



单位代码 10006

学 号 39061512

分 类 号 TP391

北京航空航天大学  
BEIHANG UNIVERSITY

# 毕业设计(论文)

## 程序代码相似性检测技术 研究与实现

学院名称	计算机学院
专业名称	计算机科学与技术
学生姓名	杨博洋
指导教师	李舟军

2013 年 6 月 4 日

程序代码相似性检测技术研究

杨博洋

北京航空航天大学

# 北京航空航天大学

## 本科生毕业设计（论文）任务书

I、毕业设计（论文）题目：

程序代码相似性检测技术研究

II、毕业设计（论文）使用的原始资料（数据）及设计技术要求：

1、原始材料：BuaaSim 相似代码集。

2、设计技术要求：设计的应用程序要能够满足代码抄袭监测、代码相似聚类，兼容 C 和 C++ 语言，并保证效率可以满足实际应用的需求。

III、毕业设计（论文）工作内容：

在研究现有代码相似性检测系统架构和算法基础之上，设计一套新的代码相似检测系统，并实现一个可对 C/C++ 代码集进行代码相似性检测、相似代码聚类的工具。

IV、主要参考资料：

《算法导论》

《柔性字符串匹配》

熊浩 代码相似性检测技术的研究工具与实现

赵长海, 晏海华, 金茂忠 基于编译优化和反汇编的程序相似性检测方法

计算机学院计算机科学与技术 专业类 390615 班

学生 杨博洋

毕业设计（论文）时间： 2013 年 2 月 20 日至 2013 年 6 月 10 日

答辩时间： 年 月 日

成 绩：

指导教师：

兼职教师或答疑教师（并指出所负责部分）：

系（教研室） 主任（签字）：



## 本人声明

我声明,本论文及其研究工作是由本人在导师指导下独立完成的,在完成论文时所利用的一切资料均已在参考文献中列出。

作者: 杨博洋

签字:

时间: 2013 年 6 月

---



## 程序代码相似性检测技术研究是实现

学 生：杨博洋

指导教师：李舟军

### 摘 要

随着计算机行业日益发展，代码抄袭现象日甚。在越发重视知识产权保护的今天，保护代码著作权、检测代码抄袭已成为一个重要需求。当下有很多用来进行代码相似性检测的工具，但都效果欠佳且很少开源，而代码抄袭的手段日新月异，网络上已出现大量用于改写他人代码至肉眼难以识别的工具，如代码混淆器等。因此，本文在对当今主流的代码相似性检测技术进行研究的基础上，实现了一套新的专门针对 C/C++ 语言的能够实现代码相似性检测、代码相似聚类的工具，并详细阐述这套工具的系统设计、算法设计和实现过程，并在最后对系统的正确率、召回率和时空效率进行了分析，并通过一份通用的测试数据与其他现有的成熟算法进行对比，表明本系统在各项指标均表现出色，对将来学校 ACM 集训队的代码查重和程序设计课程的作业抄袭检测有很大的实用价值。

**关键词：**C/C++，代码相似性检测，聚类

---



## Research on Detecting Plagiarism in Source Code

Author : YANG Bo-yang

Advisor : LI Zhou-jun

### Abstract

With the growing of computer's technology, code plagiarism is wide spreading. In the time of increasingly attach importance to protection of intellectual property, copyright protection and testing code plagiarism has become an important requirement. Today there are many tools for code similarity detection but ineffective and rarely are open source, while the method of copying code is updating, there has been a large number of tools used to rewrite the code to others cannot be identified naked-eye in the internet, such as code obfuscator. Meanwhile, this article implements a new tool for detecting C/C++ plagiarism codes based on today's mainstream technology of similarity detection, and explains the procedure of this tools' system designing, algorithm designing and implementation, and tests on correctly rate, recalled rate and spatio-temporal efficiency of this system, to show that this tool's performance is perfect enough for BUAA ACM Team's code plagiarism detection and programming homework's plagiarism detection.

**Key words:** C/C++, code plagiarism detection, cluster

---



## 目 录

本人声明 .....	I
摘 要 .....	I
目 录 .....	III
1 绪论 .....	1
1.1 论文选题的背景及意义 .....	1
1.2.1 国外研究现状 .....	2
1.2.2 国内研究现状 .....	3
1.3 研究目标与内容 .....	4
1.4 课题来源 .....	4
1.5 论文组织结构 .....	4
2 相关概念和技术 .....	6
2.1 当今主流代码相似性检测模式 .....	6
2.1.1 预处理 .....	6
2.1.2 中间代码转换 .....	7
2.1.3 比较单元生成 .....	8
2.1.4 匹配算法 .....	9
2.2 主流算法介绍 .....	9
2.2.1 基于属性度量 .....	9
2.2.2 基于串 .....	10
2.2.3 基于标识符 .....	10
2.2.4 基于树 .....	10
2.2.5 基于图结构 .....	12
2.3 MINGW 介绍 .....	13
2.4 常用字符串相似度算法 .....	13
2.4.1 Longest Common Substring .....	13





---

2.4.2 Levenshtein Distance .....	14
2.4.3 Rabin-Karp_Matcher .....	14
2.5 常用聚类算法 .....	15
2.5.1 k-Nearest Neighbor algorithm(KNN) .....	15
2.5.2 k-means .....	15
2.6 本章小结 .....	16
3 系统与算法设计 .....	17
3.1 PLAGIARISMChecker 系统设计 .....	17
3.1.1 系统需求分析与设计原则 .....	17
3.1.2 系统模块设计 .....	18
3.1.3 功能模块设计 .....	19
3.2 算法设计 .....	19
3.2.1 代码相似度计算 .....	19
3.2.2 代码相似聚类算法设计 .....	21
3.3 小结 .....	23
4 系统实现 .....	24
4.1 软件功能实现 .....	24
4.1.1 代码优化 .....	24
4.1.2 词法分析 .....	25
4.1.3 编译优化反汇编 .....	26
4.1.4 代码结构设计 .....	28
4.2 算法实现 .....	28
4.2.1 代码相似度算法实现 .....	28
4.2.2 代码相似聚类算法实现 .....	29
4.3 小结 .....	30
5 系统试验设计与分析 .....	32
5.1 基本功能测试 .....	32

---



---

5.1.1 代码载入.....	32
5.1.2 代码相似度计算和展示.....	33
5.1.3 相似代码聚类 and 展示.....	35
5.2 算法性能分析.....	36
5.2.1 与 BuaaSim 性能对比.....	36
5.2.2 C++/C 跨语言代码相似性检测.....	38
5.3 大规模真实数据测试.....	39
5.4 小结.....	40
结论.....	41
致谢.....	42
参考文献.....	43

---

## 1 绪论

### 1.1 论文选题的背景及意义

程序代码相似性的检测最初是从重复代码检测和代码的优化演变而来。重复代码是指在同一个程序中重复出现的代码块，其最初的出现大多是出于程序需要而频繁调用，但却增加了程序代码的容量和程序运行的时间，这就需要进行代码优化，即对一份源程序文件进行相似代码检测识别。在当今技术水平日益发展的今天，高校大学生的抄袭手段日益先进且日趋普遍，尤其是计算机专业的程序设计课程，抄袭现象更加严重。国外很多教育机构针对程序设计课程的源代码抄袭现象进行的调查显示，高达 85.4% 的学生承认抄袭过别人的编程作业<sup>[7]</sup>。相对于自然语言，程序语言语法非常规则，抄袭起来简单的多，完全不用理解程序，通过文本编辑器进行简单的变量替换、添加冗余代码、变换代码顺序等手段就可以改变代码的外观，且不影响程序的正常运行。

与此同时，全球计算机用户数量在不断增长，软件行业的发展达到了一个空前的规模。由于一些软件的功能的共通性(如教务管理管理软件等)<sup>[2]</sup>，使得在大型软件系统中发生抄袭、雷同现象时有发生，导致软件侵权案件屡有发生，给正当软件公司造成严重影响和危害。如不久前发生的百度搜索和 360 搜索的纠纷，有网友就通过分析网页源代码后指出 360 抄袭百度搜索的前端代码，如下图所示：



图表 1 360 抄袭百度前端代码



当人们逐渐形成知识产权保护意识过程时,开始实施相关防伪技术、识别技术甚至使用更先进的智能识别的自动化检测技术达到自主权的保护目的、提高识别效率。同样,计算机教学中也不例外的在不同程度上存在票窃他人作业或作品行为,最常见的现象就是在完成上机编程任务或编程训练考核中,被考核者抄袭他人程序代码后,稍作修改甚至不做任何修改便作为自主设计作品并提交,采用人工目测方法检测很难准确度量这种行为性质以及程度,尤其在计算机程序设计的上机考核成绩的评价中,对于此类问题除了完全抄袭行为外,由于在考核过程中存在人为主观性的影响,使得考核标准带有很大的模糊性和不确定性,在一定程度上限制了考核准确性和效率,导致增加考核真实性难度,降低考核成绩的可信度。由此引出需要解决计算机程序代码以下简称程序代码相似性的检测问题。

### 1.2.1 国外研究现状

程序代码的相似度检测的研究在国外起步较早,在 20 世纪 70 年代就有大量学者从事代码相似度检测的技术研究。1976 年, Purdue 大学的 Ottenstein<sup>[1]</sup>首次提出了应用属性计数法(attribute counting)获取相似度的方法,并用 Halstead 程序度量方法进行程序相似度计算,开发了一个用来检测 Fortran 程序相似度的系统。该方法的核心思想是提取代码的特征属性, Halstead 系统中提取了四个基于长度的软件科学参数,而后 Grier 和 Faidhi 分别使用了 20 和 24 种参数<sup>[3]</sup>来检测代码的相似性。这种方法的优点是效率高,但缺点也很明显,就是无法检测出对程序片段的抄袭,并且该方法之后的研究方向走入了死胡同,仅靠增加属性度量是无法提高系统的检测效果的。之后的学者就从属性计数的方法向基于结构度量的相似性检测方法发展。

1990 年, Komondoor 和 Howitz 提出的使用程序切片技术进行检测。通过构造出程序依赖图 PDG(Program Dependence Graph),然后使用文本比较方法,找到相似重复的子句。1992 年, B.S.Baker 通过将源代码中的函数名称、变量、常量、类型等各种标识符转换为参数化特征而提出了基于 Parameterized Matching 的重复代码检测工具 DUP,用于实现文件的比较<sup>[2]</sup>。

1996 年, Verco 和 Wise 指出对于仅仅使用属性计数法的算法,增加向量维数不能



改善正确率。改进属性计数法的措施就是加入程序的结构信息，应该结合结构度量(structure metrics)来检测相似度。基于结构度量的相似性检测方法主要是分析程序的结构信息以及执行流程。1998 年，D.Baxter 等提出了基于抽象语法树(Abstract Syntax Tree)的重复代码检测技术。这种检测技术对 C 语言程序进行语法解析，建立完备的语法树，随后应用三种算法进行重复代码检测。1998 年，Matthias Reiger 和 Stephane Ducasse 提出了一种独立于语言的相似代码检测方法，以及一个用于相似文件检测的工具 DUPLOC。除以上两类算法之外，还有学者提出了新的检测方法。比如基于案例推理的 Cogger 工具、基于神经网络的 Plague Doctor、基于优化编译与反汇编的 BuaaSim 和基于 XML 框架的 XPDec 等等。目前在网上提供代码相似性检测在线接口的工具有卡尔斯鲁厄大学的 JPlag、斯坦福大学的 MOSS、威奇塔州立大学的 Sim 和悉尼大学的 YAP3 等。这些工具普遍采用属性计数和结构分析相结合的方法进行代码相似度计算<sup>[1]</sup>。

