分类号：TP31　　编号

** 大 学

毕 业 论 文（设 计）

基于抽象语法树的 Online Judge 语义查重系统设计与实现

Design and Implementation of Online Judge Semantic Checking System Based on Abstract Syntax Tree

申请学位：工学学士

院　　系：计算机与***学院

专　　业：计算机科学与技术

姓　　名：张**

学　　号：2013******

指导教师：*********

2017 年 5 月 27 日

****计算机**学院

基于抽象语法树的 Online Judge 语义查重系统设计与实现

姓　名：张*卫

导　师：***

2017 年 5 月 27 日

****计算机****学院

[摘要] 随着计算机技术的不断发展，在线测评系统（Online Judge 简称 OJ）成为教学和竞赛项目的重要辅助平台，其大大提高了教育事业部门的工作和学习效率，成为当今计算机应用大潮下的一个成功典范。

但是随着 OJ 系统的广泛使用，抄袭等现象层出不穷，而各大 OJ 针对此也没有完善的应对机制，大大减弱了竞赛和考试项目的公平性和诚信度；另外，目前的 OJ 系统大多数采用动态评判的方式，因此对于编程思路正确但是实现代码出现错误的代码无法给分，从某种程度上也是降低了 OJ 系统的公平性，错误代码靠人工审批又降低了办公效率，因此此次对当前 OJ 系统开展课题设计。

针对上述两个问题，我将对语义相似度分析和主观题给分方法进行研究，通过 GCC 构建语法树并运用 Zhang-Shasha 算法展开语义分析，最后根据实际测试对 Zhang-Shasha 算法进行优化，并将该语义分析系统部署在 HUSTOJ 上，完成相应的数据分析和实验，寻找解决上述问题的解决方案。

[关键词] 语义相似；GCC 抽象语法树；程序标准化；HUSTOJ；Zhang-Shasha 算法

Abstract: With the continuous development of computer technology, online evaluation system (Online Judge referred to as OJ) has become an important auxiliary platform for teaching and competition projects, which greatly improve the work of the education sector and learning efficiency. Online Judge has become today's successful example of computer applications under the tide.

However, with the extensive use of OJ system, plagiarism and other emerging after another. And for the major OJ there is no perfect response mechanism, which greatly weakening the competition and examination project fairness and integrity. In addition, the current OJ system most use Dynamic evaluation, so the wrong code with correct thinking can't get any points. To some extent, this also reduce the fairness of the OJ system. At the same time, manual checking of error code reduces the efficiency, so this time I will study and design Semantic similarity system based on the current OJ system.

In this paper, to study similarity analysis and the method of subjective subject marking, I will use GCC to build the abstract grammar tree and calculate the similarity by using the Zhang-Shasha algorithm. what's more, I will optimize the Zhang-Shasha algorithm according to the actual test results. Finally, I will deploy the semantic checking system on HUSTOJ and after that analyze the system data to search the way of solving this problem.

Key words： Semantic Similarity; GCC Abstract Syntax Tree; Procedural Standardization; HUSTOJ; Zhang-Shasha Algorithm

目录

1 绪 论

为了更好的对本课题展开研究，增强本次的研究效率，因此需要对当前国内外该项目的研究形式展开分析和判断，了解国内外相似度分析方法以及重要算法的使用频率，从而为本次的具体研究和项目实习拟定计划，确定主要工作，针对以上说明做如下详细调研。

1.1 论文研究背景

在互联网以及人工智能等相关技术迅速发展的时代背景下，计算机技术日渐成熟，并成为办公、科研探索以及教学的主要辅助工具，大大提高了当今日常生活的效率和节奏。多媒体教学作为计算机技术运用的一个重要分支，更是抓住了计算机技术与教学的主潮流，成为各大高校竞相探索和运用的热点。在多媒体教

学改革大潮的众多分支中，由于具备显著提高测评速度和全面性的特性，并能够大大节省人力、减少人工评阅造成的误判，在线考试、在线测评变得炙手可热。

Online Judge（简称 OJ）是在线考试与计算机行业的完美融合，通过在线提交代码，服务器（sever）端对代码进行相应分析，完成结果的反馈。目前国内的众多高校例如浙江大学 Online Judge（ZOJ）、北京大学 Online Judge（POJ）、华中科技大学 Online Judge（HUSTOJ）、杭州电子科技大学 Online Judge（HDUOJ），国外的弗吉尼亚大学 Online Judge（UVA OJ）、Topcoder、BestCoder 等机构和学校的代码在线测评技术已经日趋成熟，在举行重大赛事和教学工作中都产生了巨大的影响力，为 OJ 技术的发展产生巨大的推动和影响作用。

无论多么美好的事情都有它的两面性，同样的 OJ 系统在具备上述巨大优势的同时，其部分情景下测评结果不合理问题的弊端同样不可小觑，这将导致评测结果的偏差，甚至在有些评测中这种偏差已经严重影响了其结果的正确性，其中出现的较为明显的问题有代码抄袭、主观题成绩测评，尤其是错误代码的成绩评测等，除国内外 ACM 等赛事中较为严格的规则测评外，其他赛事和教学成绩测评中，比如计算机等级考试、蓝桥杯赛事以及期末考试中编程设计题，如果单纯通过最后动态执行的结果评定成绩，这很明显不符合人工测评机制，导致了系统最终的错误性评测；而对于比赛中的代码抄袭现象，这严重违反了竞赛中公平、诚信的原则，必须对其进行相应程度的检测和控制，上述这些问题都无疑是比较严重的，也是当前 OJ 系统急需解决的问题。

在经过分析和调研之后发现，产生这些问题的重要原因是目前的 OJ 系统主要是基于动态代码测评方式，也就是通过不同的输入，动态运行目标代码并根据相应输入最终获得相应结果，再将结果与提前设置的正确的结果比对，如果结果相同则给分，否则不给分。很明显，这种方式对于运行时出现错误代码的处理方式是不给分的，但是在这些代码中不乏编程思想正确，但是局部存在语法错误的

代码，在处理此类测评代码时给出的零分处理显然是不准确的，也就造成了与人工判卷思想上的巨大偏差。要想解决此类问题，就需要引入相应的代码静态分析方法，对此类测评进行误差矫正，从而完全切合人工测评思想。

1.2 国内外现状及发展趋势

程序代码的相似度[1]是指通过对两个源程序代码在某种度量方式上的相似性衡量值，也就是把代码的相似度进行的量化，往往是量化为在 0～1 范围内的某个值，越接近 1 其相似度越大，反之亦然。

关于代码相似度的衡量方式，根据其识别的方式和侧重点，可以分为基于结构相似度识别、基于度量值相似度识别、基于抽象模式相似度识别、基于文本相似度识别、基于图的程序相似度识别等识别方法。针对于本文研究重点，下面对基于结构相似度识别、基于文本的相似度识别方式、基于图的程序识别以及基于度量值识别方式相融合的识别方式分析国内外研究现状。

基于文本的相似度识别方式是通过文本的中间方式对目标模式进行识别，具体可以通过 Token 串，并结合词法分析的进行识别，目前比较流行的方式有 Clone Detective[2]、CCFinder[3]、CP-Miner[4]、SIM[5]。SIM 算法是 Dick Grune 教授于 1989 年提出的用于检测自然语言相似度的工具，除此之外，该工具已经广泛用于 C++、Java、Pascal 等多种语言相似度比较，其主要思想是将编程语言变为 Token 串，对变量等进行标准化，并通过最长公共子串算法和 Hash 加速的方式获取其最终的相似度。准确来说其完成的是词法结构上的相似度。

基于结构相似度识别的方式是通过将目标语言结构化为抽象语法树（Abstract Syntax Tree 简称 AST）或者分析树这种中间结构，然后通过对比两个树之间的某些特性量化出其相似度。Yang[6]通过对两个分析树分析来比较两个相同程序不同版本之间的相似度，并通过动态规划（Dynamic Programming）的方式计算子树相似度。Baxter[7]、Raze[8]、李亚军[9]等人也通过对比抽象语法树或者分析树的方式进行结构相似度比较。准确来说该相似度识别方法是语法结构上的相似度识别。

基于图的程序相似度识别是将概念图作为识别的中间结构，其中图的形式目前比较常用的有系统依赖图（System Dependence Graph 简称 SDG[10]）、控制依赖图（Control Dependence Graph 简称 CDG[10]）、程序依赖图（Program Dependence Graph 简称 PDG[11]）。目前 Mishne[12]、Metzger[13]都通过构建图的方式提出一系列的观点和相应研究。基于度量值的识别方式是通过对于程序代码的不同属性进行综合分析，比如程序特殊标识符、函数调用关系等，通过量化公式得出最终的相似度。目前哈工大苏小红所在的团队正在研究通过基于图的程序相似度和基于度量值识别方式相结合的方法[10]，首先将模式程序和目标程序统一用 SDP 方式存储，然后采用结构度量分析的识别方法，提取出候选子结构，最后对候选子结构进行语义级别匹配，最终得到的实验效率和准确率均高于图和度量值的识别方式。

1.3 课题的主要工作及研究意义

根据动态代码测评方式带来的弊端，本课题研究动态代码测评与静态代码测试相融合的方式，使用 SIM 基于文本的代码相似度算法、基于结构的代码识别并通过度量值识别方法对结果进行综合度量，具体步骤如下：

