview 的实现在以下两个方面是优于 Clonedigger 的。第一点，Pyreview 中代码克隆检测功能的实现仍然是基于第二章中核心的代码结构化比较算法。实验证明本文中的代码结构化比较算法可以对语句序列中微量的插入和删除操作免疫，可以分析比较 gapped clone code，这原生解决了 Clonedigger 中的前两个问题。第二点，在 Pyreview 的实现中，我们将类与函数的定义以及最顶层的面向过程的代码集合作为算法的基本比较单位，虽然这在一定程度上降低了结果的精确度，但与 Clonedigger 相比，我们的结果具有更丰富的信息量，更加具有分析价值。

## 4.4 实验评估

本次实验使用的计算机的硬件条件如下：CPU 为 Intel Core i5-4570,主频 3.2GHZ; DDR3 16G 内存。操作系统为 Windows 10，用来开发的 Python 版本是 2.7.11。

为了与 Clonedigger 进行性能对比，在本次实验中，我们也采用了与 Clonedigger 官方文档中相同的实验对象，即 nltk 的 0.9.2 版本。实验对象包括 139 个 Python 源文件，总计 36427 行源代码（不包括注释和空白行）。Clonedigger 工具在检测时，是以一条语句为基本单位。而 Pyreview 是以面向对象编程中的基本组织结构作为比较的基本单位。针对 Python 语言即是以类的定义、函数的定义、最顶层的面向过程的语句序列作为基本比较单位。实验中的 nltk 项目共有 3064 个基本比较单元。

最初的实验中，本文两两比较了类型相同的所有基本单元，获得了 2370405 条记录。以相似度区间划分记录集合，结果展现在表 4-1 中。

表 4-1 完整的代码克隆检测结果

相似度区间	记录条数
0.0~0.1	2038642
0.1~0.2	180804
0.2~0.3	92981
0.3~0.4	23605
0.4~0.5	19744
0.5~0.6	10549
0.6~0.7	2356
0.7~0.8	709
0.8~0.9	605
0.9~1.0	81
1.0	329

与最初的设想不同，我们发现了大量相似度在 0.6 之上的代码块。进一步的审查发现这些代码块中语句序列的长度几乎都在 2 以下，本文认为小于 2 条语句的基本单元并没有包含足够的能够得到代码克隆检测结果的信息。于是，我们将比较的基本单元的语句序列的长度限制在 2 条以上，相似度阈值也设为 0.6，过滤后的实验结果如表 4-2 所示。

表 4-2 过滤后的代码克隆检测结果

相似度区间	记录数
0.6~0.7	114
0.7~0.8	74
0.8~0.9	55
0.9~1.0	30
1.0	13

接下来为了获得代码克隆检测结果的正确率，我们人工审核了表 4-2 中的 286 条记录对应的源文件，最终得到的结果见表 4-3。

表 4-3 代码克隆检测结果的正确率

相似度区间	记录数	正确记录	准确率
0.6~0.7	114	106	93.0%
0.7~0.8	74	66	89.1%
0.8~0.9	55	54	98.1%
0.9~1.0	30	30	100%
1.0	13	13	100%
总计	286	269	94.1%

4.4.1 与 Clonedigger 实验数据的比较

- 1) 结果数量的比较
- Clonedigger 针对 nltk 项目找到了 356 处可能的代码克隆点。Pyreview 当限制序列长度在 2 以上时找到了 286 处。当限制在 1 以上时，可找到 378 处。因二者实现算法不同，综合鉴定，这一点上，Pyreview 表现与 Clonedigger 相似。
- 2) 运行效率比较
- Clonedigger 官方数据显示，运行 nltk 项目的检测，总耗时为 2148 秒。Pyreview 单线程运行耗时 2098 秒，多线程运行（核心设为 8）可优化到 742 秒。不过 Clonedigger 官方数据仍是 08 年的数据，我们本机重新运行 Clonedigger，相同硬件条件下，Clonedigger 的运行时间为 810 秒。在运行效率上，Pyreview 较之 Clonedigger 有很大的改善。
- 3) 结果覆盖情况
- Pyreview 实现了 gapped clone detection[41]，在一定程度上，可对语句的插入和删除免疫。这两点是 Clonedigger 没有实现的。当 Clonedigger 遇到上述情况时，会认定存在多处可能的克隆点，而在语义上，人们更倾向于认定只存在一处。为此，我们也分析了两者的覆盖情况，Pyreview 得到的结果中覆盖了 Clonedigger 工具 356 处中的 328 处。



