文章编号：2095-2163(2021)06-0063-08　　中图分类号：TP391.41　　文献标志码：A

二次文本差异化的失配问题及其改进算法

公　鑫，杨春花

（齐鲁工业大学（山东省科学院）计算机科学与技术学院，济南 250300）

摘　要： 当前人工理解代码变更主要在文本差异化分析（textual code differencing）工具提供的 Hunk 集上进行，而有些工具还对每个 Hunk 进行二次差异化分析，并将结果展现在并排（side-by-side）的视图中，从而方便用户查看 Hunk 内部的变更。然而现有的二次差异化分析工具得到的结果普遍存在语句失配问题，即 Hunk 内部删除行与添加行之间的不恰当匹配以及配对的语句内部 Token 的拆分问题，影响理解事实的变更。本文首先对该问题的分布进行了调查，证实了其在二次差异化分析中的普遍性；其次，对该失配问题产生的原因进行了分析，并提出了一种对其改进的算法。该算法基于轻语法分析，将 Hunk 中的文本行映射为语句行，并将语句行识别为基于单词（Token）的序列；对所有的删除语句和添加语句进行相似性匹配，对于匹配的语句应用最长公共子序列算法获取其内部的差异 Token。算法当前采用 Java 语言实现，并在一个自制的 eclipse 展示插件中对其结果予以展示。通过在 5 个开源项目上的实验，证明了该算法可以有效克服 Hunk 内语句行之间的失配以及 Token 的拆分问题。

关键词： 文本差异化分析；二次差异化；失配；Hunk；Token

The mismatch problem in the textual re-differencing and its improved algorithm

GONG Xin，YANG Chunhua

（School of Computer Science and Technology，Qilu University of Technology（Shandong Academy of Sciences），Jinan 250300，China）

【Abstract】 Currently，manual code changes are mainly carried out on the Hunk set provided by the text code differentiation tool，and some tools also conduct secondary differentiation analysis for each Hunk，and display the results in a side-by-side view，so that users can view the changes in the Hunk. However，the results obtained by the existing secondary differentiation analysis tools generally have the problem of statement mismatch，that is，the improper matching between deleted and added lines in Hunk，and the splitting of Token within paired statements，which affect the understanding of the fact. Firstly，the distribution of the problem is investigated，which proves its universality in the second differentiation analysis. Then the reason of the mismatch problem is analyzed and an improved algorithm is proposed. Based on light syntax analysis，the algorithm maps text lines in Hunk to statement lines，and identifies sentence lines as Token based sequences. Then，similarity matching is performed for all deleted and added statements. For matched statements，the longest common subsequence algorithm is used to obtain the internal difference Token. At present，the algorithm is implemented in Java language，and the results are displayed in a self-made eclipse display plug-in. Experiments on five open source projects show that the algorithm can effectively overcome the mismatch between sentence lines in Hunk and the splitting of Token.

【Key words】 text differentiation analysis；secondary differentiation；mismatch；Hunk；Token

0　引　言

现代大型软件的开发一般基于版本管理系统由多人协作完成，开发者每天将变更的代码提交到软件仓库中，而阅读和理解代码变更则成为代码评审以及日常开发和维护工作的基础[1-2]。

当前对代码变更的审阅一般在文本差异化分析（textual code differencing）工具提供的 Hunk 集上进行。例如，图 1 是典型的文本差异化分析工具GNU-Diff 返回的一个 Hunk 示例。一个 Hunk 由一段连续的被删除（-）和插入（+）的行以及前后 1~3 个上下文行构成。一个 Hunk 集构成了源代码修改前后的代码变更，变更审阅者只需要查看这些 Hunk，就可以知道代码哪里发生了什么改变，不需要人工对比 2 个版本的源代码，提高了代码变更的理解效率。

对于复杂的变更，以行为单位的 Hunk 结果不能展现 Hunk 内部的具体变化。为此，许多工具如 Kdiff3 和 Beyond Compare 4、GitHub Diff 等对 Hunk

基金项目：山东省自然科学基金（ZR2020MF031）。

作者简介：公　鑫（1995-），男，硕士研究生，主要研究方向：代码变更分析；杨春花（1974-），女，博士，教授，主要研究方向：代码变更分析、软件演化、软件开发。