### 1.2.2 国内研究现状

国内的相关研究较少，1988 年，中国人民警官大学的张文典和任冬伟研制了一个判断 Pascal 代码抄袭的系统<sup>[4]</sup>。该系统采用的是属性计数法，主要通过四大类属性：控制类、变量类、运算符类和标准过程的统计来进行相似度计算。

2007 年北邮的王继远也研发出自己的一套程序设计课程代码抄袭检测系统，是在现有结构度量法研究的基础上，在分析结构的过程中，将标记字符串描述为 XML 文档，通过 XML 文档结构化的树形结构来描述程序。其中 XML 描述了程序的结构信息，以 C 代码为例就是宏定义、全局变量、函数定义、函数实现等<sup>[5]</sup>。在函数实现内，按照变量、控制流程等划分下一层结构，在每一个控制流程内，通过提取初始及其执行部分来划分再下一层，以此类推直到无法继续分解，并提取为纯文本的元素。通过这样规一化的过程，进而通过两个代码规一化后的文本的公共长度以及两代码提交时间的时间戳进行计算。在建立 XML 结构的过程中可以通过控制粒度来对性能和准确率进行平衡。

北航的相似代码检测工具 BuaaSim<sup>[7]</sup>是我们学校设计的一套先进的系统。通过引入编译优化技术和反汇编技术，从程序语义层面消除类似代码冗余、语句拆分、控制结构等价替换等高级抄袭手段带来的干扰，并给出一个相似度经验阈值和一个基于该经验阈



值的简单有效的聚类算法，BuaaSim 系统便是基于该相似性检测方法的一个实现，应用在北京航空航天大学高级语言程序设计教学辅助平台<sup>[6]</sup>。北航的熊皓在 2010 年发表的硕士论文中<sup>[1]</sup>，详细阐述了基于静态词法树、基于 BP 神经网络、基于抽象语法树和基于 CIL、抽象语法树和串切片的四种算法，并最终实现了一个 C/C++代码相似性检测工具 BUAA\_AntiPlagiarism<sup>[1]</sup>。

### 1.3 研究目标与内容

研究目标：实现一个可对 C/C++代码集进行代码相似性检测、相似代码聚类的工具。

研究内容：

- ①需求分析：调研代码相似度检测工具功能、性能上的需求。
- ②算法实现：设计并实现一种或多种相似代码判定算法。
- ③性能测试：通过现有的通用数据集和实际情景试用，得出性能效率等结论。
- ④实用化：将实现的算法包装入可实用化的软件中。

### 1.4 课题来源

导师指定课题。

### 1.5 论文组织结构

论文组织结构如下：

第一章：介绍了课题的研究背景，对相关研究工作进行了分析和总结，明确了本文的研究目标和主要工作内容。

第二章：介绍并分析了与本项目相关的概念、系统实现过程中需要用到的 mingw 相关知识、当下主流的代码相似算法、各种基于生成的中间代码的相似度度量算法以及聚类算法。

第三章：介绍了系统需求分析与设计原则，详细分析了代码相似度算法的原理与设计，以及软件的整体系统设计。

---



---

第四章：在第三章设计的基础上，详细阐述了算法实现和相似代码检测软件的实现。

第五章：使用 BuaaSim 提供的测试数据对系统的准确率、召回率进行测试，并通过真实的编程比赛进行效果验证，说明本系统的实用性。

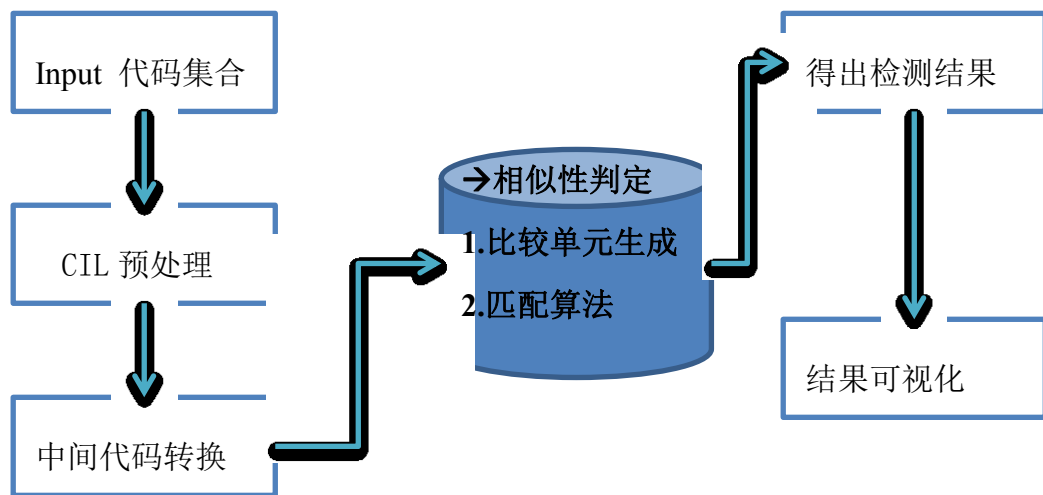
最后总结论文，对实现的程序代码相似性检测软件进行总结分析。

---

## 2 相关概念和技术

### 2.1 当今主流代码相似性检测模式

Whale 在研究过程中首次提出将检测过程分为两个阶段：代码格式转换和相似度确定。后来的很多检测方法都参考了这一框架，并将整个代码相似性检测的过程分为四个部分：预处理、中间代码转换、比较单元生成和匹配算法<sup>[3]</sup>。因此整体架构图如图表 1 所示：



图表 2 代码相似检测系统架构图

#### 2.1.1 预处理

预处理过程的目的是剔除一些与程序相似性比较无关的信息，尽可能地屏蔽那些简单抄袭手段对代码外观的影响。其操作通常包括：统一代码的布局，消除代码中一些无关字符等。通过调研现有工具得出，使用 CIL 进行 C 代码预处理的效果最好。CIL(C\_Intermediate\_Language)是对源程序进行编译和转化的一系列工具的集合<sup>[9]</sup>。它能够通过分析将 C 代码转化为等价语义的同源代码。

CIL 的转化过程主要包括了调整复杂的程序结构和删除部分冗余代码两个操作。其保留源代码的类型结构和等价语义的特性，决定了 CIL 生成的代码是有介于抽象语法树



结构和编译用中间代码之间的一种形式。使用 CIL 最大的优势是能够将任意可执行的 C 程序代码转化为一份等价语义的、具有很少控制结构的可执行代码，简化了编译和代码分析的流程。CIL 工具转化的 C 代码包括 ANSI-C 程序、Microsoft C 程序和 GNU C 程序<sup>[1]</sup>。

从本质上来说, CIL 是高耦合结构的 C 代码子集, 仅仅包含少量的语法形式。比如: C 程序代码中所有的控制结构将转换成统一的 Loop 结构、所有的函数都将会被加上返回语句 `return`、操作符 “`->`” 会被删除而且数组结构都会转化为指针类型, 这无疑降低了原始 C 程序代码中需要考虑的多类结构。此外, CIL 重要的操作是切分类型声明和调整函数结构, 这种转化后的程序具有更优的易读性和易编译性。直接计算程序代码一些表达式的值, 能够在很大程度上提升编译的效率。下图是 CIL 转换代码的举例说明:



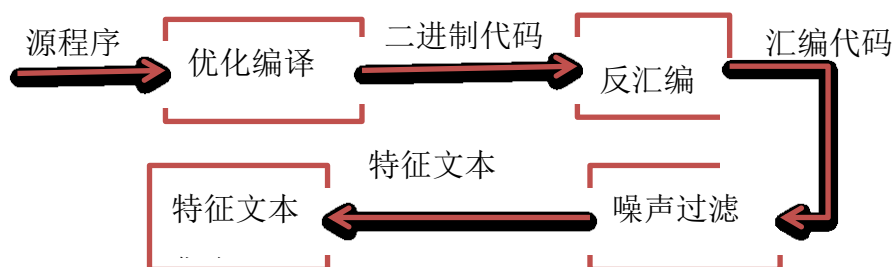
图表 3 CIL 转换代码效果图

### 2.1.2 中间代码转换

中间代码转换和预处理其实耦合度很高, 在很多地方都将预处理和中间代码转换统称为规一化。源代码的中间代码转换包括预处理和抽象语法树生成两个部分。通过部署 CIL 工具, 实施对源代码的预处理操作, 设置 CIL 参数使转换后的代码只包含三种控制结构(IF-Statement、WHILE(1)-Statement 和 GOTO-Statement)。该操作将所有的代码还原至格式统一、结构单一的原始代码, 一方面降低了相似性检测的难度, 另一方面简化了

抽象语法树的结构。

除了使用 CIL 之外,北航的赵长海<sup>[7]</sup>采用了基于编译优化和反编译的检测算法,其中在规一化的过程中先对源程序进行优化编译生成可执行二进制代码,而后反汇编获得汇编代码,再经过噪声过滤(如将偏移地址 jump 的地址统一替换为 OFFSET, call 的地址统一替换为 FUNCTION, push 的常量统一替换为 CONSTANT 等)后进行特征文本的相似计算,具体流程如图表 4:



图表 4 特征文本生成流程图

### 2.1.3 比较单元生成

比较单元的定义:将源代码的抽象表示切分成更细粒度的单位去与其它单位进行匹配以发现相似现象,切分后的单位称之为比较单元。比较单元是代码相似性检测工具发现相似现象的基本单位,工具无法检测到粒度小于一个比较单元的相似现象,因此比较单元的粒度直接关系到检测工具的检测精度。切分比较单元需要选用合适的粒度(固定或自由粒度)。固定粒度的值太大,会导致无法检测小于这个粒度的相似现象;粒度太小,检测结果失去意义。而自由粒度最主要问题是可能不合理地跨越程序的语法边界。在后续的检测流程中,可以通过一定的算法,去发现匹配的比较单元,然后利用匹配的结果进行聚合(即把位置连续的且互相匹配的比较单元聚合在一起),就能发现相似代码片段。

比较单元根据算法的不同,常见的有词、字符串、特征向量、树节点、子图结构等。本系统计划主要采用树结构的形式,使用 CDT 或 GCC 生成抽象语法树,再进行后续的匹配计算。具体的几种比较单元生成算法会在关键技术部分详细阐述。



2.1.4 匹配算法

根据所使用的源代码的抽象表示形式以及比较单元的不同，已有的相似性检测方法采用各式各样的匹配算法以发现相同或相似的比较单元对。如：字符串匹配、后缀树<sup>[10]</sup>、神经网络<sup>[11]</sup>等等。表格 2 是一些目前比较流行的、基于结构的代码相似检测系统及其匹配算法。