## 4.5 本章总结

在本章中以代码结构化比较的结果为基础，我们通过进一步地分析提供了代码克隆检测功能。Pyreview 中的代码克隆检测功能将类与函数的定义以及最顶层的面向过程的代码集合作为基本的比较单元。这种实现方式下，检测结果中会有一些部分的无用信息，但结果具有更好的可分析性和可阅读性。

与前人的工作相比（如 Clonedigger），受益于 Pyreview 中茁壮的代码结构化比较算法，我们的克隆检测算法可以分析 gapped clone code，在一定程度上对微量的语句插入和删除操作免疫。本章最后的实验也表明，相比 Clonedigger，Pyreview 除了在运行效率上大幅提升外，在结果的数量和精确度上也超过了 Clonedigger。

## 第五章 总结与展望

### 5.1 工作总结

Python 是一门非常优秀的编程语言，其强大的表达能力，极大地提升了程序员的开发效率，但同时也增加了相关源代码分析工具的开发难度。在本文中，我们设计开发了 Pyreview，一个基于抽象语法树差异提取的 Python 代码分析工具。Pyreview 提供了三种源代码分析功能，即代码结构化比较、代码变更类型分析和代码克隆检测。在代码结构化比较方面，本文针对语法树中结点的差异比较，提出了改进的字符串距离计算方法，同时实现了缓存的结点列表匹配方法，提高了已有算法的运行效率。在代码变更类型分析方面，本文提出了一个较为完整的针对 Python 语言的代码变更类型规则集合，可以有效地应对 Python 中特殊的语法糖和动态特性的使用。在代码克隆检测方面，本文提出的算法可以检测 gapped clone code，对克隆代码块中微量的语句删除或增加操作免疫。此外，在本文中，我们也进行了 Preview 向其他语言的移植工作和将 Pyreview 应用于代码演化分析的尝试。

在本文的实验中，我们首先验证了 Pyreview 在代码差异比较方面的表现，经过人工审核后，实验结果的准确率和召回率均在 98% 以上。在代码变更类型分析方面，Pyreview 准确找到了与 Python 语言特殊语法糖和动态特性使用相关的变更。而在代码克隆检测方面，Pyreview 相对之前的工作，如 Clonedigger，除运行效率的提升外，更重要的是解决了前者遗留的对 gapped clone code 的检测问题。同时，经过对结果的审查，我们发现针对相同数据集，Pyreview 可以发现更多的候选克隆代码块。

### 5.2 未来工作

针对本文的研究，还有许多需要改进和完善的地方。

(1) 代码结构化比较模块中核心算法运行效率的提升

现在的实现中，结构化比较模块核心算法的时间复杂度是  $O(n^2)$ ，当比较的

文件中发生大量代码行的改变时，这会成为 Pyreview 运行效率的一个瓶颈。在后面的开发中，除既有算法的优化外，我们会尝试利用一些 ground truth，提升算法在特定场景中的运行表现。

### (2) 代码变更分类规则的细化和相关源代码信息的二次分析

在本文中，我们虽然实现了 Python 源代码变更分类的功能。但因为相关经验的缺失，我们的分类规则还有很大的提升空间。我们还要继续完善代码分类逻辑，使其可以应对更多的特殊情况。同时，另一方面，我们希望借助这一功能，对 Python 中一些特有的语言属性作更细致的研究。目前我们针对 Python 语言中动态特性使用规律的研究正在进行中。

### (3) 代码克隆检测结果精度的提升

目前 Pyreview 提供的代码克隆检测功能还处于初步原型阶段，我们的克隆检测算法的基本比较单位是类的定义，函数的定义和最顶层的面向过程的语句序列。由于缺乏对结果的再精炼过程，现在的检测结果中会包含相当数量的无用信息。在后面的工作中，我们希望可以细化算法的基本比较单位，尽量去除结果中的无用信息。同时，本文中的代码克隆检测算法是一个后向的中间结果再分析过程，这样会导致子树差异累积扩大问题[9]。后续开发中，我们打算将代码克隆检测作为独立的工具分离出 Pyreview，重新实现更高效可靠的检测算法。