收稿日期：2021-03-11

内部的删除行和添加行之间进行二次差异化分析,从而获得细化的变更结果。图 2 是 Beyond Compare 4 对图 1 的 Hunk 进行二次差异化分析的结果,该结果展示在并排(side-by-side)窗口中,其中发生变化的行和单词(Token)进行了加亮显示。后文均默认新版本为右侧文本,老版本为左侧文本。Hunk 内的二次差异化分析方便用户快速理解 Hunk 内行的具体变化,但是当前的二次差异化分析工具得出的结果存在失配现象,主要体现为行之间的失配和一个完整 Token 的拆分现象。根据图 2 的展示,老版本(左)的 132~134 行被分析为与新版本(右)中第 161 ~ 163 行匹配对应;而新版本(右)中"Constraints"等单词(Token)的部分字符被分析为差异文本。前者是一种行失配,因为事实上老版本(左)的 132~134 行应该与新版本(右)中第 161~162 行相似性匹配,这种匹配体现了对该变量声明语句的编程风格的变化。

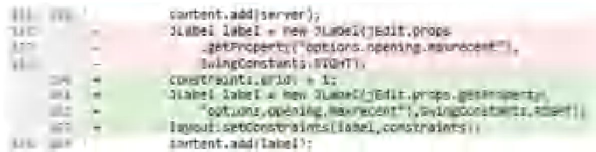

图 1 Hunk 例子

Fig. 1 Hunk example

图 2 Beyond Compare 4 分析结果

Fig. 2 Beyond Compare 4 analysis results

在前期研究工作中,作者发现这种失配现象在许多工具中都或多或少的存在,由于这种失配现象造成分析的结果不能反映事实上的代码变更,从而会误导代码的理解。为了克服该问题,本文在对该失配现象进行调查和分析的基础上,提出了一个对其进行克服的算法,并将该算法进行了实现和验证。

1 国内外研究现状

代码差异化分析是理解代码变更的基础,通过比较源代码修改前后 2 个版本,获得变化了的代码。代码差异化分析主要分为 2 类:文本差异化和树差异化。

文本差异化分析算法将源代码看成一个字符串,因此代码修改前后 2 个版本之间的比较就变成了 2 个字符串间的比较,一般基于最长公共子序列算法实现[3-4]。最经典的文本差异化算法是 GNU-Diff,目前各种版本管理系统如 Git、SVN 和客户端管理工具 source tree、以及 eclipse 等 IDE 的 External Compare1 插件中集成的 diff 工具或窗口基本都基于该算法。其它典型的文本差异化分析方法包括 L-Diff、LH Diff 、JSync 等,这些方法返回基于文本行的变更,以 Hunk 为单位返回。正如 Kdiff3、Beyonce Compared 4 等工具提供对 Hunk 内的二次差异化分析,获得 Hunk 内部的差异文本。文本差异化分析算法可以分析各种类型的源代码文件,而且效率高,因此在实践中被广泛采用[5-6]。

树差异化分析算法,将 2 个版本的源代码分别进行语法分析形成抽象语法树(AST),通过对比 2 棵 AST 之间的不同,输出变动的结点[7-8]。最著名的算法是 Change Distiller,还存在算法 GumTree、CLDIFF、MTDiff、Diff/TS 和 IJM 支持对移动代码的差异检测,而 CSLICER 和 RTED 不支持对移动代码差异检测,这些树差异算法可以输出语法意义上的代码变更集,但是由于相比文本差异化算法来讲效率低,而且一般针对特定的开发语言,因此在实践中没有被广泛采用。

为此,一些工具对 Hunk 内部进行二次差异化分析,将 Hunk 内部删除行和添加行之间进行相似性比较,获得细化的差异化结果。例如,针对图 1 的 Hunk,Beyonce Compared 4 对其进行二次分析,返回如图 2 的差异结果。但是这些二次差异化分析工具返回的结果存在许多不合理的现象,主要表现为行失配和单词(Token)失配,前者指语法上相似的语句行之间的不匹配,后者由于单词没有作为基本单位进行差异化分析造成的一个或多个单词的内部子串作为差异化结果输出-即 Token 被拆分的现象,如图 2 所示。