表格 1 主流代码相似检测系统及其匹配算法

编号	系统	算法
1	YAP3	RKR-GST
2	MOSS	String Matching
3	JPLAG	Gready String Tiling
4	Plague	LCS algorithm
5	GPLAG	Graph isomorphism algrithm
6	Bandit	LCS algorithm
7	SIM	Local Alignment

2.2 主流算法介绍

代码相似性检测技术从 1976 年发展至今，已经有大量成熟的技术方向可供参考，即使是最初的基于属性度量的方法也同样有适用的空间。本系统在尚未设计完成最终算法时，需要参考前人留下的宝贵财富，借以实现自己独到的高性能算法。本节主要介绍当今主流的五种程序代码相似性检测算法<sup>[3]</sup>。

2.2.1 基于属性度量

基于属性度量的算法是代码的相似性检测研究最初的算法，也是最容易实现、速度最快的算法。由于其对相似代码判定的直观性，当今的很多系统都是综合采用其他某种算法与基于属性度量的算法相结合来计算的。比如 Plague Doctor<sup>[12]</sup>提出从程序中提取 12 个度量值并组成特征向量，输入至 BP 神经网络中训练。北航的熊浩<sup>[11]</sup>也发表过一篇使用属性度量和神经网络的代码相似性检测方法，使用编译后反汇编代码的相似度量值、代码标识符序列化后的相似度量值以及代码风格的度量值作为神经网络的输入值进行训



练,实验结果表明此算法对数据结构修改但逻辑不变的代码召回率较低,而对于其他情况的性能较好。

### 2.2.2 基于串

基于串的检测是从源代码的文本结构及词法的角度去度量程序的相似度,不需要使用重量级的编译工具。方法支持多种编程语言甚至是纯文本文件的相似性检测。基于串的方法较树或图的检测思路具有更佳的时空效率,适用于检测大型软件系统的抄袭现象。基于串的计算方法多与基于树或基于图的算法相结合,将树或图结构转化成文本序列后,使用诸如 Rain-Karp 散列匹配算法、最长公共子串(Longest Common Substring)算法或最小编辑距离(Levenshtein Distance 或 Edit Distance)作为度量值作为相似系数或进行后续的聚类计算。

### 2.2.3 基于标识符

基于标识符的检测方法是通过将代码的各种标识符转化成特定的一些字符,从而使每一行代码可以作为一个字符串插入到各类数据结构中(如后缀树、字典树、AC 自动机[Aho-Corasick automation]等),从而可以在很短的时间内,在尽可能多的保留原文信息量的前提下找到相似的程序段。基于标识符的方法和基于串的方法相比,对于定位相似程序段更为方便,时间效率较高,但空间开销更大。

基于串的检测和基于标识符的检测在本质上并没有太多的区别,均可归结为基于文本的检测方法,可在自然语言文档复制检测的研究中找到雏形。一些方法在代码转化过程中同时涉及到了标识符和串结构,仅仅是研究者对于单个标识符在相似性判定的重要程度有所区别而已。基于标识符的检测无须依赖功能强大的编译器,检测算法时间复杂度低,同样适用于大型系统的检测。

### 2.2.4 基于树

基于树结构的算法大多首先提取抽象语法树。抽象语法树的生成的常用方法有两种,一种是 GCC(GNU Copiler Collection)中的 C 编译器可以通过调用如图表 6 所示的语句生



成抽象语法树文件:

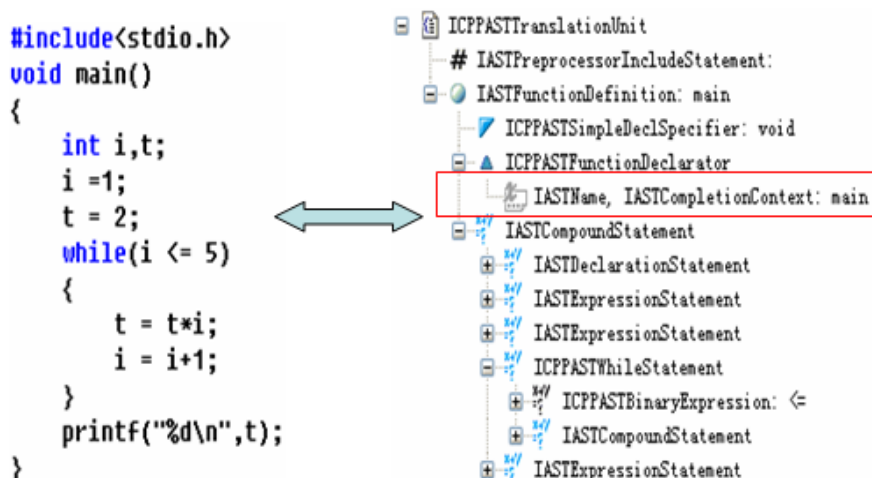
```
barty@barty-PC ~  
$ gcc -c -fdump-translation-unit -fno-builtin -ffreestanding x.c
```

图表 5 gcc 生成语法树代码

但由于生成后的语法树文件过于庞大(一个空 main 代码会生成 5816 行语法树结构代码), 因此并不方便作为相似性检测的数据进行计算。

另外一种方法是使用 Eclipse(一个开源的、基于 Java 的可扩展开发平台)中的 CDT(C/C++ Developing Tools)的语法分析功能, 可以获得粒度较大的抽象语法树。由于外部链接使 C/C++ 代码中不管在哪里声明一个变量或函数, extern 关键字都允许其他文件在不确定其定义位置的情况下使用该文件。因此, CDT 分析器在分析过程中将其语法信息存储于抽象语法树和持久文档对象(PDOM)模型中。语法信息的分离势必大大降低相似性检测的效率, 虽然通过语法树中节点可以计算出 PDOM 中的语法信息, 但是在检测过程中同时进行语法信息计算是相当低效的。

为了解决这个效率瓶颈, 使 CDT 转化出的抽象语法树包含尽可能多的语法信息。本章需要对 CDT 生成的抽象语法树中的语法信息进行重新构建。其主要操作是: 对 CDT 生成的原始语法树进行一次扫描, 利用通过 IAST-Name 查询所得的语法信息将语法树中相应节点所缺失的信息完整补充到语法树结构中。图表 7 是代码通过 CDT 生成以及信息补全后生成的语法树举例。



图表 6 代码及其对应的语法树

基于树结构的算法主要有两类，分别是对原树和对树结构串行化后的两种数据进行操作。有学者直接在语法树上寻找相似子树，使用公式  $\text{Similarity} = 2 * S / (2 * S + L + R)$  计算语法树的相似度，其中  $S$  为共有的节点， $L, R$  分别为两树中的不同节点<sup>[3]</sup>。因为以树为结构进行匹配会产生大量冗余信息且效率低下，因此多采用树结构串行化的方法来对提取出的抽象语法树进行相似判定。

### 2.2.5 基于图结构

基于图检测方法的基本思路和基于树型结构的方法类似。通过分析源代码的语法结构以及函数调用关系、控制依赖关系、数据流等，构建程序的依赖关系图(Program Dependence Graph, PDG)，匹配图中的结点，由这些结点组成的连通图被称为相似子图，由此判断代码的相似性。在该种检测思路中，图的结点和连接关系体现了程序的语法，而数据流可视为程序语义的抽象表达，因而具有较好的抄袭检测能力，且有可能检测到文本结构完全不同的抄袭代码。但是，通过静态分析工具建立代码的 PDG 图，所耗费的代价很高。图比树结构更为复杂，在其上寻找相似的子图的时空复杂度很高，故检测效率低<sup>[1]</sup>。



## 2.3 mingw 介绍

MinGW(Minimalist GNU for Windows), 是将 GCC 编译器和 GNU Binutils 移植到 Win32/64 平台下的产物, 包括一系列头文件(Win32(64)API)、库和可执行文件。对于 C 语言之外的语言, MinGW 使用标准的 GNU 运行库, 如 C++使用 GNU libstdc++。但是 MinGW 使用 Windows 中的 C 运行库。因此用 MinGW 开发的程序不需要额外的第三方 DLL 支持就可以直接在 Windows 下运行, 而且也不一定必须遵从 GPL 许可证。这同时造成了 MinGW 开发的程序只能使用 Win32API 和跨平台的第三方库, 而缺少 POSIX 支持<sup>[14]</sup>, 大多数 GNU 软件无法在不修改源代码的情况下用 MinGW 编译。

## 2.4 常用字符串相似度算法

字符串相似度计算是算法的核心所在, 主流代码相似检测系统的匹配算法也不尽相同, 如 YAP3 使用 RKR-GST 算法、JPLAG 中使用 Greedy String Tiling 算法、MOSS 中使用 String Matching、Plague 和 Bandit 使用 LCS 算法等。下文中对一些常用的字符串相似度算法进行介绍。

### 2.4.1 Longest Common Substring

最长公共子串(Longest Common Substring)问题就是求两个字符串的最长公共子串。我们称序列  $Z = \langle z_1, z_2, \dots, z_k \rangle$  是序列  $X = \langle x_1, x_2, \dots, x_m \rangle$  的子序列当且仅当存在严格上升的序列  $\langle i_1, i_2, \dots, i_k \rangle$ , 使得对  $j = 1, 2, \dots, k$ , 有  $x_{i_j} = z_j$ 。比如  $Z = \langle a, b, f, c \rangle$  是  $X = \langle a, b, c, f, b, c \rangle$  的子序列, 即子串。最长公共子串的标准解法是动态规划, 状态转移方程如下:

$$dp[i][j] = \max \begin{cases} dp[i-1][j] & \text{for } i \geq 1 \\ dp[i][j-1] & \text{for } j \geq 1 \\ dp[i-1][j-1] + 1 & \text{for } a[i] = b[j] \text{ and } i \geq 1 \text{ and } j \geq 1 \end{cases}$$

通过以上方程就可得到两个串的最长公共子串, 时间复杂度  $O(NM)$ , 空间复杂度  $O(NM)$ , 由于此问题的特殊性, 在保存状态时只需要保存矩阵中上一行的所有元素, 因



此空间复杂度可以减少至  $O(N)$ 。在求得最长公共子串的长度  $result$  后,可以以此来计算两个字符串的相似度,公式如下:

$$\text{similarity} = \frac{result}{\min(len1, len2)} * 100\%$$

#### 2.4.2 Levenshtein Distance

搞自然语言处理的应该不会对这个概念感到陌生,编辑距离(Levenshtein Distance)就是用来计算从原串  $s$  转换到目标串  $t$  所需要的最少的插入,删除和替换的数目,在 NLP 中应用比较广泛,同时也常用来计算你对原文本所作的改动数。我们借用这个思想,用来计算两个字符串的相似度。计算编辑距离依旧通过动态规划思想来实现,状态转移方程如下:

$$dp[i][j] = \begin{cases} i & \text{for } j = 0 \\ j & \text{for } i = 0 \\ \min \begin{cases} dp[i-1][j] + 1 \\ dp[i][j-1] + 1 \\ dp[i-1][j-1] + (a[i] \neq b[j]) \end{cases} & \text{for } i > 0 \text{ and } j > 0 \end{cases}$$