## 参考文献

- [1] M.M Lehman, programs life cycles and laws of Software Evolution Proc. IEEE, pp. 1060-1076, Sept. 1980.
- [2] A.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll, Predicting Source Code Changes by Mining Change History, IEEE Trans. Software Eng., vol. 30, no. 9, pp. 574-586, Sept. 2004.
- [3] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, Mining Version Histories to Guide Software Changes, IEEE Trans. Software Eng., vol. 31, no. 6, pp. 429-445, June 2005.
- [4] G.C. Murphy, J. Singer, and K.S. Booth, Hipikat: A Project Memory for Software Development, IEEE Trans. Software Eng., vol. 31, no. 6, pp. 446-465, June 2005.
- [5] H. Gall, K. Hayek, and M. Jazayeri, Detection of Logical Coupling Based on Product Release History, Proc. Int'l Conf. Software Maintenance, pp. 190-198, Nov. 1998.
- [6] Brenda S. Baker. A Program for Identifying Duplicated Code. Computer Science and Statistics, 24: 49-57, 1992.
- [7] Karp, Richard M, Rabin, Michael, Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31 (2): 249-260, March. 1987.
- [8] Fluri, Pinzger, Gall, H.C, Change distilling: Tree differencing for fine-grained source code changes extraction. IEEE Trans Softw. Engine. 33, 11, 725-743. 2007.
- [9] S.S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, Change Detection in Hierarchically Structured Information, Proc. ACM Sigmod Int'l Conf. Management of Data, pp. 493-504, June 1996.
- [10] Peter Bulychhev, Marius Minea, Duplicate code detection using anti-unification, 2008.
- [11] G. D. Plotkin, A note on inductive generalization, Machine Intelligence, 153-163, 1970.
- [12] J. C. Reynolds, Transformational systems and the algebraic structure of atomic formulas, Machine Intelligence, 5(1):135-151, 1970.
- [13] Psydiff algorithm, <https://github.com/yinwang0/psydiff>
- [14] Yuan, Y. and Guo, Y. CMCD, Count Matrix Based Code Clone Detection, 18th Asia-Pacific Software Engineering Conference. IEEE, Dec. 2011, pp. 250-257, 2011.
- [15] Chen, X., Wang, A. Y., & Tempero, E. D. A Replication and Reproduction of Code Clone Detection Studies. In ACSC pp. 105-114, 2014
- [16] Christine M. Neuwirth, Ravinder Chandhok, David S.Kaufer, Flexible Diffing in a collaborative writing system, cscw 92 pp 147-154, 1992.
- [17] P Bille, A survey on tree edit distance and related problems, Theoretical computer science, 2005.
- [18] L Torvalds, J Hamano. Git: Fast version control system. <http://git-scm.com>, 2010.
- [19] Levenshtein, Vladimir I. Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10 (8): 707-710, February 1966.
- [20] Li Yujian, Liu Bo. A Normalized Levenshtein Distance Metric, IEEE Transactions on Pattern Analysis and Machine Intelligence, pp 1091-1095 June 2007.
- [21] Thomas H.Cormen, Charles E. Leiserson, Ronald L.Rivest, Clifford Stein. Introduction to Algorithms Third Edition, chapter 15.