(1)通过 SIM 算法，首先对模式代码和目标代码进行文本相似度分析，获取其文本相似度，作为度量值相似度的分析参数。

(2)由于 GCC 编译器能生成中间的 AST[14]，因此本文通过 GCC 产生文本抽象语法树，然后将文本抽象语法树消除冗余后转为树结构存储的 AST，并进行多个阶段的标准化工作，其中包括选择语句标准化、循环语句标准化、函数调用结构标准化等，最终得到本文分析使用的语法树；

(3)对于标准化后的 AST，通过树编辑距离算法，本文使用 1989 年提出的 Zhang-Shasha[15]算法进行计算，获取模式程序与目标程序 AST 的最小编辑距离，获得结构代码相似度，作为度量值相似度的分析参数。

(4)对于(1)和(3)中得出的相似度参数，通过实验分析，最终获得基于度量值的相似度。

(5)在充分了解 HUSTOJ[16]系统结构基础上，将该系统的实现部署在 HUSTOJ 上，从而最终实现语义查重系统。

本课题的研究能有效解决单纯动态代码测评造成的代码抄袭、主观题成绩测评，尤其是错误代码成绩评测不合理等问题，运用静态代码分析并在苏小红团队的思想基础上设计多种代码相似度衡量方式，从而研究出一种符合人工判卷思想的代码相似度评判系统。

2 相关理论技术简介

由于本课题是一个理论性和实践性都很强的研究课题，在系统研究过程中涉及到编译原理、人工智能、OJ 系统、程序语义学等多方面综合性理论，因此对本文使用的理论和思想做如下介绍。

2.1 GCC 及其工作流程

GCC 编译器（GNU Compiler Collection）是编译器由斯托曼和 Len Tower 编写的支持多平台、多语言的编译器，支持多种操作系统[17]，且是 Linux 系统的预装的编译器，目前扩展支持 C、C++、Fortran、Java 等多种语言，不过本文所提到 GCC 指的是支持 C 语言的 GNU 编译器。其由三个组成部分，分别是前端、后端和机器描述部分，其中前端与编程语言密切相关，后端则与具体程序语言无关，而机器描述部分与具体机器相关，为了能够更好的支持多平台的编译，GCC 编译器增加了 AST 和 RTL（寄存器转移语言）中间表示形式，AST 能够更好的完成语法结构和 RTL 的转化过程，使得整体具有更好的相通性。图 2.1 描述了 GCC 的各部分编译流程。

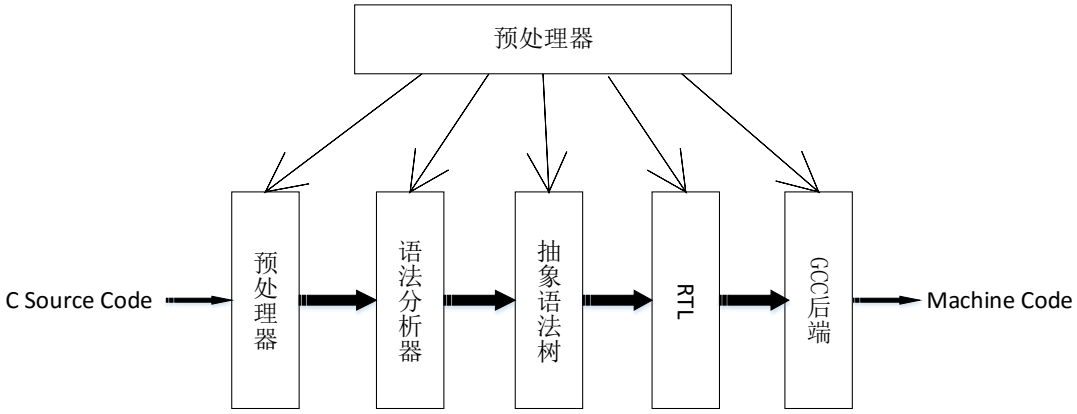

图 2.1　GCC 编译运行流程

2.2 GCC 文本抽象语法树

GCC 生成抽象语法树[18]作为其中间结构，其包含了源程序所有信息，且与源程序的过程一一对应，在 GCC 中通过增加选择编译参数-fdump-translation-unit 即可得到中间的文本抽象语法树，其以.tu 为后缀名。GCC 生成的文本语法树的每个节点都是有节点编号（以@开头）、节点名称和节点属性字段三部分构成，其中属性字段可能包含多个属性节点，结构如图表 2.1。

表 2.1　节点格式

节点编号@	节点名称	属性字段 1	…	属性字段 n

其中文本语法树中的节点类型共有 6 种，如表 2.2 所示。

表 2.2　节点类型

节点名称	节点名称命名特点
标识符结点(Identifier node)	identifier_node 或函数名
类型结点(Types)	后缀:_type
声明结点(Declarations)	后缀：_decl
语句结点(Statements)	statement_list
表达式结点(Expressions)	后缀：_expr
常数结点(constants)	后缀：_cst

在该文本抽象语法树中，每个节点以@开头，具体举例如下。

@3898 var_decl name:@3902 srcp:switch.c:3

其中@3898 为该节点的编号，该编号在 AST 中唯一；var_decl 是节点的名称，其中下划线后面部分是 declaration 的标志，标明该语句是声明节点；冒号后面是该节点的属性字段，属性字段中每个属性由属性名和属性值构成，其中属性值可能是值也可能是子节点编号，子节点编号同样以@开头，在本例中 name 属性名所对应的属性值是子节点编号，即为@3902 节点，而 srcp（source）属性名对应的属性值为 switch.c:3。

2.3 SIM 介绍

SIM[5]是由阿姆斯特丹 Dick Grune 教授于 1989 年研发的一种文本程序相似度检测工具，虽然提出时间较早，但是目前负责 SIM 工具的 Dick Grune 教授仍旧在不断更新该工具。其很好解决了教学中的代码抄袭现象。

在讲述 SIM 核心算法之前，首先对最长公共子序列进行规范化说明。假设原字符串长度为 len，那么从原字符串中删除 0~len 个字符而得到的新串即可称为原串的子序列，而公共子序列就是两个或者多个串公共的子序列，由于其可能存在多个公共子序列,因此第 i 个公共子序列记为$\mathrm{constr}(i)$,其构成的集合记为Ω，易知最长公共子序列即为：

$$\max\{\mathrm{length}(\mathrm{constr}(i))\}$$

其中$\mathrm{length}(\mathrm{constr}(i))$是$\Omega$中的字符串元素的长度。最长公共子序列问题作为最经典的动态规划问题，问题通过推导动态规划转移方程，在不断解决子问题后，最终获得原问题的解。这种字符串最长公共子序列匹配思想主要适用于模糊匹配，例如生物学科中生物学检测、计算 DNA 链之间的关系[19]。

SIM 的核心算法思想就是通过求取模式代码和目标代码最长公共子序列计算两个代码的相似度,与一般最长公共子序列不同点是在获取两个文本字符串的最长公共子序列时，对两个字符串是(match)否(nmatch)匹配分别赋予不同的权值，从而获取最终的匹配权值，也就是最后的相似度参数。设 str1，str2 分别

为两个字符串，其中$\mathrm{Distance}(i,j)$代表字符串$\mathrm{str}_1[1\ldots i]$和$\mathrm{str}_2[1\ldots j]$间的最优匹配权值，则具体动态规划转移方程为如式 2.1。

$$\mathrm{Distance}(i,j)=\max\begin{cases}\mathrm{Distance}(i-1,j-1)+\mathrm{score}(s[i],t[j])\\ \mathrm{Distance}(i-1,j)+g\\ \mathrm{Distance}(i,j-1)+g\\ 0\end{cases}\tag{2.1}$$

其中

$$\mathrm{score}(s[i],t[j])=\begin{cases}m,if(s[i]=t[j])\\ d,otherwise\end{cases}\tag{2.2}$$

2.2 式中的的m为匹配时的权值，d为不匹配时的权值，g为差距分数，其中一般将m设置为正数，而d和g作为不匹配的“扣分”设置为负数。

SIM 算法首先调用词法分析进行扫描，并针对不同的关键字、标点符号、注释标识符进行相应的转换规则，转换为相应的 Token 流，并调用 Linux 下的 flex 词法分析器将 Lex 文件标准化为 C 语言的程序代码。然后将模式代码所对应的 Token 流划分为段，每个段都被看做是模式代码的一个域，不同的域分别与目标代码进行匹配，从而识别出了顺序交换的程序。在识别时，可以对不同的 Token 匹配权值进行相应的修改，从而在匹配时对重点词素提高重视程度，SIM 的具体执行流程如图 2.2 所示。