计算出编辑距离后,相似度的公式和 LCS 的类似,如下:

$$\text{similarity} = \frac{\text{distance}}{\min(len1, len2)} * 100\%$$

#### 2.4.3 Rabin-Karp\_Matcher

假定字符集  $\Sigma = \{0, 1, 2, \dots, 9\}$ , 每一个字符对应一个十进制数字(更一般情况,假定每个字符是基数为  $d$  的表示法中的一个数字,  $d = |\Sigma|$ ), 可以用一个长度为  $k$  的十进制数字来表示由  $k$  个连续字符组成的字符串<sup>[15]</sup>。

例如,字符串 "31415" 对应于十进制数 31415。

已知模式  $P[1..m]$ , 设  $p$  表示其相应十进制数地值, 类似地, 对于给定的文本  $T[1..n]$ . 用  $ts$  表示长度为  $m$  的子字符串  $T[s+1 \dots s+m]$  ( $s = 0, 1, \dots, n-m$ ),  $ts = p$  当且仅当  $[s+1..s+m] = P[1..m]$ ; 因此  $s$  是有效位移当且仅当  $ts = p$ 。类似地, 可以在  $\Theta(m)$  时间内,





根据  $T[1..m]$  计算出  $t_0$  的值。如果能在总共  $\Theta(n - m + 1)$  时间内计算出所有的  $t_s$  的值, 那么通过把  $p$  值与每个  $t_s$  (有  $n - m + 1$  个) 进行比较, 就能够在  $\Theta(m) + \Theta(n - m + 1) = \Theta(n)$  时间内求出所有有效位移。

为了在  $\Theta(n - m)$  时间内计算出剩余的值  $t_1, t_2, \dots, t_{n-m}$  可以在常数的时间内根据  $t_s$  计算出  $t_{s+1}$ , 先看例子, 假如  $m = 5$ ,  $t_s = 31415$ , 我们去掉高位数字  $T[s + 1] = 3$ , 然后在加入一个低位数字  $T[s + 5 + 1]$  (假设为 2), 得到:  $t_{s+1} = 10(31415 - 10000 * 3) + 2 = 14152$ 。如果预先计算出  $10^{m-1}$  (在这里只需简单地在  $O(m)$  时间内计算出就可以), 那么就可以在常数时间计算出  $t_{s+1}$ 。因此, 可以在  $\Theta(m)$  时间内计算出  $p$  和  $t_0$ 。然后在  $\Theta(n - m + 1)$  时间内计算出  $t_1, \dots, t_{n-m}$  并完成匹配。

## 2.5 常用聚类算法

### 2.5.1 k-Nearest Neighbor algorithm(KNN)

K 最近邻(k-Nearest Neighbor, KNN)分类算法, 是一个理论上比较成熟的方法, 也是最简单的机器学习算法之一。该方法的思路是: 如果一个样本在特征空间中的  $k$  个最相似(即特征空间中最邻近)的样本中的大多数属于某一个类别, 则该样本也属于这个类别。KNN 算法中, 所选择的邻居都是已经正确分类的对象。该方法在定类决策上只依据最邻近的一个或者几个样本的类别来决定待分样本所属的类别。KNN 方法虽然从原理上也依赖于极限定理, 但在类别决策时, 只与极少量的相邻样本有关。由于 KNN 方法主要靠周围有限的邻近的样本, 而不是靠判别类域的方法来确定所属类别的, 因此对于类域的交叉或重叠较多的待分样本集来说, KNN 方法较其他方法更为适合。在实际应用中, 为提升效率多使用 KD-Tree 来存储  $N$  维空间向量, 对于本系统来说空间消耗过大, 并不适合因此不再赘述。

### 2.5.2 k-means

K-means 算法是硬聚类算法, 是典型的局域原型的目标函数聚类方法的代表, 它是数据点到原型的某种距离作为优化的目标函数, 利用函数求极值的方法得到迭代运算的



调整规则。K-means 算法以欧式距离作为相似度测度,它是求对应某一初始聚类中心向量  $V$  最有分类,使得评价指标  $J$  最小。算法采用误差平方和准则函数作为聚类准则函数。

$k$  个初始类聚类中心点的选取对聚类结果具有较大的影响,因为在该算法第一步中是随机的选取任意  $k$  个对象作为初始聚类的中心,初始地代表一个簇。该算法在每次迭代中对数据集中剩余的每个对象,根据其与其各个簇中心的距离将每个对象重新赋给最近的簇。当考察完所有数据对象后,一次迭代运算完成,新的聚类中心被计算出来。如果在一次迭代前后,  $J$  的值没有发生变化,说明算法已经收敛。

算法的流程如下:

- 1) 从  $N$  个文档随机选取  $K$  个文档作为质心
- 2) 对剩余的每个文档测量其到每个质心的距离,并把它归到最近的质心的类
- 3) 重新计算已经得到的各个类的质心
- 4) 迭代 2~3 步直至新的质心与原质心相等或小于指定阈值,算法结束

考虑到本算法需要指定  $K$  个质心,而对于实际的代码抄袭检测问题来说,并没有一个方法能够找到合适的质心,因此这个算法也无法适用本系统。由于常见的两种聚类算法都不适合代码相似性检测系统的使用,我设计了一个更为简单、高效的聚类算法,详见后文中的算法设计部分。

## 2.6 本章小结

本章主要介绍了程序代码相似性检测系统所用到的和已知的相关技术。介绍了当今主流的代码相似性检测模式和技术、字符串计算中常用的高级数据结构、常见的字符串相似度算法以及常见的基于文本的聚类算法。

---



### 3 系统与算法设计

#### 3.1 PlagiarismChecker 系统设计

##### 3.1.1 系统需求分析与设计原则

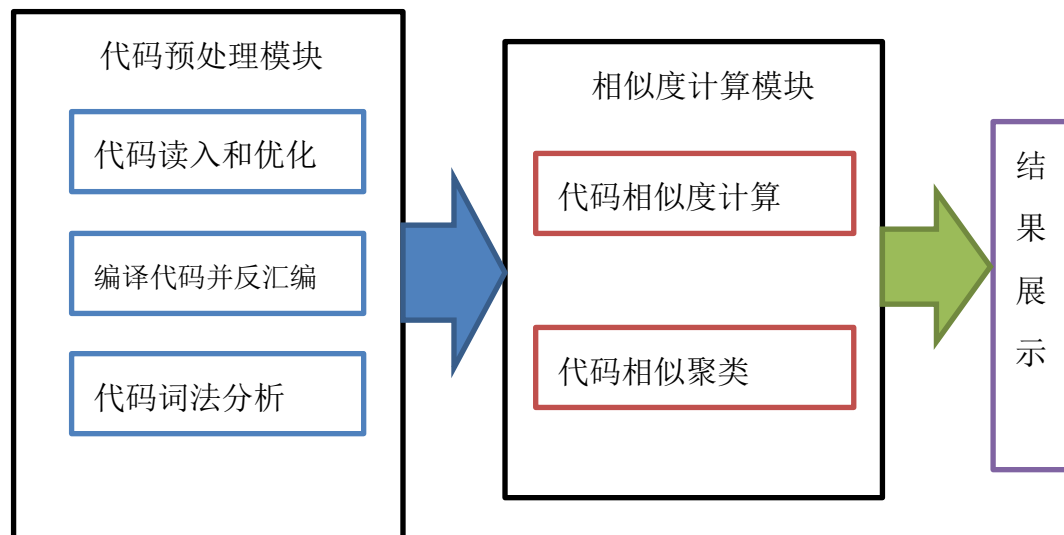
2001 年 Edward L.Jones 将程序相似代码之间的修改方法重新总结,在不影响程序结果的前提下,将相似代码分为十类情况<sup>[8]</sup>,如下表所示:

表格 2 常见代码抄袭手段

No.	分类名称	No.	分类名称
1	逐字拷贝	6	改变代码块顺序
2	更改注释语句	7	改变表达式的操作符和操作数顺序
3	更改空白字符	8	改变代码块中语句的顺序
4	重新命名标识符	9	增加语句和变量
5	更改数据类型	10	用等价语句替换原有语句

对于企业级的代码抄袭,由于代码量巨大因此抄袭手段多为最简单的几种方法,而对于高校学生间程序设计作业的代码抄袭,则相对复杂一些。因此准确率和召回率的计算取决于对于中短代码、高技术性抄袭的判定性能。特别的,对于这类系统来说,召回率的重要性要远远大于准确率。因此项目的根本目标是相对各类现有 C 语言代码相似性检测系统,有更高的召回率。

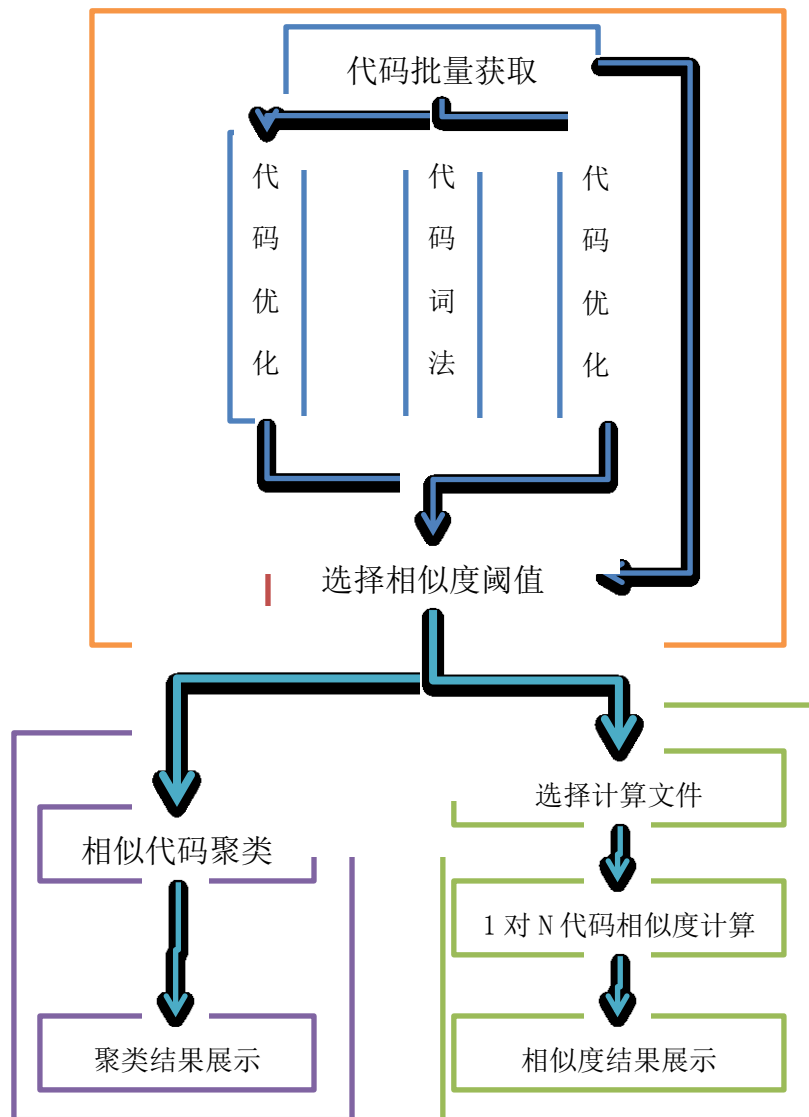
### 3.1.2 系统模块设计



图表 7 代码相似性检测系统模块设计

系统主要分为代码预处理模块、相似度计算模块和结果展示模块三部分。在代码预处理模块中，主要有代码的批量读入和优化、编译代码及反汇编、代码的词法分析三部分模块。在相似度计算模块中，主要有代码的相似度计算和代码的相似聚类两部分。对于批量载入的代码，首先经过预处理模块进行预处理，生成可用于计算的中间代码，而后再对其进行相似度的各种计算，最后将代码间的相似度结果、以及代码相似聚类后分组的结果通过可视化工具展示出来，供用户进行更为精确的分析判断。