- [22] L.R. Dice, Measures of the Amount of Ecologic Association between Species, *ESA Ecology*, no. 26, pp. 297-302, July 1945.
- [23] P. Jaccard, The Distribution of the Flora in the Alpine Zone, *New Phytologist*, vol. 11, no. 2, pp. 37-50, Feb. 1912.
- [24] G.W. Adamson and J. Boreham, The Use of an Association Measure Based on Character Structure to Identify Semantically Related Pairs of Words and Document Titles, *Information Storage and Retrieval*, vol. 10, nos. 7-8, pp. 253-260, July-Aug. 1974.
- [25] Z. Xing and E. Stoulia, UMLDiff: An Algorithm for Object- Oriented Design Differencing, *Proc. Int'l Conf. Automated Software Eng.*, pp. 54-65, Nov. 2005.
- [26] J. Weidl and H.C. Gall, Binding Object Models to Source Code: An Approach to Object-Oriented Re-Architecting, *Proc. Computer Software and Applications Conf.*, pp. 26-31, Aug. 1998.
- [27] B. Fluri and H.C. Gall, Classifying Change Types for Qualifying Change Couplings, *ICPC*, pp. 35-45, 2006.
- [28] M. Hellman, A cryptanalytic time-memory trade-off, *IEEE Transactions on Information Theory*, pp. 401-496, Jul 1980.
- [29] Python ast Module, <https://docs.python.org/2.7/library/ast.html>.
- [30] Ruby ripper module, <http://ruby-doc.org/stdlib-2.1.2/libdoc/ripper/rdoc/index.html>
- [31] John McCarthy, Massachusetts. Recursive Functions of Symbolic Expressions and Their Computation by Machine. *Communications of the ACM*, 1960.
- [32] Esprima parser, <http://esprima.org/>
- [33] Bray, Tim. JSON Redux AKA RFC7159. Ongoing. Retrieved 16, March 2014.
- [34] Will Snipes, Code Hot Spot: A tool for extraction and analysis of code change history, *ICSM*, pp. 392-401, Sept 2011.
- [35] C. K. Roy, J. R. Cordy. A Survey on Software Clone Detection Research, 2007.
- [36] Miryung Kim, Vibha Sazawa, David Notkin, Gail Murphy. An empirical study of code clone genealogies, *FSE-13*, pp. 187-196, 2005.
- [37] W. Evans, C. Fraser, F. Ma. Clone Detection via Structural Abstraction, 2007.
- [38] M.H. Sorensen, R. Gluck. An algorithm of generalization in positive supercompilation, In *Logic Programming: Proceedings of the International Symposium*, MIT Press, 1995.
- [39] C. Oancea, C. So, and S. M. Watt. Generalization in Maple. In Ilias S. Kotsireas, editor, *Maple Conference 2005*, pages 377-382, Waterloo, Ontario, 2005.
- [40] P. Bille. A Survey on Tree Edit Distance and Related Problems, 2005.
- [41] Y. Ueda, T. Kamiya, S. Kusumoto, On detection of gapped code clones using gap locations, *Software Engineering Conference Ninth Asia-Pacific*, pp. 327-336. 2002.

## 致谢

首先，我要感谢我的导师徐宝文教授，为我确定了论文的研究方向，在论文最后的定稿阶段也倾注了徐老师的大量心血。徐老师在繁忙的工作之余必定会腾出时间关心我们在生活和学术上遇到的问题。在生活上，徐老师教会了我许多为人处事的道理，使我学会更成熟的思考问题。在学术上，徐老师治学严谨，对研究领域有着独到的见解和敏锐的洞察力，这些都促使着我不断进步，不断追逐老师的脚步。

其次，感谢陈林老师对本篇论文的修改，陈老师的建议对本文的写作提供了莫大的帮助。同时衷心感谢实验室的所有老师。他们给我们提供了优越的学术环境，从他们每一次的学术探讨中，我都受益匪浅，无论在学术还是生活上都丰富了自己的阅历和见识。感谢周毓明老师、李言辉老师、许蕾老师在学术研究中给予了我很多的启迪和帮助。

再次，感谢实验室的所有成员，感谢陈芝菲、曹赖平等对本文的指导及建议，感谢王蓓蓓、汪睿等同学对我的鼓励及支持。

最后，要特别感谢我最亲爱的家人，是他们的理解与关怀，让我可以全心全意的投入到学习生活中，让我一直没有迷失自己的理想和方向。

三年的硕士生涯已经接近尾声，对于这三年有太多的情感需要述说，我学会了成长，学会了分享，感谢在这三年中所有人对我的无私帮助。

## 攻读硕士学位期间主要的研究成果

### 专利

[1] 徐宝文、李清言、陈林，一种基于抽象语法树差异提取的代码变更分析方法，专利申请号：201610321503.8