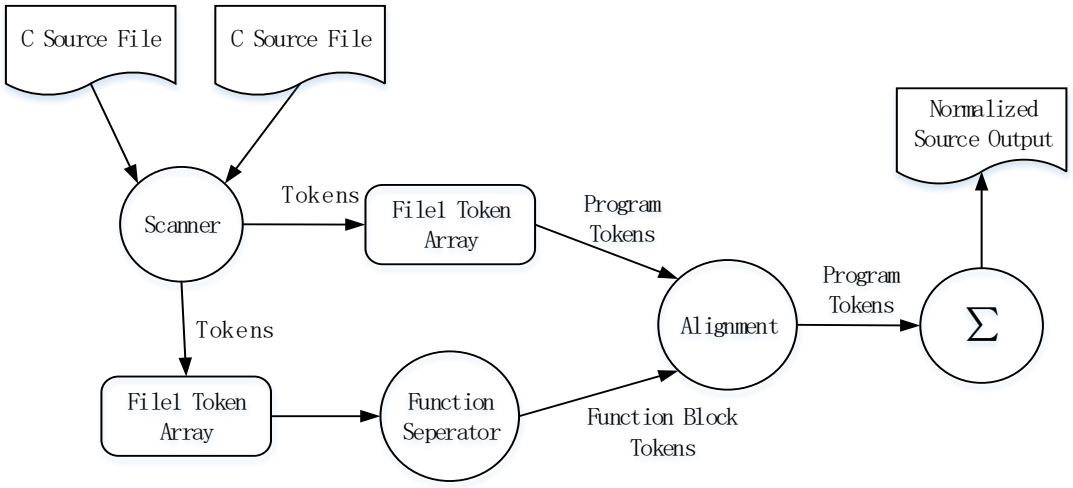

图 2.2　SIM 执行流程

2.4 树结构识别算法

树结构的相似度识别是将源程序转化为树的存储形式，通过比对两棵树的某些特性得出最终的相似度，该种方法通过分析模式代码和目标代码的语法结构特点，将树的特点的异同量化为最终的相似度，是代码相似度比较中一个重要的途径。在树结构识别算法中，基于树的编辑距离算法以其具有高效率、高准确率而得到普遍重视。

2.4.1 树编辑距离算法

所谓的编辑距离（Edit Distance），又称为 Levenshtein 距离[20]，是 1965

年俄罗斯科学家 Vladimir Levenshtein 提出的一个概念。指的是对于两个字符串，从目标串（原串）转变为模式串所需要的最少的编辑操作的次数，其中编辑操作是针对目标串而言的，一共有三种编辑操作，分别是在目标串中删除（Delete_i）一个字符，插入（Insert_i）一个字符，或者是修改（Update_i）一个字符。

树编辑距离[21]最初是由 Tai[22]提出的，其基本思想是通过编辑操作，将目标树T_1（原树）转化为模式树T_2所消耗的最小代价，也就是从T_1映射为T_2（记为$f(T_1 \to T_2)$）所进行编辑操作的最小代价，其基本操作是上述编辑距离的扩展，并应用到了树上，树编辑距离可以表示为 2.3 式

$$\text{Distance}(T_1, T_2) = \min\{f(T_1 \to T_2) | T_1\text{为目标树}, T_2\text{为模式树}\} \quad (2.3)$$

基本操作集为如 2.4 式。

$$\Omega = \{\text{Delete}(j), \text{Insert}(i,j), \text{Update}(i,j) | i \in \text{T}，j\text{为修改的目标节点}\} \quad (2.4)$$

2.4 式$\text{Delete}(i)$、$\text{Insert}(i,j)$、$\text{Update}(i,j)$分别指的是：

(1) $\text{Delete}(i)$：删除操作，i 表示T_1中的节点，将以 i 节点为父亲节点的所有节点作为 i 节点父亲节点的子节点，如图 2.3 所示，其中 i 节点为 B。

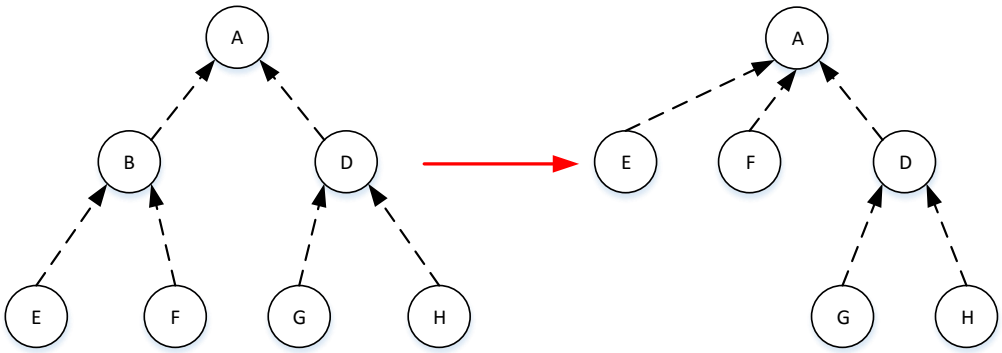

图 2.3　$\text{Delete}(i)$

(2) $\text{Insert}(i,j)$：插入操作，i 表示T中的节点，j 表示新插入节点，将 j 节点插入到 i 节点上，作为 i 节点的孩子节点。如图 2.4 所示，i 节点为 C，j 为 F。

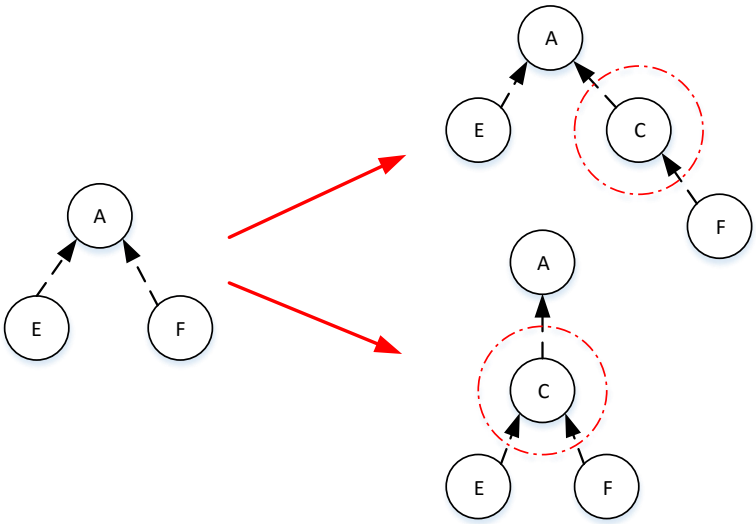

图 2.4　$\text{Insert}(i,j)$

(3) Update(i,j)：修改节点，i 表示T中的节点，j 表示修改的目标节点，将 i 节点信息更新为 j 节点信息。如图 2.5 所示，i 节点为 B，j 节点为 C。

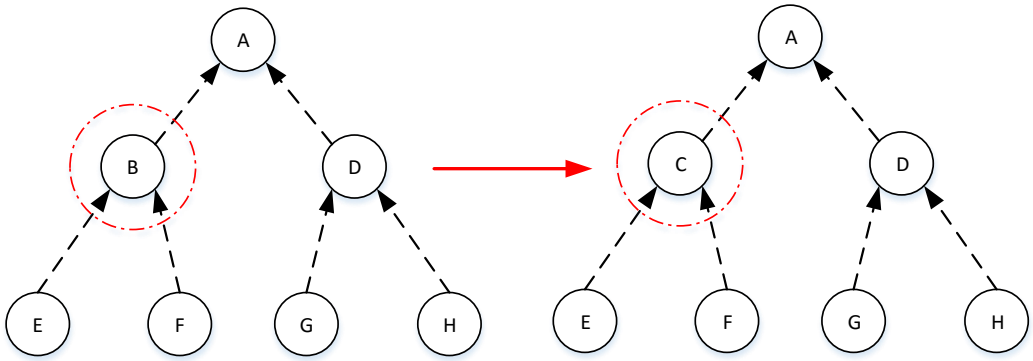

图 2.5　Update(i,j)

目前经典的树编辑距离算法有 Tai 和 Zhang-Shasha，其主要不同是 Tai 树编辑算法是针对无约束的有序树编辑距离算法，而 Zhang-Shasha 算法是有约束的有序树编辑算法，所谓的有无约束指的是树编辑操作是否受到约束，比如 Zhang-Shasha 算法就是有约束的编辑距离算法，其编辑操作都是通过后序遍历后，优先从子节点开始的。如下 2.4.2 节详细介绍 Zhang-Shasha 算法。

2.4.2 Zhang-Shasha 算法

Zhang-Shasha 算法是 Zhang 和 Shasha 于 1989 年提出的树编辑距离算法，在其论文中明确证明了无序树的树编辑距离问题属于 NP 问题，因此，该算法主要用于计算有约束的有序树树编辑距离，因为增加了约束条件，也就是其限制了从子节点优先编辑，因此在很大程度上提高了其计算效率，这也是其与 Tai 算法的显著不同，因此 Zhang-Shasha 算法广泛应用于生物基因匹配和剽窃检测[23]等领域。

由于国内该算法研究较少，且部分论文出现错误理解，通过查阅国外相关文献后，下面对该算法进行详细描述。

设两棵有序树分别为T_1和T_2，其分别有N_1和N_1个节点，将有序树 T 的第 i 个节点标记为$T[i]$。如上 2.3.1 所述，编辑操作产生一个映射，如图 2.6 说明了将T_1变换为T_2的详细过程。