### 3.1.3 功能模块设计



图表 8 代码相似性检测系统功能模块

## 3.2 算法设计

### 3.2.1 代码相似度计算

考虑这个系统的特殊性，我设计了一种基于 RK 字符串匹配思想的两字符串相似度计算算法。算法流程如下：



step 1. 删除所有空白字符

step 2. 设置一个合适的步长(设置为 5-10 性能最优)

step 3. 对所有相邻步长长度的字符串进行 Hash, 保存至一个结果队列中

step 4. 对两个结果队列中的 Hash 值进行排序, 计算其中的公共元素数量 simHash

step 5. 计算公式为  $\text{similarity} = \frac{\text{Simhash}}{\min(\text{len1}, \text{len2})} * 100\%$

算法的伪代码如下:

Function GetSim(string a, string b)

```
1. hash_a ← gethash(a)
2. hash_b ← gethash(b)
3. i ← 0
4. j ← 0
5. loc ← 0
6. for i ← 0 until len_a - 1 do
7.   begin
8.     while loc < len_b and b[loc] < a[i] do
9.       begin
10.        loc ← loc + 1
11.      endwhile
12. if (loc >= len_b) break
13.   if (b[loc] == a[i])
14.     begin
15.       Simhash ← Simhash + 1
16.       loc ← loc + 1
17.     end
18.   endifor
19. return Simhash / min(len_a, len_b)
```



算法的时间复杂度为  $O(\log N)$ ，通过这种算法相比 LCS、Levenstein Distance 等方法有更高的时空效率和更高的召回率，尤其是对于通过代码块重排顺序的抄袭方法有非常突出的表现。

### 3.2.2 代码相似聚类算法设计

对于相似代码聚类这个问题，与其他聚类问题有较大区别。首先，代码相似聚类并没有明确的重心，而是类似链式的抄袭传播链，这就需要更多的考虑代码之间的间接关系而非直接联系(如抽象出的向量之间的欧式距离等)。其次，在确定一个相似阈值以后，不同代码之间的相似是具有传播性的，也就转化成类似图论中的传递闭包问题，让每个检测出来的集合最大化即可。因此设计的聚类算法并不使用常见的 KNN 算法，也不用传统的 KD-Tree 等数据结构，而是使用**并查集**来实现聚类算法。

并查集是一种树型的数据结构，用于处理一些不相交集合(Disjoint Sets)的合并及查询问题。常常在使用中以森林来表示。需要注意的是，一开始我们假设元素都是分别属于一个独立的集合里的。合并两个不相交集合操作很简单：先设置一个数组 `Father[x]`，表示 `x` 的“父亲”的编号。那么，合并两个不相交集合的方法就是，找到其中一个集合最父亲的父亲(也就是最久远的祖先)，将另外一个集合的最久远的祖先的父亲指向它。合并的伪代码如下：

```
Function Union(int x, int y)
```

```
1.  $fx \leftarrow \text{GetFather}(x)$   
2.  $fy \leftarrow \text{GetFather}(y)$   
3. if  $fx = fy$  then  
4.   return FALSE  
5. else  
6.   begin  
7.      $fa[fx] \leftarrow fy$   
8.   return TRUE  
9. end
```



判断两个元素是否属于同一集合仍然使用上面的数组。则本操作即可转换为寻找两个元素的最久远祖先是否相同。可以采用递归实现。查询的伪代码如下：

```
Function Same(int x, int y)
```

1. if GetFather(x)  $\neq$  GetFather(y) then
2.     return TRUE
3. else
4.     return FALSE

获取一个元素的祖先需要用到路径压缩优化以便保证查询和合并的时间复杂度。如果一个节点的直接父亲不是它自己那么意味着它是自己的祖先，否则将它直接父亲的祖先赋值给自己，作为自己新的直接父亲节点，于是查询和插入的期望平摊时间复杂度均为  $O(N)$ 。获取祖先的伪代码如下：

```
Function GetFather(int node)
```

1. if fa[node]  $\neq$  node then
2.     fa[node]  $\leftarrow$  GetFather(fa[node])
3. return fa[node]

在并查集算法的基础之上，通过之前的相似度计算模块，可以获得相似聚类的结果。声明并查集变量 set 用来记录聚类结果。相似聚类的伪代码如下：

```
Function Cluster()
```

1. set.init()
2. for i  $\leftarrow$  0 until n - 1 do
3. begin
4.     for j  $\leftarrow$  i + 1 until n - 1 do
5.         if similar(text[i], text[j]) then set.Union(i, j)
6. end
7. return set





---

### 3.3 小结

本章首先介绍了 PlagiarismChecker 的需求分析与设计原则,并给出了整体的系统模块设计和功能模块设计,确定了工作的正确路线。

随后介绍了为系统设计的代码相似度算法,在优先召回的基础原则之上,根据 Rabin-Karp 算法的基础提出了一个时空复杂度均很优秀的文本相似度计算算法,并给出伪代码。

最后创新地提出了一个基于并查集的相似代码聚类算法,在对并查集操作进行介绍的基础上给出了聚类算法的伪代码。

---



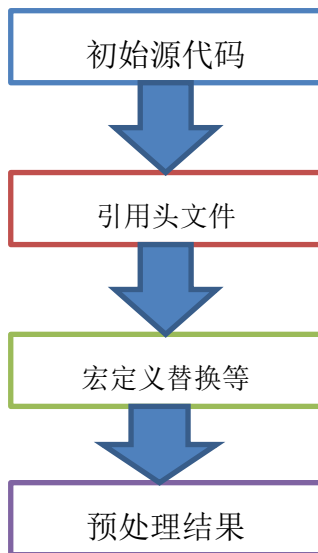
## 4 系统实现

本章主要介绍了相似代码检测系统，包括代码相似度计算算法和代码相似聚类算法的实现。基于上一章描述的系统设计和算法设计，使用 C# 的 Form 框架实现了在 Windows 环境下运行的、较易使用的相似代码检测软件。

### 4.1 软件功能实现

#### 4.1.1 代码优化

首先对源代码进行优化，通过调用 gcc/g++ 的运行参数来实现完整的 include 过程，并对 define 的表达式进行正则计算，使代码变成无 include、无 define 的一份简单代码，完美解决通过添加宏定义来实现代码修改的抄袭方法。



图表 9 代码优化预处理流程

引用头文件的原理很简单，将源代码中以“#include”开头的语句后紧跟的标准库内头文件完整拷贝至当前代码，并将未出现过的代码进行剔除，以实现头文件的引用。除引用文件之外，宏定义也是 C/C++ 语言允许宏带有参数。在宏定义中的参数称为形式参数，在宏调用中的参数称为实际参数。对带参数的宏，在调用中，不仅要宏展开，而且



要用实参去代换形参。带参宏定义的一般形式为:

```
#define 宏名(形参表) 字符串
```

在字符串中含有各个形参。带参宏调用的一般形式为: 宏名(实参表)。例如:

```
#define M(y) y*y+3*y /*宏定义*/
```

```
k=M(5); /*宏调用*/
```

在宏调用时, 用实参 5 去代替形参 y, 经预处理宏展开后的语句为:

```
k=5*5+3*5
```

实际的源代码优化的效果图如下, 其中代码经过自己实现的工具进行了代码高亮:



```
1 #include <stdio.h>
2 #define max(x, y) ((x)>(y)?(x):(y))
3 int main() {
4     int s = 0, i;
5     for (i = 1; i <= 10; ++i)
6         s += i;
7     printf("%d\n", max(5050, s));
8     return 0;
9 }
```

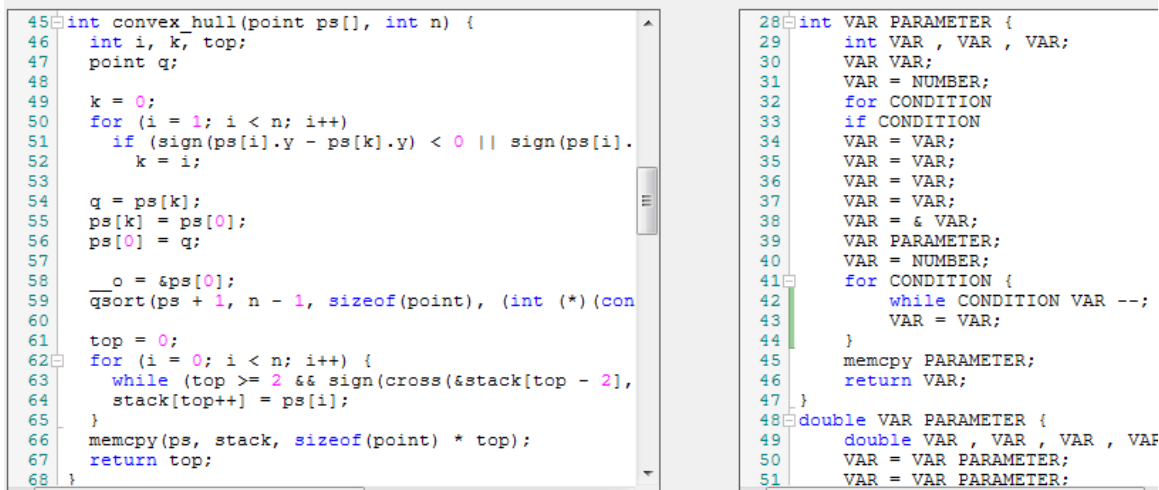
```
1 int main() {
2     int s = 0, i;
3     for (i = 1; i <= 10; ++i)
4         s += i;
5     printf("%d\n", ((5050)>(s)?(5050):(s)));
6     return 0;
7 }
```