这两种失配现象在大多数对 Hunk 进行二次差异化分析的工具中普遍存在,给变更理解者造成不好的体验,同时容易误导给出错误的变更结果。

因此,为了克服此现象,本文提出基于 Token(单词)的 Hunk 内二次差异化分析算法,以 Token 为基本单位,进行语句行间的相似性匹配以及相似行内的 Token 差异分析。该算法已实现,并在 5 个开源项目进行了验证。

2 二次差异化分析中的失配问题

2.1 失配问题的占比

为了查清失配问题在二次差异化分析中的分布,从前期工作中一直采用的数据集中随机选取了

部分提交版本，分别利用典型的二次差异化分析工具 Kdiff3、Beyonce Compared 4 对其分析，人工对其结果中的失配情况进行分析。

数据集是由 5 个开源项目组成，分别从各个项目的有效文件中各随机抽取 100 条，共 500 条数据，每条数据为一文件的某 2 个版本。这些数据分别利用工具 Kdiff3、Beyonce Compared 4 分析后，人工判断其差异结果，具体分析结果如图 3 所示。在 Kdiff3 和 Beyonce Compared 4 工具对该数据集分析结果中，均有 50% 以上的结果存在错误，其中由于编程风格等原因造成的语句失配与由于 Token 部分差异而造成的 Token 失配所占比重尤为突出，可见这些问题在现有工具中是普遍存在的，所以本文的研究是有意义的，同时该数据集用来验证本文算法及工具是有效的。

（a）Kdiff3 分析结果　　（b）Beyond Compare 4 分析结果

图 3　Kdiff3、Beyond Compare 4 差异分析情况

Fig. 3　Kdiff3、Beyond Compare 4 difference analysis

2.2　问题分析

图 2 所示是对图 1 所示的 Hunk 应用 Beyond Compare 4 得到的差异结果。从该结果可以看出，老版本（左）的第 132～134 行被分析为与新版本（右）中第 161～163 行对应，新版本 160 行分析为添加行，黑色的字符为差异字符。从该结果容易看出，BeyondCompare4 是将每一个删除行和添加行分别视为一个字符串，然后应用最长公共子序列算法分别得到的差异的字符，没有体现每个删除行和添加行之间的匹配关系，也没有以单词（Token）为单位，造成了一个 Token 的内部子串被部分差异化的现象。新版本的第 163 行中的最后一个单词 constraints，被解释为其中的“c”和“rai”被增加。如果一个 Hunk 中存在大量的修改行，且修改无规律的话，这种分析结果将会干扰变更理解，增加理解者的负担。

图 4 是对图 1 所示 Hunk 应用 KDiff3 工具得到的分析结果，可以看出，依据 KDiff3 的结果，新版本（右）中的第 160 行、161 行、和 163 行被分析为添加行，老版本（左）中的 132 和 133 行被分析为删除行，老版本的第 134 行被分析为修改成新版本的第 162 行（即老版本的第 134 行和新版本的第 162 行二者相似性匹配）。然而，从理解者的角度，容易看出，老版本的第 132～134 行属于一条赋值语句，其实际被修改成了新版本中第 161～162 行所属的赋值语句，这种修改实际是编码风格上的改变（即仅仅涉及到回撤、Tab 等分隔符的改变）。因此，KDiff3 的这个结果一定程度上给理解者造成误导。

图 4　Kdiff3 分析结果

Fig. 4　Kdiff3 analysis results

分析图 4 中 KDiff3 的结果，不难看出，其首先在删除的文本行和添加的文本行之间进行了相似性匹配，然后在匹配的文本行内部又进行了一次最长公共子序列算法，得到差异的子串。但是编码风格的修改，可能会使一条语句分散到连续的多个文本行中，因此这种基于文本行的行与行之间的匹配，不能保证得到的是一条完整语句之间的匹配。因此，要克服 KDiff3 的这种局限性，需要首先识别一条完整语句对应的文本行，然后再进行语句间的匹配。

通过以上分析，可以看出造成行失配和 Token 失配现象是由于在二次差异化分析时没有实现语法意义上行与行之间的匹配，以及匹配的语句内部的差异以字符为单位而不是 Token 为单位进行的。针对这两点，提出基于 Token 的 Hunk 内二次差异化分析算法。