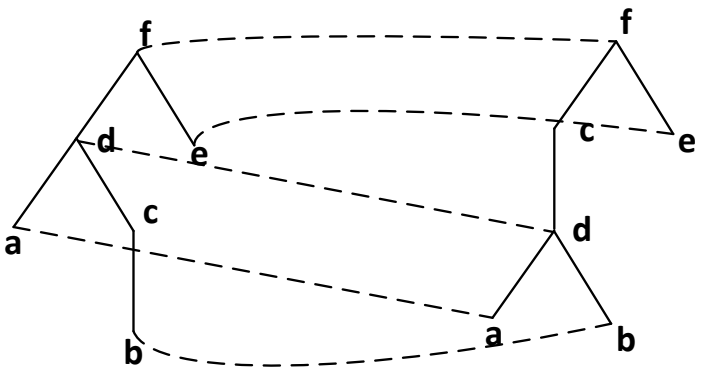

图 2.6　$f(T_1 \rightarrow T_2)$

图 2.6 中虚线连接的节点代表着如果$T_1[i] \neq T_2[j]$，那么$T_1[i]$需要通过$Update(i,j)$操作改为$T_2[j]$，如果$T_1[i] = T_2[j]$，则无需修改，如图中 e 节点；在T_1中没有虚线连接的节点为$Delete(i)$操作，即为删除节点 i，如图 2.6 中T_1中的 c 节点；在T_2中没有虚线连接的节点对应$Insert(i,j)$操作，即在T_1中的i节点上插入T_2中的j节点信息，如图 2.6 中T_1中的 d 插入T_2中的 c 节点中，上述的映射说明了T_1映射为T_2的全过程。

和上 2.3.1 所述稍有不同，$f(T_1 \rightarrow T_2)$表示T_1通过映射关系 f 映射为T_2，这里对映射关系 f 进一步规范化：

f 映射关系是满足如下条件的任意一对(i,j)，其中i为T_1中的节点编号，j为T_2中的节点编号：

(1)$1 \leqslant i \leqslant N_1$，$1 \leqslant j \leqslant N_2$；

(2)对于映射关系里的任意一对映射$< i_1, j_1 >$，$< i_2, j_2 >$满足

① $i_1 = i_2$当且仅当$j_1 = j_2$

② 如果$T_1[i_1]$和$T_1[i_2]$是兄弟关系，$T_1[i_1]$是$T_1[i_2]$的左兄弟当且仅当$T_2[j_1]$ 是$T_2[j_2]$的左兄弟

③ 如果$T_1[i_1]$和$T_1[i_2]$是祖孙关系，$T_1[i_1]$是$T_1[i_2]$的祖先节点当且仅当$T_2[j_1]$是$T_2[j_2]$的祖先节点。

上述的关系定义从上例的图中可以形象的表达为：两条映射的虚线不能相交，也就是限定映射时的条件。

如图 2.6，如果设 I 和 J 分别表示T_1和T_2中没有被虚线连接的集合，也就是说 I 和 J 集合分别表示删除（Delete）和插入（Insert）操作的节点，那么我们可以定义 M 映射对应的编辑距离为：

$$\gamma(M) = \sum_{(i,j) \in M} \gamma(T_1[i] \rightarrow T_2[j]) + \sum_{i \in I} \gamma(T_1[i] \rightarrow \emptyset) + \sum_{j \in J} \gamma(\emptyset \rightarrow T_2[j])$$

除此之外，映射关系 M 可以被分解，设M_1为从T_1到T_2的映射关系，M_2为从T_2到T_3的映射关系，那么定义为 2.5 式。

$$M_1 \circ M_2 = \{(i,j) | \exists k \ (i,k) \in M_1 \ and \ (k,j) \in M_2\} \tag{2.5}$$

那么$M_1 \circ M_2$同样是一个映射关系，且满足 2.6 式。

$$\gamma(M_1 \circ M_2) \leq \gamma(M_1) + \gamma(M_2) \tag{2.6}$$

2.6 式说明了从T_1到T_3的直接映射代价是小于先从T_1到T_2再从T_2到T_3的代价和的，而这个定论在 Zhang-Shasha 的论文里面也给出了严格的证明。

最后定义树编辑距离如 2.7 式。

$$\sigma(T_1, T_2) = \min\{\gamma(M) | M \text{ 是从树} T_1 \text{到} T_2 \text{的映射}\} \tag{2.7}$$

其实上述理论描述是形式化的树编辑距离，详细规范了树编辑距离的定义，下面正式说明 Zhang-Shasha 算法。

首先先对两棵待比对的树进行后序遍历（Post-ordering Traversal），并按照遍历时的先后顺序对树节点编号，编号方式如下图 2.7 所示。

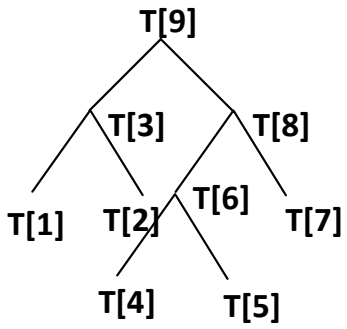

图 2.7　遍历次序

为了算法讲述的简洁性，做如下定义：

(1) l(i)表示以T[i]节点为根节点子树的最左孩子节点，当且仅当T[i]是叶子节点时，其l(i)=i；
(2) p(i)表示T[i]节点的父亲节点，且$\mathrm{p}^0(i)$=i，$\mathrm{p}^1(i)$=p(i)，$\mathrm{p}^2(i)$=p($\mathrm{p}^1(i)$)…；
(3) $\mathrm{anc(i)} = \{\mathrm{p}^k(i) | 0 \le k \le \mathrm{depth}(i)\}$，表示 i 节点的父亲节点集合；
(4) forest(i)表示 T[1..i]组成的树或森林，而 tree(i)表示 T(l(i)..i)组成的树；
(5) size(i)表示 tree(i)子树的节点个数；
(6) $\mathrm{foresdist}(\mathrm{T}_1[i' \dots i], T_2[j' \dots j])$表示$\mathrm{T}_1[i' \dots i]$和$\mathrm{T}_1[i' \dots i]$树距离；
(7) treedist(i,j)表示以 i 和以 j 为根节点的子树距离；

设$i_1 \in \mathrm{anc}(i)$，$j_1 \in \mathrm{anc}(j)$因此两棵树的距离即为 2.8 式。

$$\begin{aligned}&\mathrm{forestdist}(\mathrm{T}_1[\mathrm{l}(i) \dots i], \mathrm{T}_2[\mathrm{l}(j) \dots j]) = \\ &\min\begin{cases}\mathrm{forestdist}(\mathrm{T}_1[\mathrm{l}(i) \dots i-1], \mathrm{T}_2[\mathrm{l}(j) \dots j]) + \gamma(\mathrm{T}_1[i] \to \emptyset); \\ \mathrm{forestdist}(\mathrm{T}_1[\mathrm{l}(i) \dots i], T_2[\mathrm{l}(j) \dots j]) + \gamma(\emptyset \to \mathrm{T}_2[j]); \\ \mathrm{forestdist}(\mathrm{T}_1[\mathrm{l}(i_1) \dots \mathrm{l}(i)-1], \mathrm{T}_2[\mathrm{l}(j_1) \dots \mathrm{l}(j)-1]) \\ \quad +\mathrm{forestdist}(\mathrm{T}_1[\mathrm{l}(i) \dots i-1], \mathrm{T}_2[\mathrm{l}(j) \dots j-1]) \\ \quad\quad +\gamma(\mathrm{T}_1[i] \to \mathrm{T}_2[j]).\end{cases}\end{aligned} \tag{2.8}$$

针对不同情况，该公式可以细化为：

(1) 如果$\mathrm{l}(i) = \mathrm{l}(i_1)$且$\mathrm{l(j)} = \mathrm{l}(j_1)$，那么

$$\begin{aligned}&\mathrm{forestdist}(\mathrm{T}_1[\mathrm{l}(i) \dots i], \mathrm{T}_2[\mathrm{l}(j) \dots j]) \\ &= \min\begin{cases}\mathrm{forestdist}(\mathrm{T}_1[\mathrm{l}(i) \dots i-1], \mathrm{T}_2[\mathrm{l}(j) \dots j]) + \gamma(\mathrm{T}_1[i] \to \emptyset); \\ \mathrm{forestdist}(\mathrm{T}_1[\mathrm{l}(i) \dots i], \mathrm{T}_2[\mathrm{l}(j) \dots j]) + \gamma(\emptyset \to \mathrm{T}_2[j]); \\ \mathrm{forestdist}(\mathrm{T}_1[\mathrm{l}(i) \dots i-1], \mathrm{T}_2[\mathrm{l}(j) \dots j-1]) + \gamma(\mathrm{T}_1[i] \to \mathrm{T}_2[j]);\end{cases}\end{aligned}$$

(2) 如果$\mathrm{l}(i) \ne \mathrm{l}(i_1)$或$\mathrm{l(j)} \ne \mathrm{l}(j_1)$，那么

$$\begin{aligned}&\mathrm{forestdist}(\mathrm{T}_1[\mathrm{l}(i) \dots i], \mathrm{T}_2[\mathrm{l}(j) \dots j]) \\ &= \min\begin{cases}\mathrm{forestdist}(\mathrm{T}_1[\mathrm{l}(i) \dots i-1], \mathrm{T}_2[\mathrm{l}(j) \dots j]) + \gamma(\mathrm{T}_1[i] \to \emptyset); \\ \mathrm{forestdist}(\mathrm{T}_1[\mathrm{l}(i) \dots i], \mathrm{T}_2[\mathrm{l}(j) \dots j]) + \gamma(\emptyset \to \mathrm{T}_2[j]); \\ \mathrm{forestdist}(\mathrm{T}_1[\mathrm{l}(i) \dots i-1], \mathrm{T}_2[\mathrm{l}(j) \dots j-1]) + \mathrm{treedist}(i,j)\end{cases}\end{aligned}$$