图表 10 源码优化效果图

#### 4.1.2 词法分析

通过对 C/C++的词法进行研究, 实现了一个能对 C/C++代码进行词法分析的工具。

具体效果如下:



```
45 int convex_hull(point ps[], int n) {
46     int i, k, top;
47     point q;
48
49     k = 0;
50     for (i = 1; i < n; i++)
51         if (sign(ps[i].y - ps[k].y) < 0 || sign(ps[i].
52             k = i;
53
54     q = ps[k];
55     ps[k] = ps[0];
56     ps[0] = q;
57
58     o = &ps[0];
59     qsort(ps + 1, n - 1, sizeof(point), (int (*)(con
60
61     top = 0;
62     for (i = 0; i < n; i++) {
63         while (top >= 2 && sign(cross(&stack[top - 2],
64             stack[top++] = ps[i];
65     }
66     memcpy(ps, stack, sizeof(point) * top);
67     return top;
68 }
```

```
28 int VAR_PARAMETER {
29     int VAR, VAR, VAR;
30     VAR VAR;
31     VAR = NUMBER;
32     for CONDITION
33     if CONDITION
34     VAR = VAR;
35     VAR = VAR;
36     VAR = VAR;
37     VAR = VAR;
38     VAR = &VAR;
39     VAR PARAMETER;
40     VAR = NUMBER;
41     for CONDITION {
42         while CONDITION VAR --;
43         VAR = VAR;
44     }
45     memcpy PARAMETER;
46     return VAR;
47 }
48 double VAR_PARAMETER {
49     double VAR, VAR, VAR, VAR;
50     VAR = VAR PARAMETER;
51     VAR = VAR PARAMETER;
```

图表 11 词法分析效果图

从图中可以看出, 在词法分析的过程中, 将很多常见的混淆部分进行了替换, 如表



达式、条件语句、自定义标识符等等，替换规则如下：

表格 3 词法分析中的替换规则

源代码内容	代码段举例	词法分析结果
计算表达式	$2+(3*a+b)$	NUMBER
条件表达式	$(i = 0; I < n; ++i)$	CONDITION
变量	$num[x]$	VAR
函数参数列表	$(int\ x, int\ y, int\ z)$	PARAMETER
其他	while	while

通过这样替换，能够使得对于变量名替换、表达式等价替换这种方式修改代码的抄袭方法有更高的召回率，实测效果非常好。

#### 4.1.3 编译优化反汇编

编译优化结果的生成和反汇编通过 gcc/g++ 来进行，方法如下：

表格 4 调用 mingw 进行编译优化反汇编

C 语言方法
<code>gcc -c -O3 file.c -o file.o</code>
<code>gcc -d file.o &gt; file.a</code>
C++语言方法
<code>g++ -c -O3 file.cc -o file.o</code>
<code>g++ -d file.o &gt; file.a</code>

在生成初步的汇编代码后，对其中无关逻辑的部分进行等价替换，增加召回率。具体的替换规则如下：

表格 5 汇编代码优化中的替换规则

原始内容	举例	转换为
函数堆栈声明	<code>00000000 &lt;__Z4nodeiiii&gt;:</code>	startfunc
无条件跳转指令	<code>jmp 1a98</code>	jmp offset



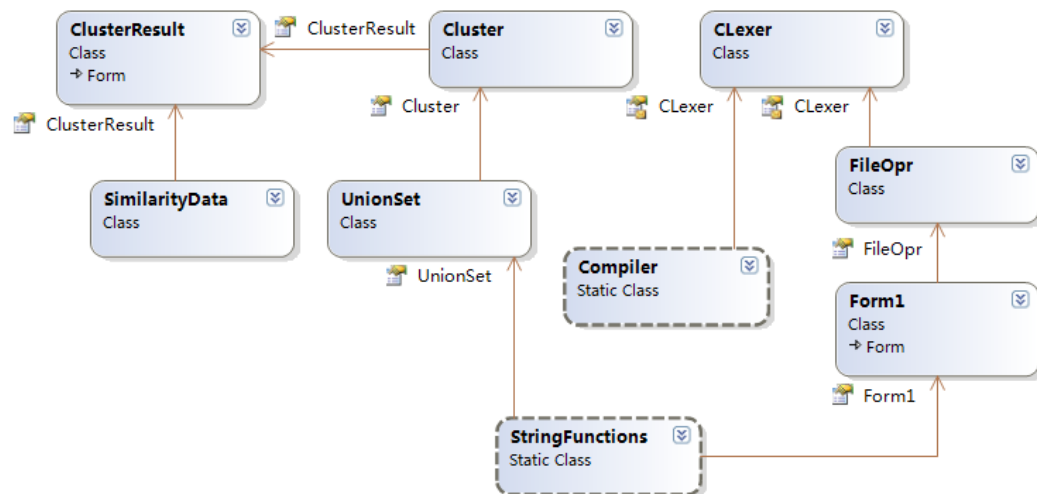
跳转子程序指令	call 1d0e <_main+0x2ba>	call func
大于/小于	jl/jg	jg
大于等于/小于等于	jle/jge	jge
压栈/出栈指令	push %ebp	push const
条件跳转指令	je 1c38 <_main+0x1e4>	je 1c38<func>
堆栈空间名称	xchg %ax,%ax	xchg var,var

生成代码的 demo 效果如下，可以看出优化的程度还是很高的，对于左侧代码中的从 1 到 10 求和，直接在汇编代码中体现为\$0x37，即 16 进制的 55。通过编译优化的代码可以将很多人为复杂化的代码统一简化，就更容易从逻辑层面进行相似度计算。

<pre>1 #include &lt;stdio.h&gt; 2 #define max(x, y) ((x)&gt;(y)?(x):(y)) 3 int main() { 4     int s = 0, i; 5     for (i = 1; i &lt;= 10; ++i) 6         s += i; 7     printf("%d\n", s); 8     return 0; 9 } 10</pre>	<pre>1 function 2 push    %ebp 3 mov     %esp,%ebp 4 and     \$0xffffffff0,%esp 5 sub     \$0x10,%esp 6 call    function 7 movl    \$0x37,0x4(%esp) 8 movl    \$0x0,(%esp) 9 call    function 10 xor     %eax,%eax</pre>
--	--

图表 12 汇编优化生成代码效果图

#### 4.1.4 代码结构设计



图表 13 相似检测系统代码结构图

其中 Compiler 类和 StringFunctions 类是两个静态类,分别实现了与调用编译器相关的函数以及字符串的各种处理、以及字符串的各种相似性计算功能的函数。UnionSet 实现了一个并查集的数据结构, SimilarityData 和 CluterResult 都是用来保存相似结果的数据结构,并在 Cluster 中调用各种数据结构来实现代码相似聚类。在 CLexer 中实现了一个完整的 C 语言高级词法分析器,并最终通过 Form 框架中的各种调用来实现整个软件的功能。

#### 4.2 算法实现

算法的具体实现参照之前的算法设计部分,将算法的伪代码通过 C#语言实现,并封装成类用于软件中的计算模块。根据需求的不同,通过不断试验得出三个阈值,并通过测试一份 BuaaSim 提供的通用数据验证了算法的有效性。

##### 4.2.1 代码相似度算法实现

首先调用 gethash 方法,将两个文本的 hash 序列算出,并按照递增顺序排列。

```
x = gethash(text1, q), y = gethash(text2, q);
```



```
Array.Sort(x), Array.Sort(y);
```

在计算完哈希序列之后,按照维护两个指针的方法将两个序列中的公共部分统计出来,得出公共哈希元素的数量 `totsame`。

```
for (int i = 0; i < lenx; ++i) {  
    while (locy < leny && y[locy] < x[i]) locy++;  
    if (locy >= leny) break;  
    if (y[locy] == x[i]) ++totsame, ++locy;  
}
```

最后按照下面的公式,用 `totsum` 和两个文本的长度计算出相似度结果。

```
return ans = (double)(totsame) / (double)Math.Min(lenx, leny);
```

其中 `gethash` 为 Rabin-Karp 算法的核心部分,主要是按照算法流程获得一个 hash 后的数组,再进行后续工作。`gethash` 的代码如下,首先预处理第一段步长 `q` 的哈希值:

```
for (int i = 0; i < q; ++i)  
    res[0] = (res[0] * 2 + text[i]) % Consts.BIGPRIME;
```

其中 `BIGPRIME` 选用的是 64 位 `int` 的标准大质数 111111111111111111。而后通过前一个步长 `q` 的哈希值,推算出下一个步长 `q` 的哈希值,最后将数组全部计算出:

```
for (int i = q; i < len; ++i)  
    res[i - q + 1] = ((res[i - q] - text[i - q] * (1 << (q - 1))) * 2 + text[i]) % Consts.BIGPRIME;
```

这两个函数的计算过程中都有一个参量 `q`,表示在进行 Rabin-Karp hash 的过程中用到的步长。经验表明,步长设定为 5-10 时综合性能较好,步长过大时无法发挥这个算法灵活多变的特点,召回率较低,特别的,针对汇编语言代码的相似度,前人很多算法计算的颗粒度均为整行,这也是步长的上限了;步长过短时,准确率会大幅降低,考虑极限情况有对[0..255]这些字符串中的字符进行桶计数作为余弦向量,再进行代码聚类的,时间性能出众,但准确率较低,只适合超大规模代码相似计算。通过实验得出步长为 10 的时候最适合代码的相似度计算。上述算法的时间复杂度为  $O(N\log N)$ ,空间复杂度  $O(N)$ ,是非常优秀的算法。

#### 4.2.2 代码相似聚类算法实现

代码相似聚类算法如前一章所述,实现了一个基于并查集的相似代码聚类算法。算法的第一步显然是初始化并查集:



```
graph = new UnionSet(size);
```

而后进行遍历, 算出所有代码间的相似度结果, 并将相似度大于阈值的两份代码所在的集合合并。考虑到时间效率的问题, 我对相似度的计算次数进行了优化, 使得运行时间是原来的二分之一, 效率提升非常明显。具体代码如下:

```
for (int i = 0; i < size; ++i) {  
    baseForm.setProgressBar((int)(i * 1000 / size));  
    List<SimilarityData> edge = calTheFile(files[i], i + 1);  
    for (int j = i + 1; j < size; ++j)  
        if (edge[j].isSimilar(threshold)) graph.mergeUnionSet(i, j);  
}
```

最后通过集合运算, 将所有元素个数大于 1 的集合整理并以交互界面的形式呈现给用户, 具体的界面设计在 ClusterResult 中有所体现, 考虑到与本算法并无太大关联, 就不再赘述交互展示的代码, 具体的交互界面会在下一章中有所体现。下面是集合运算部分的代码:

```
for (int i = 0; i < size; ++i)  
    if (countResult[i].Count > 1 && countResult[i].Count < Consts.MAXSIMGROUPSIZE) {  
        for (int j = 0; j < countResult[i].Count; ++j)  
            temp.Add(files[countResult[i][j]]);  
        clusterResult.Add(temp);  
    }  
ClusterResult newForm = new ClusterResult(clusterResult);
```

通过如上的代码, 便在  $O(N^2)$  的时间复杂度和  $O(N)$  的空间复杂度内, 将相似代码的聚类结果返回给用户。此部分通过设置不同的相似度阈值, 可以获得不尽相同的效果, 将阈值设低则可以将逻辑相似的代码聚类起来, 亦即作业或比赛中相同题目的代码; 将阈值设为与代码相似算法的阈值相同则可以找出互相抄袭的小团队, 作为平时课程作业考核的重要参考。