3　基于轻语法分析的 Hunk 内二次差异化分析算法

基于对二次差异化分析方法产生原因的分析，可以发现失配问题的主要原因是在 Hunk 的删除行和添加行之间未进行语法意义上行之间的相似性匹配。因此，本文提出基于轻语法分析的 Hunk 内二次差异化分析算法，之所以称为轻语法分析是指针对流行编程语言如 Java、C 等的语句构成形式，将 Hunk 中的文本行映射为语句行，并从每个语句中抽取其单词（Token）构成一个 Token 序列，然后对语句所对应的 Token 序列之间应用相似性匹配和最长公共子序列算法，从而实现二次差异化分析。

3.1　算法架构

算法的总体架构如图 5 所示。算法的输入是一个 Hunk，该 Hunk 是通过 GNU-Diff 算法产生 Hunk 集中的一个元素，由删除行、添加行和上下文行构成，输出二次差异化分析后的 Hunk，包括删除的语

句、添加的语句、以及更新语句内部的差异单词(即添加的 Token 和删除的 Token)。

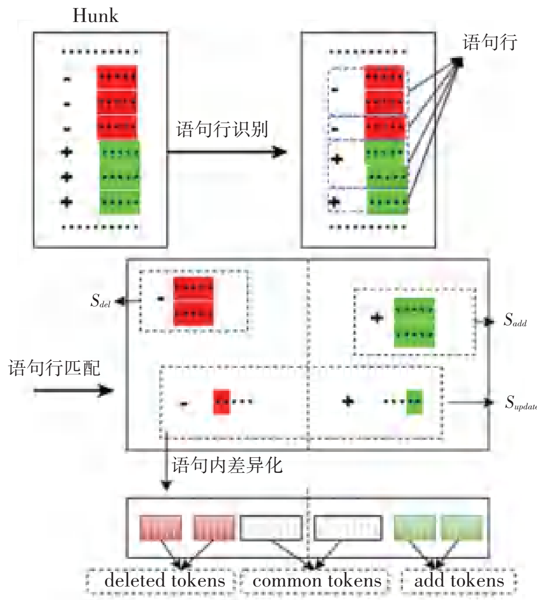

图 5 算法框架图

Fig. 5 Algorithm framework

算法主要由语句识别、语句相似性匹配和语句内部差异化 3 个步骤构成。

首先,对每个删除的行和添加的行分别进行语句识别。该步骤将根据源文件所用编程语言的特点,将跨行的语句进行合并处理,必要的话需要添加相应的上下文,该步骤结束将得到一个删除语句序列和一个添加语句序列。

其次,对所有的删除语句和添加语句进行相似性匹配,得到最相似的语句对。其中,匹配的语句识别为语句更新(update);删除语句中未匹配的语句识别为语句删除(del);添加语句中未匹配的语句识别为语句添加(add)。

最后,对每个语句更新,分别产生老版本语句和新版本语句的 Token 序列,对这两个 Token 序列进行差异化分析,得到差异的 Token。

3.2 算法

算法 HunkDiff 实现了提出的算法,其描述如算法 1 所示。

算法输入一个 Hunk h,由 3 部分构成:删除行集合 L_-、添加行集合 L_+ 和上下文行集合 $L_{context}$。其中,Hunk h 是 GNUDiff 算法分析得到 Hunk 集的一个元素。

一般地,一个差异化分析算法的输出是一个编辑操作(edit operations)集合,常见的编辑操作分为添加 ADD,删除 DEL,和更新 update。因此,本算法的输出是差异化分析后的 Hunk h',其是一个四元组:(S_{DEL}, S_{ADD}, S_{update}, $Delta$),其中 S_{DEL} 为删除语句操作的集合;S_{ADD} 为添加语句操作的集合;S_{update} 为更新语句操作的集合。

对于每一个语句更新操作,设为 $< s_-, s_+ >$,其中 s_- 是老版本的语句,s_+ 是新版本的语句,则该算法会分析此语句内部的 *Token* 差异,将识别的结果存到 *Delta* 中。具体地说,*Delta* 中的每个元素是一个三元组:($< s_-, s_+ >$, T_{DEL}, T_{ADD}),其中 $< s_-, s_+ >$ 是一个语句更新操作;T_{DEL} 是删除的 *Token* 集合;T_{ADD} 是新增的 *Token* 集合;剩余的 *Token* 则是未修改的。

该算法的步骤解释如下:

算法首先通过辅助函数 identifySentences 分别对删除行集合 L_- 和添加行集合 L_+ 进行语句识别,根据源文件编程语言的特点,得到语句集合 S_- 和 S_+;

其次,对集合 S_- 和 S_+ 中语句逐一进行相似性匹配,得到集合 S_{DEL}、S_{ADD} 和 S_{update}。相似性匹配算法 *matching* 在算法 2 中描述;

最后,对于集合 S_{update} 中的每个语句更新操作 $< s_-, s_+ >$,执行语句内的差异化分析,对应于算法的第 7 ~ 15 行。通过辅助函数 TokenSplit 对 s_- 和 s_+ 中的 *Token* 进行识别,得到各自的 *Token* 序列 l_- 和 l_+;对序列 l_- 和 l_+ 执行最长公共子序列算法,得到一个最长公共子串的下标序列 $index_-$ 和 $index_+$,其中 $index_-$ 代表公共子串在老版本中的下标,$index_+$ 代表公共子串在新版本中的下标,具体定义如下:

$index_- = (i_1, i_2 \ldots i_x), for\ o \in (1, \ldots x) \wedge i_o \in (1, \ldots m) \wedge m = len(l_-)$,

$index_+ = (i_1, i_2 \ldots i_y), for\ p \in (1, \ldots y) \wedge i_p \in (1, \ldots n) \wedge n = len(l_+)$.

从 l_- 中抽取不在 $index_-$ 中的下标序列,得到老版本中的 *Token* 差异集合 T_{DEL},该功能用辅助函数 $genDiffTokens(l_-, index_-)$ 来实现。同理,依据 $index_+$,得到新版本中的 *Token* 差异集合 T_{ADD}。因 genDiffTokens 函数实现简单,在此省略其描述。

算法 1 HunkDiff

输入 一个 Hunk $h = (L_-, L_+, L_{context})$,其中 L_-、L_+ 和 $L_{context}$ 分别为删除、添加的文本行和上下文行。

输出 二次差异化后的 Hunk $h' = (S_{DEL}, S_{ADD}, S_{update}, Delta)$,其中 S_{DEL} 为删除的语句集合;S_{ADD} 为添加的语句集合;S_{update} 为更新的语句集合;*Delta* 包

含每条语句更新内部的差异 *Token*。

1　$S_- \leftarrow identifySentences(L_-)$; // 删除语句行识别

2　$S_+ \leftarrow identifySentences(L_+)$;// 添加语句行识别

3　$(S_{DEL}, S_{ADD}, S_{update}) \leftarrow matching(S_-, S_+)$, where S_{DEL} Í S_-, S_{ADD} Í S_+,

$S_{update} = \{ <s_-, s_+> | s_- \in S_- - S_{DEL}, s_+ \in S_+ - S_{ADD} \}$;

//语句匹配

4　$Delta \leftarrow$ Æ

5　for each $<s_-, s_+> \in S_{update}$ do

6　$l_- \leftarrow TokenSplit(s_-)$;

7　$l_+ \leftarrow TokenSplit(s_+)$;

8　$(index_-, index_+) \leftarrow LongestCommonSubs-equence(l_-, l_+)$;

9　$T_{DEL} \leftarrow genDiffTokens(l_-, index_-)$; // 语句内差异化

10　$T_{ADD} \leftarrow genDiffTokens(l_+, index_-)$; // 语句内差异化

11　$Delta \leftarrow Delta \cup \{(<s_-, s_+>, T_{DEL}, T_{ADD})\}$;

12　*end for*

13　$h' = (S_{DEL}, S_{ADD}, S_{update}, Delta)$ // 返回结果

若 Hunk 中删除语句行为 m，添加语句行数为 n，且存在最大相似语句对 $\max(m,n)$；每个相似语句对必存在有限序列的字符串，即相似语句对的差异化分析为常数级；所以算法 1 的时间复杂度为 $O(m*n*\max(m,n))$。

3.3　相似性匹配 matching 算法的描述

算法 matching 实现了语句行之间的相似性匹配，其描述如算法 2 所示。

该算法输入语句行集合 S_- 和 S_+，输出语句删除操作集合 S_{DEL}、语句增加操作集合 S_{ADD} 和语句更新操作集合 S_{update}。算法设置辅助变量：相似对序列 *Simlist*，匹配标志数组 $Matched_-$ 和 $Matched_+$。