至此，Zhang-Shasha 算法的算法思想已经阐述完毕，关于 Zhang-Shasha 算法的伪代码在后续章节 4.3 给出。

2.5 HUSTOJ 简介

HUSTOJ 是华中科技大学教学团队开发设计的开源 Online Judge 系统，虽然其他开源 Online Judge 系统如 POJ、ZOJ 虽然也曾开放源码，但是因为其缺乏相

关的配置文档和开发文档，较难部署，国内外几乎没有能部署成功的，HUSTOJ 因为具备完善的开发文档，且其架构是易于搭建并兼容性较强的 LAMP（Linux + Apache + MySQL + PHP）架构[24]，因此被多次成功移植并投入使用，我校 Online Judge 系统正在使用的就是基于 HUSTOJ 部署的 OJ 系统，本文也借助该 OJ 系统并结合我校 OJ 系统数据完成课题的实验。

HUSTOJ 存在两种运行模式，分别为数据库连接（默认）和 HTTP 方式。数据库模式通过 core 轮询数据库 solution 表,从而完成 web 模块和 Core 模块的交互,而 HTTP 方式则是通过 core 访问 web 端的 admin/problem_judge.php 来实现交互的,本次实验使用的第一种默认的数据库连接方式。

HUSTOJ 主要由 Web 和 Core 两部分构成，其中 Web 主要用于显示修改题目、提交代码，以及提交后对后台 solution 和 source_code 表的修改，以供 Core 判题程序使用；而 Core 由 judged、judge_client、sim 三部分组成，其中 judged 为判题服务进程,这里的_d 就是 daemon 的意思,主要负责轮询数据库中的 solution 表（数据库模式）或者 web 端（HTTP 模式），当出现新的判题任务时产生 judge_client 进程。Judge_client 为实际的判题程序，负责运行环境和数据的准备工作，并检测待测程序的系统调用，对待测程序给出运行结果。而 sim 部分则是可以通过参数选择是否调用的查重模块，其运用的是第三方的应用程序，如 2.3 节所述的文本相似度分析工具。可以在 judge.conf 和 db_info.inc.php 文件中通过修改 OJ_SIM_ENABLE 和 OJ_SIM 参数，将其设置为 1 来实现该模块的调用。

HUSTOJ 的运行模式图如图 2.8 所示

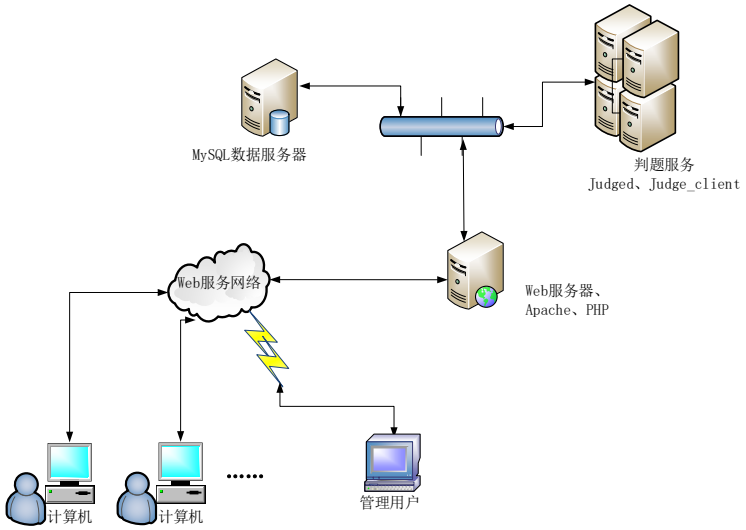

图 2.8　HUSTOJ 运行模式图

3 GCC 文本抽象语法树解析和标准化

本次相似度分析所使用语法树是借助于 GCC 自动生成的，这样一方面提高了系统实现的效率，保证了开发更具应用性，另一方面由于 GCC 完成了部分的优化，因此减轻了语法树标准化的难度。但是 GCC 生成的文本抽象语法树存在信息冗余，需要对其去冗余和重构操作，并在后续的过程中需对抽象语法树进行标准化，从而进一步提高系统的准确性。

3.1 AST 解析

在 2.2 节已经阐述了可以通过 GCC 生成文本抽象语法树，但是要想得到真正的 AST 需要对该.tu 文件进行处理，也就是本节的 AST 解析过程。而对于 AST 解析，哈工大李鑫给出了形式化的定义[25]，通俗来讲就是对文本抽象语法树进行冗余消除变换和重建变换，下面分别阐述。

由于 GCC 生成抽象语法树是为了完成语法结构和 RTL（寄存器转移语言）的转化过程，因此考虑到实现底层优化等功能，增加了大量编译中间节点，虽然具备了一定的完备性和优化性，但是因为大量冗余信息导致该文本不适合直接生成 AST。在后续的实际实验中发现，这些带冗余信息的文本抽象语法树节点是最终 AST 节点数的 1500 倍左右，因此首先需要去除冗余信息。在去除冗余信息后，就可以初步获得中间的抽象语法树了，该语法树通过树节点的形式进行存储，也就是上述中的重建变换。

3.1.1 冗余消除变换

为了保证消除冗余后的节点信息生成的语法树是完整的，将 GCC 生成的文本语法树节点标记为三种状态：

(1) useful_node：已经确定为抽象语法树中的重要节点。
(2) useless_node：已经确定不是抽象语法树中的节点信息。
(3) unknow_node：目前该节点的是否为抽象语法树节点未知。

通过分析 GCC 生成的文本语法树，发现在距离树根节点较近的函数声明（function_decl）节点中，有 srcp 属性字段，其属性值就是编译文件的文件名，因此可以从该节点出发，查找语法树的重要节点。结合上面的节点状态得到如下推论：包含 srcp 属性字段的节点可以确定为 useful_node，而不包含该节点的可以标记为待定节点，鉴于李鑫给出的算法存在错误[25]，现设计算法规则如下：

(1) 顺序遍历节点，如果 i 节点为 useful_node，则其子节点全部标记为 useful_node 节点；
(2) 顺序遍历节点，如果 i 节点为为 useless_node，则将其子节点中的待定节点标记为 useless_node；
(3) 由于上述标记删除了函数调用的信息，因此对于 call_expr 节点标记为 useful_node 节点；

(4) 顺序遍历所有节点后，节点标记为 useful_node 的即为抽象语法树中的节点信息。

算法 3.1：冗余消除变化算法
输入：GCC 编译得到的文本抽象语法树
输出：规范化后用于生成抽象语法树的节点
算法过程：
变量声明：node childnode
Step 1：预处理
Trim&getnodeline 对输入的文本抽象语法树标准化为每行一个节点，且去除每行多余的空格
Step 2：直接节点标记
foreach 字符行 i
if 属性节点含有 srcp 字段 then
if srcp 字段值 = 编译文件名 then
标记该节点状态为 useful_node
else
标记该节点状态为 useless_node
else 标记该节点状态为 unknown_node
Step 3：循环标记所有节点
foreach 字符行 i
记录当前节点为 node
if 节点状态不为 unknown_node
获取子节点编号 childnode
While 当前节点还存在未遍历子节点 do
Begin
if node 节点标记是 useful_node then
标记 childnode 为 useful_node
if node 节点标记是 useless_node&&childnode 是 unknown_node then
标记 childnode 为 useless_node
childnode=getNextNodeNum
End
Step 4:函数调用节点标记
for 字符行
if 节点名为 call_expr then
将该节点标记为 useful_node

算法 3.1 中，对李鑫算法主要改进是对于标记为 useful_node 节点的所有子节点都标记为 useful_node，而不是只修改子节点标记是 unknown_node 的节点，因为这样会丢失掉一些常量、变量类型等节点信息。

3.1.2 重构变换

在筛选出标记为 useful_node 节点后，就得到了构建 AST 的节点，但是因为筛选出的节点是不连续的，因此不便于以后的节点识别和存储，因此在重构变换

前先对节点编号进行哈希变换。哈希之后，在构建树的时候，针对被筛选出来的节点，由于子节点还存在相对于构建 AST 没有用的节点属性节点，因此在重构的同时，再完成该属性字段的冗余内容的消除工作，这些冗余字段包括简单类型的 type 字段、节点中的 srcp、algn 字段、子节点中已经被消除的子节点、标识符节点中的 lngt 子节点.