### 4.3 小结

本章首先介绍了基于上一章的系统设计的 UI 交互设计界面、以及相似度、相似聚类算法的具体实现。本系统的实现遵循了前文提出的设计原则, 并达到了目标设计要求, 达到了预期效果。实验结果及分析将在第五章进行阐述。







## 5 系统试验设计与分析

本章主要针对需要代码抄袭检测的各种场景,设计具体的实验对本文中的程序代码相似性检测系统进行了性能分析,并和现有算法的准确率、召回率、时空效率进行对比,验证了其优势。以下所有测试的工作环境为 Windows7 64 位操作系统,处理器为 Intel Core i7-2620M CPU 2.7GHz,内存为 4.00GB。

### 5.1 基本功能测试

基本功能测试主要从用户的角度出发,对各种可能的操作、各种可能的输入数据情况进行功能上的测试,以确保作为一个软件可以正常平稳运行。

#### 5.1.1 代码载入

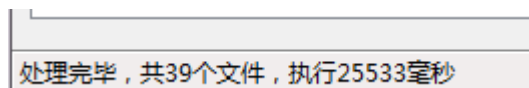
代码载入的功能一共有两个,一是将指定文件夹及其子文件夹下指定的文件(本系统中指扩展名为 c/c++/cpp/cc/cxx 的所有文件)通过调用系统的预处理功能生成源代码、与处理后的源代码、词法分析后的代码以及汇编代码四类结果,二是清除软件所在目录下的所有临时文件,以减少硬盘空间开销。新载入的文件若之前已计算结束则不再执行新的预处理过程,减少了重复计算的次数,极大的提升了效率。代码载入的效果如下:

---



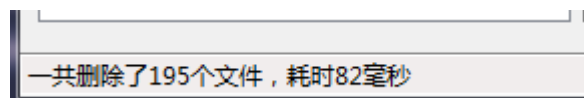
图表 14 代码载入效果图

在载入过程中会在左侧列表中动态显示已经预处理完毕的文件列表，并在下方的进度条显示完成度。载入的时间效率如下：



图表 15 代码载入时间效率

结果显示，一共预处理 39 个文件，平均预处理每个文件需要 654 毫秒。清除缓存的时间效率如下：



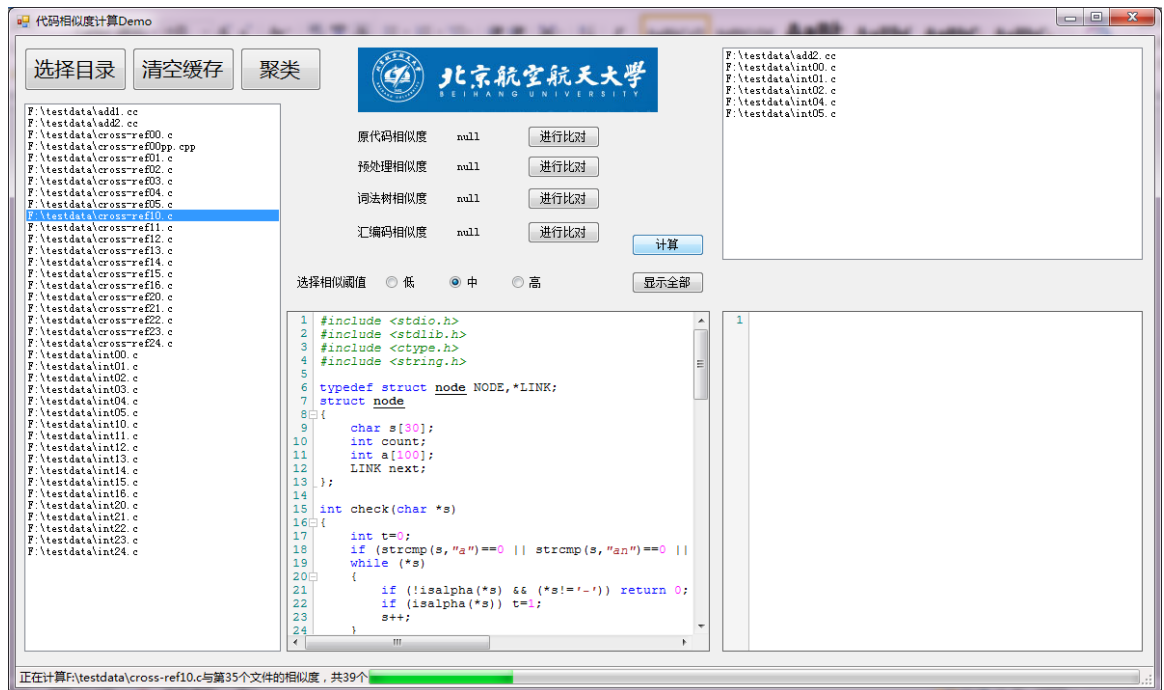
图表 16 代码预处理时间效率

结果显示，对于之前预处理产生的 195 个文件，删除文件的效率还是很高的，平均每个文件只需要耗时 0.42 毫秒。

### 5.1.2 代码相似度计算和展示

在单击选择左侧文件列表中的文件后，在左侧代码框中会显示出对应的源代码。选

择相似阈值、单击右侧的计算按钮后,经过一系列计算,会在右侧的列表中显示出和左侧选中代码疑似相似的代码列表,如下图所示为计算过程中的截图:



图表 17 代码相似度计算过程图

计算过程中依旧会显示进度状态,并且右侧的列表会动态更新,显示疑似抄袭代码列表。单击右侧列表后,选择对比的内容按钮,会在右边的代码框中显示疑似相似的代码对应的内容,如选择词法分析结果的对比,效果图如下:

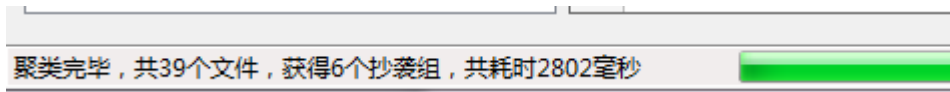


图表 18 疑似相似代码词法分析结果对比效果图

可以看出运行效率还是很出众的，与 39 份代码进行相似度计算仅耗时 335 毫秒。并且对比两份词法分析的结果可以发现，即使源代码重合度较低，词法分析的相似度计算结果也有可能非常高，甚至接近 100%。

### 5.1.3 相似代码聚类 and 展示

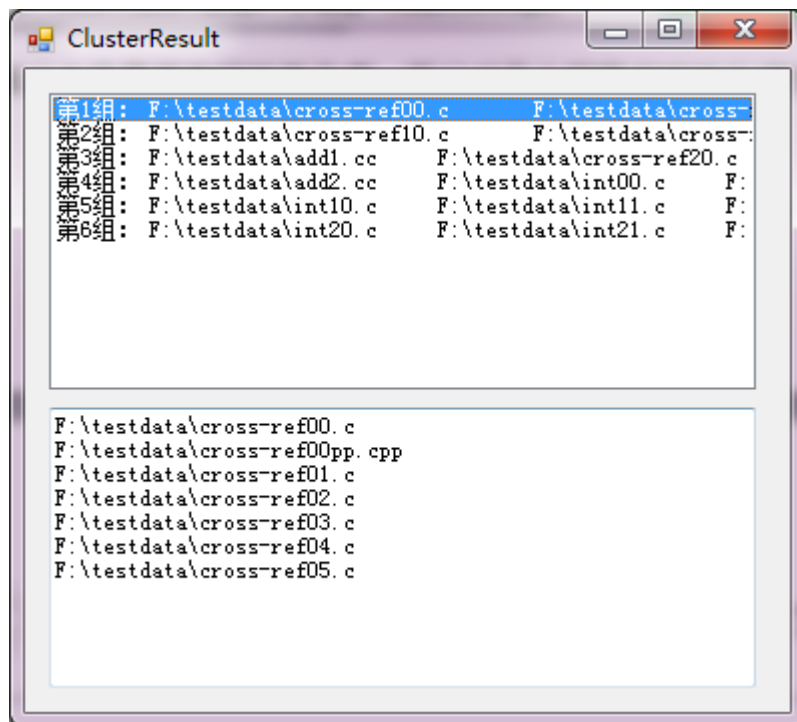
当已经选择了一个指定的文件夹，并对文件夹下所有相关文件进行预处理之后，单击左上方的聚类按钮，会提示进行聚类计算，并在下方显示进度条。在计算完成后，会显示聚类的结果和耗时，如下图：



图表 19 相似代码聚类时间效率

聚类算法由于是平方级的复杂度，所以对于大规模代码的处理效果不是太好，但是对于绝大多数日常作业的抄袭检测已经足够，39 份代码总共耗时不到 3s。聚类完成后，

会弹出一个新的窗口，如下图所示：



图表 20 相似代码聚类结果集合展示

在上方的列表中会依次显示聚类的集合列表，每个集合至少有两个元素且不会太多，以避免由于低阈值造成的聚类失效问题。单击上方列表中的一个元素，会在下方的文本框中显示所有的文件名列表，用户可以通过文件名列表来依次进行查证，进而确定出最后的组团抄袭情况。

## 5.2 算法性能分析

### 5.2.1 与 BuaaSim 性能对比

数据集选取 BuaaSim 提供的 C 语言评测程序集。该评测程序集内包含了两组 C 语言程序，分别对应两个问题的实现：统计出现次数最多的整数(简称题目 1)和交叉引用生成器(简称题目 2)。题目 1 的程序平均源代码长度 50 行左右，题目逻辑比较简单；题目 2 的程序平均源代码长度 150 行左右，题目难度较大，涉及多个函数调用。每一组内有三个程序，由三个学生独立完成，组内的其它程序都是基于这三个程序进行抄袭。为



了保证样本比较的公平性, 没有告知样本提供者本系统使用的相似性检测算法; 同时, 为了保证样本的典型性, 要求样本提供者在不改变程序最终输出结果的情况下, 尽量采用高级的手段抄袭程序, 而不仅仅是简单的修改注释、更改变量名和重新排版。评测集内两个题目名称分别为 `int` 和 `cross-ref`, 在代码文件名中都有所体现。目录内的原始程序以及对应的抄袭程序如下表所示:

表格 6 相似代码集合抄袭情况

	原始程序	抄袭程序
题目 1	<code>int00</code>	<code>int01</code> , <code>int02</code> , <code>int03</code> , <code>int04</code> , <code>int05</code>
	<code>int10</code>	<code>int11</code> , <code>int12</code> , <code>int13</code> , <code>int14</code> , <code>int15</code> , <code>int16</code>
	<code>int20</code>	<code>int21</code> , <code>int22</code> , <code>int23</code> , <code>int24</code>
题目 2	<code>cross-ref00</code>	<code>cross-ref01</code> , <code>cross-ref02</code> , <code>cross-ref03</code> , <code>cross-ref04</code> , <code>cross-ref05</code>
	<code>cross-ref10</code>	<code>cross-ref11</code> , <code>cross-ref12</code> , <code>cross-ref13</code> , <code>cross-ref14</code> , <code>cross-ref15</code> , <code>cross-ref16</code>
	<code>cross-ref20</code>	<code>cross-ref21</code> , <code>cross-ref22</code> , <code>cross-ref23</code> , <code>cross-ref24</code>