算法首先将 S_- 中的每条语句与 S_+ 中的每条语句逐一进行相似性比较，将对应的语句及其相似值存入序列 *Simlist*。相似性比较在 2 条语句所对应的 *Token* 串之间进行，相似性计算采用 *Jaccard* 系数计算[9]。已知 *Token* 串 $l_- = (t_1, t_2, \dots, t_m)$ 和 $l_+ = (t_1, t_2, \dots, t_n)$，*Jaccard* 值计算如式(1)：

$$jaccard(l_-, l_+) = \frac{l_- \cap l_+}{l_- \cup l_+}. \quad (1)$$

其次，对序列 *Simlist* 中的语句对依据其相似值进行递减排序。对排序后的序列 *Simlist*，逐个元素取出，进行配对。设当前元素为 (s_-, s_+, sim)，如果这 s_- 和 s_+ 都没有和任何语句配对（即标志数组 $Matched_-$ 和 $Matched_+$ 中该语句的对应标志为 *false*），则 (s_-, s_+) 视为相似语句，加入集合 S_{update}，同时将这 2 条语句的配对标志设为 *true*。通过这个过程，可以将最相似的语句识别。

最后，语句集合 S_- 和 S_+ 中剩余的语句分别加到 S_{DEL} 和 S_{ADD} 中返回。

算法 2　*matching*

输入　一个删除语句行集合 S_- 和一个添加语句行集合 S_+。

输出　语句删除操作集合 S_{DEL}，语句增加操作集合 S_{ADD}，S_{update} 为语句更新操作集合。

1　*Simlist* ← Æ; $Matched_-[\,] \leftarrow$ false; $Matched_+[\,] \leftarrow$false; //辅助变量初始化

2　for each $s_- \in S_-$ do

3　　$l_- \leftarrow TokenSplit(s_-)$;

4　　for each $s_+ \in S_+$ do

5　　　$l_+ \leftarrow TokenSplit(s_+)$;

6　　　$sim \leftarrow Jaccard(l_-, l_+)$;

7　　　$Simlist \leftarrow Simlist \cup \{(s_-, s_+, sim)\}$;

8　　end for

9　end for

10　sort (*Simlist*) // 按相似度递减排序

11　$S_{update} \leftarrow$ Æ; $Temp_- \leftarrow$ Æ; $Temp_+ \leftarrow$ Æ;

12　for each $(s_-, s_+, sim) \in$ Simlist do

13　　if (! Matched [s_-.index] && ! Matched [s_+.index]) then

14　　　$S_{update} \leftarrow S_{update} \cup \{(s_-, s_+)\}$;

15　　　*Matched* [s_-.index] ← true;

Matched [s_+.*index*] ← true;

16　　　$Temp_- \leftarrow Temp_- \cup \{s_-\}$; $Temp_+ \leftarrow Temp_+ \cup \{s_+\}$;

17　　end if

18　end for

19　$S_{DEL} \leftarrow S_- - Temp_-$; $S_{ADD} \leftarrow S_+ - Temp_+$;

3.4　示例

流行编程语言 Java、C 的语句，绝大多数是以“;”、“{”或“}”为语句结束标记，辅助函数 *identifySentences* 以此为一条语句的结束标记，对于

不存在此标记的一行或多行语句，将借助上下文中的标记符。以图 1 所示 Hunk 为例，辅助函数 *identifySentences* 在输入该 Hunk 后，首先根据 java 语言的语句特点，将以“;”、“{”或“}”结束标记符结尾的语句行划分记录为一条语句，使得由于个人编程风格等原因而跨行的语句，在差异分析时能够完整的参与比较，可能存在无结束标记符的语句行，为此将借助上下文的语句行。得到的删除集合 S_- 和添加集合 S_+，分别记录一条语句｛$<132, 134>$｝和 3 条语句｛$<160, 160>$, $<161, 162>$, $<163, 163>$｝。

根据集合 S_- 和 S_+ 进行语句的相似性匹配，得到相似对序列 $Simlist = \{<132, 134>, <161, 162>, 1\}$，由于序列 *Simlist* 内只包含了一组相似度为 1 的元素，所以该 *Hunk* 不存在语句更新操作和删除操作，仅存在语句增加操作集合 $S_{ADD} = \{<160, 160>, <163, 163>\}$。