下面定义基于 AST 的数据存储结构如下：

```
struct Child ///定义子节点存储结构
{
    string attribute;          ///属性节点名
    int childnodenum;          ///属性节点号
};
struct Node{
    vector<Child>Childlist;    ///子节点列表
    vector<string>attribute;   ///节点属性列表
    int index;                 ///用于后续 Zhang-Shasha 算法编号标记
    int leftmost;              ///用于后续 Zhang-Shasha 算法记录 l(i)
    string nodename;           ///节点名称
};
vector nodelist[nodenum];
```

上面每个节点中都有两个动态数组，其分别记录子节点信息和本节点属性列表，通过动态数组的方式减少了用指针造成程序易出错、可读性差的问题。这里增加属性列表(attribute)，用来存储当前节点属性，使其与子节点列表区分开，便于后续的遍历等其他操作。另外由于后面需要用 Zhang-Shasha 算法对 AST 进行分析，因此节点信息里面的 index 和 leftmost 变量在该部分不会使用，其使用将在后续章节给出说明。

算法 3.2：节点哈希算法

输入：初步规范化文本节点

输出：哈希并消除冗余字段并用树结构存储的 AST

算法流程：

变量声明：hash[节点个数]、cnt=0 计数器、nodelist[筛选出的节点个数]

```
Step 1：//get hash index
  foreach 文本节点 i
    if 节点标记是 useful_mark then
      hash[i]=cnt++;
Step 2：
  cnt=0; ///计数器清零
  foreach 文本节点 i
     if 节点标记是 useful_mark then
     Begin
     设置 nodelist[cnt].nodename
```

```
while 没遍历完所有子节点
  Begin
    判断是否上述中的冗余字段
    if 该子节点不是冗余字段 then
     Begin
       该子节点编号初始化为 childnode
       if 该属性字段有子节点编号 then
            childnode = hash[childnode];
            nodelist[cnt].edgelist.push_back( childnode );
       else
            nodelist[cnt].attribute.push_back(该属性信息 );
     End
  End
 cnt++;
End
```

3.2 AST 标准化

获得树结构的 AST 后，因为 GCC 生成的最初的语法树实际是图结构，比如变量节点、函数调用节点中，其子节点中都包含类型这个子节点，如果类型相同，就都会指向相应的类型节点，这样就也就变成了图，即使通过上述的操作仍旧没有解决这个问题，因此该 AST 其实只能算是伪 AST，需做基本标准化。除此之外，函数中由于存在多种选择语句，例如 if 语句、if/else 语句、switch 语句，虽然语法不同，产生的语法树不同，但是其语义是相同的，因此需要做标准化。

3.2.1 基本标准化

首先针对变量等情况，完成初步标准化，也就是去除常量、变量等引用造成的环，其具体算法思想是：首先找出 function_decl 节点，因为出现在该节点之前的节点都是常量声明或者类型定义，对于 function_decl 后的节点 node，如果包含 function_decl 前的节点 childnode，那么直接将 childnode 的属性值复制到 node 属性节点下，在对图顺序遍历完成后，其实就已经消除了 funcition_decl 前的节点了；而对于出现在 function_decl 之后的节点中，其 node 节点可能包含的 childnode 满足编号 node<childnode，也就是变量的多次引用等情况，由于这种情况下子节点 childnode 删除对本文中 AST 分析没有影响，为了让本文研究更具扩展性，不做删除，而是将其变为属性节点，这样也不会影响抽象语法树的分析，增强了可扩展性。

算法 3.3：基本标准化
输入：树结构存储的 AST
输出：变量、常量标准化的 AST
算法流程：

此处省略一个算法伪代码 ^.^ ……YY 一下吧！欢迎与我交流!!!

通过该标准化后，就去除了伪 AST 中因为变量、常量等造成的环，但是在函

数调用和数组指针引用时，还存在环的情况，关于对该情况下的环，将在 3.3.3 节予以消除。

3.2.2 选择语句标准化

由于 C 语言中存在多种选择语句，因为其语法结构不同，因此造成其生成的抽象语法树也有很大的差异，因此需要对选择语句进行标准化，其中选择结构的标准化又分为 if/else/if、if 和 if/else、冒号表达式、switch 语句其互相嵌套形式。

在分析 GCC 生成的 AST 以及查阅相关文献后发现，GCC 的 AST 中实际的选择语句只有 cond_expr 和 switch 两种形式，而含 if 的选择表达式中生成的 cond_expr 根据有无 else 分为两种形式，其不同之处在于无 else 分支的语句生成的 cond_expr 只有两个子节点，否则为三个子节点。其中 cond_expr 节点结构如图 3.1 所示，switch 语句节点结构如图 3.2 所示，针对以上分析，现将选择语句统一标准化为如图 3.3 的形式。

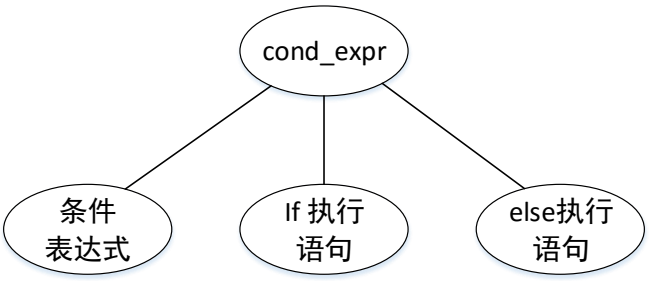

图 3.1　cond_expr 节点结构

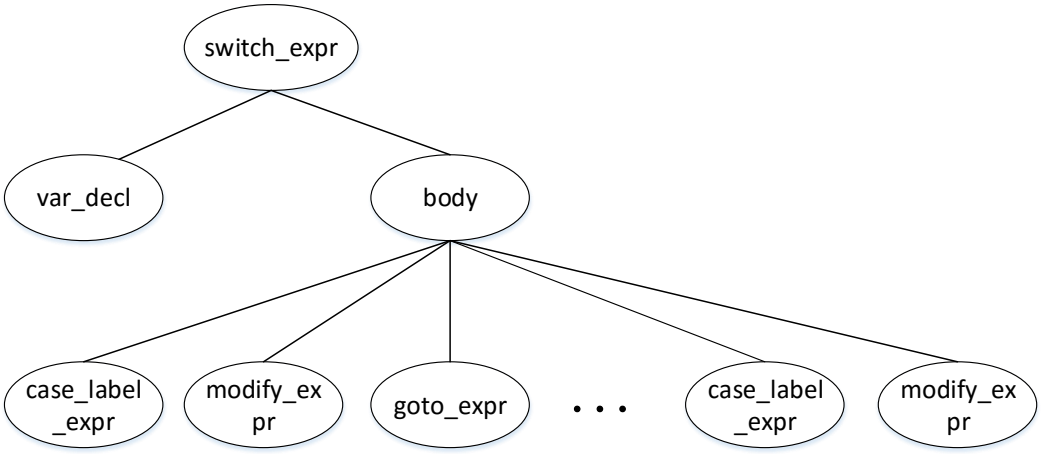

图 3.2　switch 语句节点结构

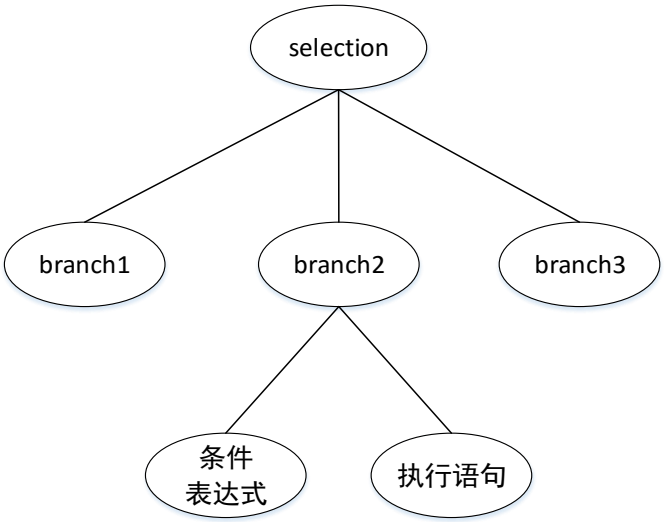

图 3.3　选择语句标准化结构

对于 cond_expr 节点，需要添加在 AST 上添加 selection，并且如果有 else 语句，还需要添加 branch 节点，且最后的 else 语句默认没有执行条件，其他情况下，直接将 cond_expr 节点修改为 branch 节点，第一和第二个节点作为条件表达式和执行语句即可，如果 cond_expr 第三个节点仍然是 cond_expr，则说明是嵌套的 else/if，删除之，并将其作为最新的 branch 节点；对于 switch 语句相对复杂，需要对每个 case 新建 branch 节点，并将 switch_expr 下的判断变量与每个 case 下的值连接成等号判断表达式（eq_expr）作为条件表达式。另外因为 switch 中有 break 和 goto 语句，也就是对应的树上的 goto_expr，标准化时按照执行语义将该 case 下的语句与 goto_expr 之前的所有语句整合为 statement_list 作为当前 branch 的执行语句，至此选择语句规格化完毕。

针对上述算法思想描述，现详细描述算法思想如算法 3.4 所示。

算法 3.4：选择语句标准化
输入：完成基本标准化的 AST
输出：选择结构标准化后的 AST
算法流程：

此处省略一个算法伪代码　^.^　…… YY 一下吧！欢迎与我交流!!!