在阈值设定为 0.75 时, 统计出准确率、召回率如下:

表格 7 详细召回情况统计

	相似代码对	正确召回	误召回	未召回
数量	220	214	0	6
比例	100%	97.3%	0%	0.3%

准确率 100%, 召回率 97.3%, F 值 98.6%, 均达到了很高的水准。其中几对未检测出的相似代码的修改多为将一个函数拆为两个函数的情况, 只有汇编代码的相似度较高, 均超过 60%, 但在将阈值设为 60%后会产生大量误召回, 大幅降低准确率, 因此不可再调低阈值。介绍 BuaaSim 系统的论文中并未提到具体的准确率和召回率, 而只是提到了聚类的结果。下表是人工分类、BuaaSim、JPlag 和本系统的聚类结果:

表格 8 人工聚类、BuaaSim、JPlag 和本系统聚类结果对比

题目 1	题目 2



人工聚类	{0, 1, 2, 3, 4, 5}	{0, 1, 2, 3, 4, 5}
	{10, 11, 12, 13, 14, 15, 16}	{10, 11, 12, 13, 14, 15, 16}
	{20, 21, 22, 23, 24}	{20, 21, 22, 23, 24}
BuaaSim	{0, 1, 2, 3, 4, 5}	{0, 1, 2, 3, 4, 5}
	{10, 11, 13, 14, 15, 16}	{10, 11, 13, 14, 15, 16}
	{20, 21, 22, 23, 24}	{20, 21, 22, 23, 24}
JPlag	{0, 2, 3, 4, 5}	{0, 1, 2, 3, 4, 5}
	{10, 11, 13, 16}	{10, 11, 13, 14, 15, 16}
	{20, 22, 23, 24}	{20, 21, 22, 23, 24}
PlagiarismChecker	{0, 1, 2, 3, 4, 5}	{0, 1, 2, 3, 4, 5}
	{10, 11, 12, 13, 14, 15, 16}	{10, 11, 12, 13, 14, 15, 16}
	{20, 21, 22, 23, 24}	{20, 21, 22, 23, 24}

可以看出, PlagiarismChecker 在代码聚类方面有极为出色的表现, 与人工聚类的结果完全相同, 相比 BuaaSim 和 JPlag 系统表现更为优秀。

### 5.2.2 C++/C 跨语言代码相似性检测

本系统相比于 BuaaSim 等其他代码抄袭检测系统, 不仅准确率、召回率和聚类效果出众, 而且能对不同语言间的代码抄袭进行检测。本系统实现了针对由 C 代码改为 C++ 代码、或由 C++ 代码改为 C 代码的代码修改手段的抄袭检测, 以下是对该功能的测试。

测试数据为根据 BuaaSim 中的 C 代码修改出的 C++ 代码, 具体修改情况如下:

表格 9 C++ 相似代码数据的抄袭情况详表

文件名	抄袭源代码	主要修改方法	检测相似代码
add1.cc	cross-ref20.c	1) 插入大段可编译 cpp 代码 2) 修改头文件格式	ref20,ref21,ref22, ref23,ref24
add2.cc	int00.c	1) 添加宏定义	int00,int01,int02,





		2) 使用 namespace 将变量结构复杂化	int04,int05
cross-ref00pp	cross-ref00.c	1)使用 class 实现 2)使用 namespace	ref00,ref01,ref02, ref03,ref04,ref05

测试表明,对于 class 替换 struct 的修改方式,召回性能较好,但对于使用 namespace 进行变量名复杂化的修改方法有一定几率判断错误,总体性能表现依然良好,准确率 100%、召回率 95%,足够为程序设计或数据结构\算法课程作业或 ACM/ICPC 竞赛中进行代码抄袭检测工作。

### 5.3 大规模真实数据测试

使用 2012 年 12 月北航 ACM 校队选拔比赛的数据,使用其中抄袭现象较为明显的 G 题的代码数据进行测试。筛选后共计 42 份返回正确结果的有效代码,预处理消耗时间 35.983 秒,聚类消耗时间 5.832 秒,获得 6 个抄袭组,结果见下表:

表格 10 BUAACPC 2012 G 题相似代码聚类结果

组别	代码列表
1	BCPC12061077.cpp, BCPC12061094.cpp
2	BCPC11061104.cpp, BCPC11061111.c
3	BCPC11211011.cpp,BCPC11211011.cpp, BCPC11211084.cpp, BCPC11211005.cpp
4	BCPC_11211117.cpp, BCPC12061142.c, BCPC11211055.cpp, BCPC11211114.cpp
5	BCPC10091043.cpp, BCPC10091043.cpp, BCPC10091043.cpp, BCPC11061192.cpp
6	BCPC12211009.c, BCPC12211009.c

经过人工验证,这些代码均无一误判,对于 C/C++的跨语言判定也都全部正确。由于实际比赛中抄袭代码的手段较为简单,因此在解决了跨语言抄袭判定的问题之后,可以基本杜绝比赛中的抄袭现象,保证各类 ACM/ICPC 竞赛的公平、公正。



## 5.4 小结

本章对程序代码相似性检测算法、相似代码聚类算法和一个相似代码检测系统 PlagiarismChecker 进行了测试。第一个实验测试了系统的各项基本功能是否运行良好,结果表明提供给用户的所有功能都运行正常无异常,且用户体验良好。第二个和第三个实验都是对算法性能进行测试。第二个实验对算法的性能进行了对比实验,使用的是 BuaaSim 提供的相似代码集,结果表明,本系统不仅时间性能良好,而且在准确率、召回率上都有非常突出的表现,聚类效果非常好,超过了 BuaaSim 和一个常用的相似代码检测工具 JPlag,完全具有实用性。第三个实验对系统的 C/C++跨语言相似代码检测性能进行了测试,使用了基于 BuaaSim 提供的代码集修改出的 3 份 C++代码,在代码集中进行相似检测和聚类测试。测试结果表明,系统对 C++的支持较好,有非常高的准确率和较高的召回率。第四个实验是选用大规模、真实的代码集合对系统进行测试,结果表明对于实际 ACM/ICPC 竞赛中的代码抄袭的检测效果和聚类效果非常好,并且由于 ACM/ICPC 竞赛中多为 C/C++代码,因此基本能够满足 ACM/ICPC 竞赛的代码抄袭检测需求,具有很高的使用价值,并将在北航各类 ACM/ICPC 竞赛中投入使用。

---



## 结论

本文针对现在日趋严重的代码抄袭现象, 基于现有的代码相似性检测模式及相似文本检测算法、相似文本聚类算法等, 提出了同时兼容 C/C++ 的程序代码相似性检测算法和相似代码聚类算法, 并通过软件架构上的设计, 最终实现了一套性能优良的程序代码相似性检测系统, 实现了本课题的研究目标。

论文首先分析了国内外代码相似检测系统的研究现状, 以及当今代码抄袭的严峻形势, 进而提出了本课题的研究目标与内容, 即实现一个可对 C/C++ 代码集进行代码相似性检测、相似代码聚类的工具。而后阐述了当今主流的代码相似检测系统架构、主流的相似代码检测算法和字符串处理算法。并介绍了用于预处理代码、生成可执行二进制文件和反汇编的开源工具 mingw。而后介绍了常用的字符串相似度计算算法 LCS、编辑距离算法和 RKHash 算法三种算法, 并介绍了两种常用的无监督文本聚类算法 KNN 和 K-means 算法。在第三章介绍了本系统的架构和算法设计, 分析了系统的需求, 总结出 10 种常见的代码抄袭手段, 并设计了系统的模块架构和功能模块架构, 以及代码相似度计算算法和代码相似聚类算法, 给出了算法的流程和伪代码。之后详细介绍了系统实现的过程, 对代码优化、词法分析、编译优化反汇编这三块功能的实现进行了介绍, 并阐述了系统的代码结构设计, 共有意后续维护者参考。第四章后半段详细讲述了代码相似度计算算法和相似代码聚类算法的实现, 并给出了核心代码。最后一章记述了对系统基本功能、算法性能的测试结果, 并得出了本系统的效率和准确率、召回率完全具有实用价值的结论。

---



## 致谢

感谢毕业设计的指导教师李舟军教授，让我选择了一个十分喜欢的题目，也十分有价值的题目。也感谢同在李舟军教授下做毕业设计的同学们，和你们每周一次的讨论让我收获很大。

感谢 ACM 队的队友们，在今年要一同参加世界总决赛的情况下我不能尽全力训练，感谢你们的包容和刻苦努力训练。

感谢提供测试数据的董适等同学，为算法测试给予了宝贵的帮助。

还要将感谢献给我的家人，是他们给予的默默支持帮助我顺利的完成了毕业设计的工作。

最后感谢所有曾帮助和支持我的老师、同学和朋友们。

---



## 参考文献

- [1] 熊浩. 代码相似性检测技术的研究与工具实现[D]. 北京; 北京航空航天大学, 2010.
- [2] 张鹏. C 程序相似代码识别方法的研究与实现[D]. 大连; 大连理工大学, 2007.
- [3] 程金宏, 刘东升. 程序代码相似度自动度量技术研究综述[J]. 内蒙古师范大学学报, 2006, 4: 457-461.
- [4] 张文典, 任冬伟. 程序抄袭判定系统[J]. 小型微型计算机系统, 1988, 9(10): 34-39.
- [5] 王继远. 一种用于软件作业评判系统的程序结构分析算法的设计与实现[D]. 北京; 北京邮电大学, 2007.
- [6] 熊浩, 李舟军, 晏海华. 代码相似性检测技术: 研究综述[J]. 计算机科学, 2010, 8: 9-14.
- [7] 赵长海, 晏海华, 金茂忠. 基于编译优化和反汇编的程序相似性检测方法[J]. 北京航空航天大学学报, 2008, 6: 711-715
- [8] Edward L.Jones. Metrics Based Plagiarism Monitoring. The Consortium for Computing in Small Colleges, Vermont, 2001:253-261.
- [9] <http://kerneis.github.com/cil/> [OL]. 2002.
- [10] 刘楠, 刘超, 李虎. 基于 Eclipse 平台的克隆代码检测插件的设计[J]. 计算机科学, 2008, 35(11.专刊)
- [11] 熊浩, 晏海华, 李舟军, 郭涛, 黄永刚. 一种基于 BP 神经网络的代码相似性检测方法[J]. 计算机科学, 2010, 3: 159-164.
- [12] Steve Engels, Vivek Lakshmanan, Michelle Craig. Plagiarism Detection Using Feature-Based Neural Networks[C]. In: Proceedings of ACM SIGCSE, 2007.



- 
- [13] Young-Chul Kim, Yong-Yoon Cho and Jong-Bae Moon. A plagiarism detection system using a syntax-tree [C]. In: Proceedings of International Conference on Computational Intelligence, 2004. 1: 23-26.
- [14] <http://zh.wikipedia.org/wiki/Mingw> [OL]. 2013.
- [15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. 算法导论 [M]. 北京. 机械工业出版社, 2006: 558-559.
-