如果存在集合 S_{update}，会将 S_{update} 中元素的语句 s_- 和 s_+ 进行差异化分析，得到的基于 Token 的最长公共子序列，进而得到 T_{DEL} 和 T_{ADD}，即得到 *Delta*，具体结果如图 6 所示。

4 算法实现与验证

4.1 算法实现

该算法采用 Java 语言实现，已知源文件的 2 个版本，首先通过 GNU-Diff 得到 Hunk 集，然后对每个 Hunk 应用提出的算法，获得 Hunk 的二次差异化分析结果。同时，设计和实现了一个 Diff 工具 HunkDiff，依托 Eclipse 平台，借助于 Eclipse 插件 External Diff 对项目提交数据进行获取后，将提交中的源代码文件的修改采用当前的算法实现抽取结果，基于可视化框架 JavaFX 设计了 Diff 展示窗口对结果进行展示。

该工具对图 1 的 Hunk 应用提出的算法后得到的结果如图 6 所示。可以看出，图 6 的结果克服了行失配和 Token 失配，符合变更的真实意图。相比 Beyonce Compared 4 和 Kdiff 3 的分析结果，HunkDiff 工具通过忽略回撤符进行语句间的差异分析，避免了这种由于编写风格而产生的差异，减少了对不必要代码行的分析；同时基于 Token 的语句内差异分析，使得差异分析结果更加准确，提高了代码差异分析准确率。

此外，本工具为了能够更加简单、明了的展示代码差异，还对结果的展示方面做了进一步的处理，采用了 All/Diff 模式，如图 7 所示，该模式可选择是全文整体展示差异的模式，还是省略相同部分仅展示差异的模式。这使得差异分析结果更加准确、清晰的展示出来，极大的提高了开发者与代码评审者阅读和理解代码的能力。

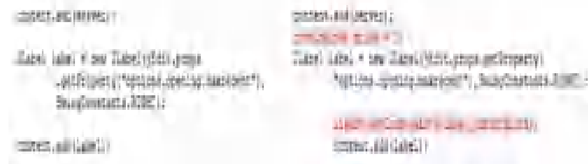

图 6　HunkDiff 分析结果

Fig. 6　HunkDiff analysis results

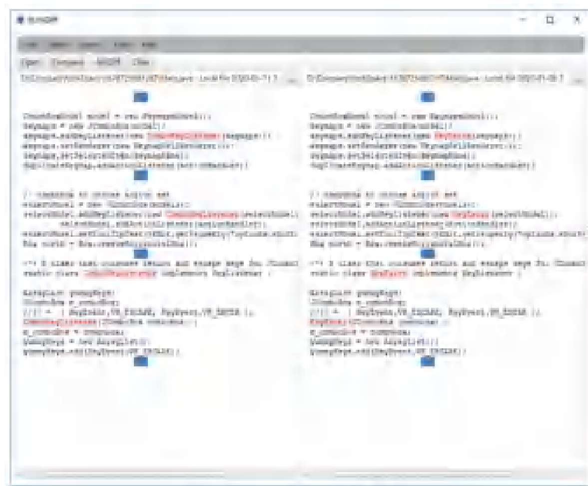

图 7　All/Diff 模式

Fig. 7　All/Diff mode

4.2 算法验证

本文随机抽取了 5 个开源项目中的部分提交记录作为数据源，将提出的算法与 Kdiff3、Beyonce Compared 4 的结果进行了验证和分析。

4.2.1 数据来源

表 1 展示了 5 个开源项目的概况。该数据集抽取自 5 个开源项目的某个时间段的提交，采用 MiniGit 工具构建，一直用于本课题组的差异化分析方法的研究中[10]。其中 eclipseJDTCore 开源项目，这是一个针对 Java 的集成开发环境。Maven 开源项目，这是一个通过信息描述来管理项目的构建、报告和文档的一个开源的管理工具。jEdit 开源项目，这是一个跨平台的文本编辑器。google-guice 是一个轻量级的依赖注入容器。folly 是一个 c++组件库，包含在 Facebook 上广泛使用的各种核心库组件。本文算法主要是针对 java 和 c++代码文件变更的分析，其中代码文件数量约为实际文件数的 60%，同时去掉一些只是更改注释代码文件，有效文件约为