3.2.3 循环语句标准化

C 语言中的循环语句分为 while、for、do_while 三种语法结构，但是在 gcc 生成语法树时，生成的 while 和 for 是一样的，而 do_while 因为其执行语句一定会执行一次，因为相较于其他两种循环语句少了一个 goto 节点，因为其语法语义相似，且对本次分析结果影响较小，故在本文标准化中不做处理。

但是在循环语句中，因为 GCC 为了语法树的规范化，执行语句 statement_list 下的子节点都为表达式，也就是以_expr 结尾的节点，label_decl 作为 label_expr 节点的子节点，当出现 goto 时，会构成环，因此在这里需要处理 label_expr 节点，将其子节点删除，如此操作不会改变原来语

义，反而便于 AST 语法树处理。因为思路简单，这里不在详写算法伪代码。

3.2.4 函数调用及其他标准化

函数调用、数组变量中，因为多次调用或者数组变量声明，导致出现环，由于该次研究主要是语句块的语义匹配，针对函数调用出现的环，做如下处理：树中第一次出现该函数调用（call_expr）则将其作为该节点的子树进行分析，之后的其它调用只分析到 call_expr 节点，对调用的函数不再二次分析，在尽量保持语义的基础上避免了环状结构对生成 AST 的影响。

在分析数组变量等其他出现环的情况下，统一运用处理函数调用的思想予以处理，为了保证研究的可持续性，实际操作是将不再二次分析的节点修改为属性节点，从而最终消除了伪语法树中的所有环状结构，并生成最终标准化后的 AST。具体算法描述如算法 3.5 所示。

算法 3.5：函数调用及其他标准化

输入：选择、循环语句标准化后的伪 AST

输出：完成函数调用及其他标准化后的 AST

算法流程：

变量声明：

此处省略一个算法伪代码　^.^　…… YY 一下吧！欢迎与我交流！！！

至此，完成了伪 AST 的标准化，并最终去环，从而得到了用于分析编辑距离的 AST，后续章节将对该 AST 进行语义相似度对比。

3.3 复杂度分析

上面的标准化需要对节点进行遍历，在每一次标准化中，设节点个数为 m，节点的子节点个数为n_i，所以其实际时间复杂度为$\sum_{i\le m} n_i$,所以其时间复杂度是 O(m*m)，因此设常数 k，则总的时间复杂度 k*O(m*m)。

对于算法中的空间开辟，因为文本语法树的节点信息是从文本中加载的，且每次加载 1 个节点，空间复杂度可以看做是常数级的；前期在转化为 AST 节点时做了 hash 处理，因为哈希函数采用了直接 hash 的方式，设文本 AST 节点个数为 M，则空间复杂度 O(M)；设 hash 之后的节点个数为 N（N<M），由于节点中有子节点列表和属性列表，且子节点个数为常数级别的，则节点空间复杂度为 O(k*N),k 为常数。综上，其空间复杂度为 O(N)。

4 基于 AST 的相似度设计

在获得标准化的 AST 后，需要解决的的核心问题就是如何通过两棵 AST 对其进行相似度分析，本次采用的是通过最小编辑距离的方式，相对于其他相似度分析算法，其具有相对较高的时空复杂度和准确性，因此最终选取的是最小编辑距离算法中比较实用的 Zhang-Shasha 算法，并通过优化完成相速度分析过程。

4.1 引言

对于获取的 AST，如何展开分析，也就转化为了对两棵树结构相似度的分析，常见的方法有树核（kernel）算法和树编辑距离算法，由于 kernel 方法相对复杂且时空开销比较大，本文选择树编辑距离算法，而相较于 Tai 方法，Zhang-Shasha 在算法效率和空间开销上都有明显的提高，因此做为本文 AST 相似度分析的核心算法。在 2.4.2 节已经详细描述了 Zhang-Shasha 算法的思想，本节再结合 AST 相似度分析做进一步优化。

4.2 Zhang-Shasha 算法

算法首先需对子状态进行初始化，也就是一些子操作状态，比如从当前子树删除为空子树，或者从空子树插入节点，然后用上面介绍根据 $l(i_1)$ 和 $l(i)$ 以及 (j_1) 和 $l(j)$ 相等与否的关系分别用转移方程式，在解决子问题的基础上，最终完成树距离的计算，这里用伪代码更详细地给出 Zhang-Shasha 算法实现的算法思路，如下算法 4.1。

算法 4.1：Zhang-Shasha 算法

输入：两棵标准化后的 AST T_1 和 T_2

输出：T_1 与 T_2 的树编辑距离

算法流程：

声明：function treedist(i,j)

此处省略一个算法伪代码 ^.^ …… YY 一下吧！欢迎与我交流！！！

可参考 http://blog.csdn.net/u014665013/article/details/77587248

4.3 基于 AST 的 Zhang-Shasha 算法优化

研究目前 Zhang-Shasha 算法应用后发现，其多数是直接用于求树距离，因此其删除、添加、修改操作多被设置为 1，少数对删除、添加、修改赋予不同权值，但是在本文 AST 相似度分析中，由于程序代码多为相对简单的语法，因此依然将三种操作赋值为 1，但是根据语法树特点和人工判题的思维特点，程序的差异主要集中在重要的关键点上，比如循环、选择语句等，因此针对此特点，本文细化 Zhang-Shasha 算法的权值，使其更符合本文的应用规则。

设 map<node,cost>代表对节点名为 node 的节点进行三种操作的权值，

Default 表示对节点进行上面操作的默认权值，T_1、T_2分别代表待对比的两棵树，delcost[i]表示从T_1树中删除 i 节点的权值，inscost[j]表示从T_2树上插入 j 节点的权值，relcost[i,j]代表将T_1中 i 节点修改为T_2中 j 节点的权值，且易知由于插入和删除节点都设置为默认值 1，因此对T_1树插入和对T_2树删除在权值上是等价的，这里权值设置规则如下：

上面已经说得本意很清楚了，这里就不共享了 ^*^ ……欢迎与我交流!!!

针对上述描述，具体增加初始化函数，并对 treedist 做相应修改，如算法 4.2 所示。

算法 4.2：改进的 Zhang-Shasha 算法

输入：两棵标准化后的 AST T_1和T_2

输出：T_1与T_2的带权树距离

算法流程：

函数声明：Main()、initial()、function treedist(i,j)；

此处省略一个算法伪代码 ^.^ …… YY 一下吧！欢迎与我交流!!!

4.4 算法复杂度分析

由于算法只计算以keyroot数组中为根节点的子树，而根据算法性质和 Zhang-Shasha 的论证[15]，得到：

$$|\text{keyroot}(T)| \leqslant |\text{leaft}(T)|$$

其中 leaft（T）为树的子节点集合（本节中的表示方法与 2.3.2 一致，可参考）

下面计算每个节点的计算次数，对于每个节点其计算次数为记为：

$$\text{coldepth}(i) = |\text{anc}(i) \cap \text{keyroot}(T)|$$

而为了衡量算法最坏复杂度，需要获得每个节点计算次数的最大值，记为

$$\text{coldepth}(T) = \max|\text{anc}(i) \cap \text{keyroot}(T)| = |\text{depth}(T) \cap \text{keyroot}(T)|$$

从而得到：

$$\text{coldepth(T)} = \min\{\,|\text{depth}(T)|\,, |\text{leaft}(T)|\,\}$$

由于每个树只计算以keyroot中元素i为根节点的子树，则计算时计算次数为子树的节点个数，即$\text{size}(i)$，而所有子树的节点和即：

$$\sum_{i \in keyroot} \text{size}(i) = \sum_{i=1}^{i=N_1} \text{coldepth}(i) < \sum_{i=1}^{i=N_1} \text{coldepth}(T) = \text{N}_1 * \text{coldepth}(T)$$

所以算法的时间复杂度即为：

$$\text{O}(\,(\text{N1} * \text{coldepth}(T_1)) * (\text{N2} * \text{coldepth}(T_2))\,)$$

从而得其时间复杂度：

$$\text{O}((\text{N1} * \min\{|\text{depth}(T_1)|, |\text{leaft}(T_1)|\}) * (\text{N1} * \min\{|\text{depth}(T_2)|, |\text{leaft}(T_2)|\}))$$

根据算法的流程易知，其复杂度为辅助数组的空间大小，即为$\text{O}(\text{N}_1 * \text{N}_2)$而改进后的时空复杂度均为常数级，故其整体复杂度不变。

5 代码相似度在 HUSTOJ 上的部署

为了更好的对设计的语义查重系统进行分析和应用，因此将该系统部署在 HUSTOJ 上，因此在本章详细描述系统的部署，最终融合文本相似度、结构相似度的度量值分析将在下一章进行描述。

本文的部署根据 HUSTOJ 运行结果分为两种情况，分别针对运行结果不正确（Wrong Answer）和运行正确（Accept），具体部署思想将在 5.1、5.2 小节分别给出。而本次的侧重点是错误代码的测评，因此在后端设置考试模式，以减少资源的消耗，具体也将在 5.3 节给出。

5.1 Wrong Answer 代码查重系统部署

此处省略一个具体实现　^.^　…… YY 一下吧欢迎与我交流!!!

5.2 Accept 代码查重系统部署

此处省略一个具体实现　^.^　…… YY 一下吧，欢迎与我交流!!!

为了能够及时在前端显示主观题测评结果，对 web 测评结果显示界面也做了相应修改，从而能够显示错误时产生的最终评分，其配置文件，也就是 state.txt, 和原系统配置文件路径相同，如此设计使系统更加完整规范，更便于前端和 core 模块中的 judge_client 逻辑代码的处理。

6 度量值相似度分析与实验结果

对于运行结果 Wrong answer 和 Accept 的代码，在进行文本和结构化性相似度分析以后，需要对重点参数和权值进行设置。为了更好的适应语义分析的需求，在部署 HUSTOJ 之前，需要对系统结果进行预估和分析，从而更加准确的确定其参数。

6.1 C 语言代码提取和文本语法树生成

本文使用的分析代码为本学校 OJ 系统的实际 Accept 代码数据，因此具备相对较高的参考性，对于最终度量值相似度分析的参数设置有较高的实际意义。但是由于之前 sorce_code 中存储的代码虽然有.c(c 代码)和.cc（c++代码）的区别，但是其实际是在.cc 的文件中有大量 C 语言执行代码，且出现了空文件，为了数据集的完备性，对程序语言判断后，提取 C 语言代码。

对于获取的代码数据，由于其导出时按照实际文件存储结构，因此需要对问价递归搜索判断，其判断逻辑如下：

(1) 对于以.c 结尾的文件，如果非空则直接作为后期使用数据集；

(2) 对于以.cc 结尾的文件，如果非空，且其文件中不存在"iostream"、"bits/stdc++.h"、"cstdio"、"using namespace std"等 c++的头文件信息，就作为 C 语言分析数据集。

具体实现从略，继续 YY…欢迎与我交流。。。

6.2 度量值参数设置

如前文描述，由于单纯文本相似度存的弊端，而单纯结构分析会忽略一些比如变量、表达式等细节，结合之前的度量值分析，本文也采用这种方式。

本文度量的综合属性主要是文本相似度结果 textSim、结构相似度结果 treeSim、程序的规模 scale、函数的个数 function_num。

此处省略一个具体参数和实现方式 ^.^ …… YY 一下吧！ 欢迎与我交流。。。

6.3 测评

这里是对本次进行的数据分析，有部分误差，需要进一步处理… 欢迎与我交流。。。

对于最终的数据分析，存在一定误差，分析过后发现其误差来源主要：

(1) 本校 OJ 后台测试数据不足，动态测评时获取的 pass_rate 误差较大；

(2) 后台 AC 可对比的代码不足，在出现新算法时，缺少可参照的“相似代码”。

因此其对应的解决方案是增加动态测试数据和样例程序的数量，但是在实际中增加多样化的样例数据也存在难度，因此比较好的解决方案是，在考试过

后，对所有错误代码重新评判，因为考试过后 AC 的样例库比较完整，也就在很大程度上减小了误差。

结 束 语

在经过毕业设计前期学习，到白热化系统设计、实现和实验数据分析、参数确定，再到最后毕业设计论文撰写，期间经历各种技术和理论上的瓶颈，甚至有一度近乎停滞的阶段，但是这个历程中无论是对 Linux 编程、HUSTOJ 系统的方案理解，还是对编译原理底层设计、编辑算法理论的认识，都有了更加深刻的理解，使我的能力进一步提高。

首先是对编译原理，更准确的是编译期间对代码的编译和优化过程，虽然之前也有写过手工构造编译器，但是因为后来功能的增加，导致最终构造的编译器十分臃肿，数据结构设计不合理，之前也从来没有过代码优化的概念，在接触 GCC 后，发现其生成的 AST 结构相当匀称，相较于手工构造的编译器 GCC 实现功能也是特别完善的。在此期间还了解到代码相似度的多种计算方式，尤其是对结构化的相似度分析，认识到相应算法，比如树编辑距离中的 kernel 算法、Tai 算法、Zhang-Shasha 算法等，接触了之前很少涉猎的多个应用型算法。

其次，由于后期设计到数据分析，因此多个程序调用了 Linux 系统方法，因此对 Linux 下编程也有了实践性的认识。除此之外，本文借助于 HUSTOJ，因此对其整体的判题机制、判题流程、整体系统构建也有了一定程度的认识。

在设计过后，发现了一些需要后期完善的功能和设计，整理如下：

(1)虽然基于结构化的相似度分析能在但一定程度上添加语义理解，但是因为其数据结构的限制，对一些细节性的处理能力不足，比如表达式中变量和常量语义相同，但是其产生的语法树是有很大差异的，因此要得到功能相对完善的语义相似度分析，还是需要用图的相似度分析，通过图的存储方式可以让这些问题得到进一步解决，目前比较好的结构和算法是通过系统依赖图来实现的，而哈工大的研究团队在该方面有较多研究和相应成果。

(2)在标准化时，由于该系统的目标是相对简单的语义理解处理，因此在选择语句时，switch 语句统一将内部的 break、goto 语句标准化为 break，因此相应的语义理解有偏差，另外针对表达式等一些变量细节等也做了简单处理，语法树叶子节点知识表达式语句，需要后期的进一步处理和改善。

(3)由于本次主要针对 C 语言，因此可以在此基础上扩展 C++的语义分析功能。

(4)为了让该系统更加适应人工阅卷，需要在该基础上通过更多的实验和实际试题进行测试，目前主要针对 Wrong Answer 代码，在以后的使用中可以针对其他错误格式代码加以修改，从而更加符合判题流程。

因为前期对本系统研究较少，网上可查资料也比较少，本文设计也仅仅是抛砖引玉，在整理相应资料后提出一种相对较新的计算思想，我也将在之后公开源码，希望能对后来研究者一些借鉴和帮助。

致 谢

大学时光如白驹悄然流逝，虽然还有很多事情没来得及做，但是大学带给我的成长和历练却是永恒的，给自己留下了宝贵的财富。在这段不长不短的时光里，烟台大学为我营造了很好的成长、学习环境，学院老师们也给我的成长提供了难以言表的帮助，在这里感谢学校、学院老师带给我的成长，尤其是 ACM 实验室，这个温馨的环境让我学到了很多知识，也让我孵化出了很多想法和见解。

尤其感谢***教授给予我的辛勤指导！密切关注我们组毕业设计的进度，尤其是我的毕业设计完成情况，不知多少次***老师在百忙中专程来实验室给我梳理思路、设计方案，在生活上也给了我很多关心，交流想法，碰撞思想的火花。

尤其感谢实验室***老师和***老师带给我的帮助，封老师不但是实验室的精神支柱，更是我心灵导师；周老师的话也是句句肺腑，给我的学业和生活提出意见，给予帮助；还有实验室的小伙伴们，谢谢你们，实验室能遇到你们，就够了！

感谢身边的学长和朋友们对我的学习上的帮助，让我的毕业设计得以顺利完成。

衷心谢谢你们！

参考文献

[1]. 程金宏. 程序代码相似度度量研究[D]. 内蒙古师范大学,2007.
[2]. H.Yoshiki,K.Toshihiro,K.Shinji and I.Katsuro. Method and Implementation for Investigating Code Clones in a Software System. Information and Software Technology, 2007
[3]. S. Livieri, Y. Higo, M. Matsushita and K. Inoue. Very-large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder. IEEE Proceedings of the 29th International Conference on Software Engineering, 2007:106~115
[4]. Z. Li, S. Lu, S. Myagmar, Y. Zhou. CP-Miner: a Tool for Finding Copy-Paste and Related Bugs in Operating System Code. Proc of Operating System Design and Implementation, 2004:289~302
[5]. David Gitchell and Nicholas Tran. Sim: A Utility For Detecting Similarity in Computer Programs[A]. In Proceedings of the 30th SIGCSE Technical Symposium, March 1999
[6]. W.Yang. Identifying Syntactic Differences between Two Programs. Software: Practice and Experience. 1991
[7]. I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna and L. Bier. Clone Detection Using Abstract Syntax Trees. International Conference on Software Maintenance, 1998:368~377
[8]. A. Raza, G. Vogel and E. Plodereder. A Tool Suite for Program Analysis and Reverse Engineering. Proceedings of the 11th Ada-Europe International Conference on Reliable Software Technologies, Porto, Portugal, 2006:71~82
[9]. 李亚军,徐宝文,周晓宇. 基于 AST 的克隆序列与克隆类识别. 东南大学学报. 2008,38(2):228~232
[10]. 王甜甜. 结构语义相似的程序识别方法研究[D]. 哈尔滨工业大学,2009.
[11]. J. Krinke. Identifying Similar Code with Program Dependence Graphs. Proceedings Eighth Working Conference on Reverse Engineering, 2001
[12]. G. Mishne. Source Code Retrieval using Conceptual Graphs. Master of Logic Thesis. Institute for Logic, Language and Computation (ILLC) University of Amsterdam. 2003.
[13]. R. Metzger, Z. Wen. Automatic Algorithm Recognition: A New Approach to Program Optimization. MIT Press. 2000
[14]. 李鑫,王甜甜,苏小红,马培军. 消除 GCC 抽象语法树文本中冗余信息的算法研究[J]. 计算机科学,2008,(10):170-172.
[15]. K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. SIAM J. Computing, 18(6):1245{1262, Dec. 1989.
[16]. HUSTOJ 官方网站：http://acm.hust.edu.cn/
[17]. 彭冲. GCC 编译器[J]. Internet:共创软件, 2002(9):88-90.
[18]. 王相懂,张毅坤. 基于 GCC 的抽象语法树对 C++源程序结构的分析[J]. 计算机工程与应用,2006,(23):97-99+105.
[19]. 李琼. 生物识别模糊性与生物密钥算法研究[D]. 哈尔滨工业大学, 2005.
[20]. 周汉平. Levenshtein 距离在编程题自动评阅中的应用研究[J]. 计算机应用与软件,2011,(05):209-212.
[21]. Reis, D. Castro, et al. "Automatic web news extraction using tree edit distance." Proceedings of the 13th international conference on World Wide Web. ACM, 2004.
[22]. 韦龙宝. Tai 树编辑距离算法的存储优化与树的纵向归并算法[D].中国工程物理研究院,2015.
[23]. 邓爱萍,徐国梁,肖奔. 程序源代码剽窃检测串匹配算法的研究[J]. 计算机工程与科学,2008,(03):62-64+68.
[24]. 余立强. LAMP 架构搭建与网站运行实例[J]. 网络与信息,2011,(08):50-52.
[25]. 李鑫. GCC 抽象语法树的解析及控制依赖子图的建立方法研究[D]. 哈尔滨工业大学,

2008.