实际文件的 50%，有效提交次数也大大降低，由于一个代码文件可能提交了多次，所以本文中一个实验数据为一文件的某次提交记录。

表 1　项目概况

Tab. 1　Project overview

项名称	提取时间段	提交次数	文件个数
J Edit	1998/09～2012/08	6 275	1 416
Maven	2003/09～2014/01	8 845	3 051
guice	2006/08～2013/12	1 004	1 172
eclipse	2001/06～2013/10	19 123	7 346
folly	2012/06～2019/12	7 823	1 730

4.2.2　语句相似性判断阈值的选择

在 Hunk 内语句间相似对比时，通过设置语句的相似阈值，避免对不相似语句和完全相同语句的对比分析，只对较为相似的语句组进行后续的差异分析，进一步提高算法的效率。本文设相似度阈值 *SIM*，表示语句间相似程度，阈值越大，要求的相似度程度就越高。随机的抽取开源项目中一些有效数据为实验数据，分别设置阈值 *SIM* 为 0.35、0.5、0.65、0.8，得到的差异结果准确率见表 2。

表 2　不同阈值 SIM 结果

Tab. 2　Different threshold SIM results

阈值 *SIM*	检测文件的个数	正确的个数	准确率/%
0.35	36	30	82.3
0.5	34	31	91.2
0.65	37	29	78.4
0.8	31	19	61.3

根据表 2 结果，当阈值 *SIM* 为 0.35 时，判断语句为相似语句的标准较低，会导致一些无变更关系的语句也被判定为相似语句，造成差异误区；而当阈值 *SIM* 为 0.8 时，判断语句的标准较高，会使一些变更较大的语句没被判定相似语句，进而差异结果不尽合理；所以当阈值 *SIM* 为 0.5 时，语句间相似比较结果更加合理，差异结果的正确率为最优，因此后续实验阈值 *SIM* 为 0.5。

4.2.3　实验结果

表 3 为 Kdiff3、Beyonce Compared 4、和 HunkDiff 工具对 5 个开源项目中各 100 条有效数据的差异分析结果，工具 HunkDiff 的差异分析结果的正确率平均在 85%左右，相比其它工具，极大提高了差异识别的准确率。本文工具不仅较好的实现其它差异工具基本能够识别的差异问题，还较好的解决了当前存在的语句、Token 失配问题。

表 3　Kdiff3、Beyonce Compared 4、HunkDiff 差异分析结果

Tab. 3　Kdiff3、Beyonce Compared 4、HunkDiff analysis results

项目	文件总数	Kdiff3 准确率%	Beyond Compare 4 准确率%	HunkDiff 准确率%
J Edit	100	37	40	83
Maven	100	38	39	85
guice	100	36	34	89
eclipse	100	40	42	84
folly	100	55	53	87

对于上述 Kdiff3、Beyonce Compared 4 工具差异分析准确的数据，HunkDiff 工具基本也能实现对其差异分析；对差异分析不准确的数据，应用提出的算法工具对其进行了分析，分析结果见表 4。对当前工具差异不准确数据集的差异分析正确率平均为 70%以上，有效的克服了现有工具存在的差异不合理问题。

表 4　HunkDiff 分析结果

Tab. 4　HunkDiff analysis results

工具	项目	失配文件数	HunkDiff 正确数	HunkDiff 准确率%
Kdiff3	J Edit	58	40	69
	Maven	56	42	75
	guice	60	42	70
	eclipse	53	40	75
	folly	45	35	78
Beyond Compare 4	J Edit	55	39	71
	Maven	53	40	75
	guice	59	42	71
	eclipse	51	39	76
	folly	47	34	72

5　结束语

针对当前 Hunk 内二次差异化算法存在的失配问题，提出了一种基于 Token 的二次差异化算法，通过文本行向语句行的映射和语句间的相似性匹配，解决了语句失配的现象。对于更新的语句，采用基于 Token 的最长公共子序列算法，获取内部差异的 Token 集，克服了 Token 被拆分的问题。该算法当前采用 Java 语言实现，并设计了一个 HunkDiff 工具对结果进行展示。通过在 5 个开源项目的数据集上对该工具进行了实验验证，表明了该差异分析算法的可行性。该工具能够帮助软件开发者与测试者分析代码，及时发现并解决一定的软件开发、维护以及

（下转第 75 页）