ソフトウェア工学 123－4
（1999．5．14）

プログラムの正規化に基づいた差分抽出法の提案

尾崎憲幸[†] 吉田敦[††]
山本晋一郎[†††] 阿草清滋[†]

広く用いられているUNIXのdiffは，行単位での差分を出力する汎用的なツールであるが，改行や空白の挿入，削除や，コメントの変更などソースプログラムの意味に影響を与えない場合でも差分として出力する．そのため，そこからソースプログラムの意味的な変化を理解するには，多大な労力が必要である．本論文では，プログラムの意味を考慮した差分を抽出する方法として，プログラムの正規化に基づいた差分抽出法を提案する．差分抽出の前段階として，ソースプログラムを正規化し，正規化したソースプログラムの差分を抽出することで，プログラムの意味に影響を与えない差分を排除する．さらに，構文要素の「移動」について判定し，その結果を提示する．提案した抽出法に基づいてツールを実装し，実際のソースプログラムに適用することで，差分の理解が容易になることを示す．

A difference identification method based on the program normalization

NORIYUKI OZAKI,[†] ATSUSHI YOSHIDA,[††] SHIN'ICHIROU YAMAMOTO[†††]
and KIYOSHI AGUSA[†]

Diff in UNIX is a popular tool to find the differences between two files. But the differences between tow source programs found by it contain meaningless differences in semantics of programs, for example, insertion and deletion of white spaces and modification of comments. Because of this meaningless differences, we need a many effort to understand the semantical differences in source programs. In this paper, we proposed a difference identification method based on the program normalization to find semantical differences in programs. In this method, we normalize programs before the difference identification in order to remove meaningless differences. And in the identification, we detect not only the insertion and deletion of syntactic elements in programs but also the their movement. We show the effectiveness of our method in program understanding by applying our method to practical source programs.

1. はじめに

異なる二つの版のソースプログラムの差異を知るためには，ソースプログラム内の変更内容を記述したコメント，変更のログや仕様書が用いられる．しかし、変更によって新たに生じたバグへの対処が必要な場合，仕様書などから得られる変更内容の概要ではなく，ソースプログラム中の具体的かつ詳細な変更を知る必要がある．

このような場合，まず，差分抽出ツールによって差分を得て，その差分から具体的な作業内容を推測する必要がある．一般的には，UNIXのdiffなどのテキストを入力対象とし，行を単位とした差分を出力するツールが使われる．しかしソースプログラムを対象とした場合，行単位の出力はプログラムの構成単位である構文要素とは必ずしも一致せず，意味に影響を与えない変更も差分として出力される．プログラマは，頭の中で，抽出された行単位の差分を構文要素の差分に変換し，その中から実質的な差分を抽出しているため，無駄な時間と労力を費やさなければならない．

ソースプログラムを入力対象とする差分抽出ツールは，プログラムの差分理解の支援として重要である．ソースプログラムを対象とした差分抽出ツールとして，Semantic diff[1)] が既に提案されている．文献1)では，関数単位に，ステートメントリストを比較して差分を抽出している．しかし，同一の名前の関数についての比較のみを行っており，他の関数との比較を行っていない．また，ステートメントのみを扱い，各種の宣言について

† 名古屋大学工学部
School of Engineering, Nagoya University
†† 豊橋技術科学大学 知識情報工学系
Department of Knowledge-based Information Engineering, Toyohashi University of Technology
††† 愛知県立大学情報科学部
Faculty of Information Science and Technology, Aichi Prefectural University

は取り扱っていない．文献 1) では，プログラムの意味に影響を与えない差分を取り除くため，空白，コメントの差異を無視し，意味的な差異のない変数に定数値を代入するステートメントの移動を差分から除去している．しかし，文献 1) の手法では，除去されないプログラムの意味的な差異のない差分が多く残り，不十分である．

本研究では，文献 1) の手法をもとに，すべての構文要素を対象とした差分抽出を行い，各差分を「削除」「挿入」「移動」「変更」の四つに分類する．また，プログラムの意味に影響を与えない差分を除去するために，差分抽出の前段階としてプログラムの正規化を行う．正規化を採用することで，差分抽出ツールを見通し良く作成できる．

この手法に基づいてツールを実装し，実用的なプログラムのソースプログラムの二つの版に適用し，差分を抽出する．その結果を UNIX の diff の結果と比較，評価することで，本手法の効果を示す．

2. 正 規 化

ソースプログラムの表現方法を統一することで，意味に影響を与えない差分が抽出されることを防ぐことができる．正規化の内容を以下に挙げる．

- 空白、コメントの除去
  意味に影響を与えないため除去する．
- 条件文の block 補完
  条件文が単文で {} で囲まれてない場合，これを補完する．
- 複数宣言の分離
  `int x,y;` のような宣言を`int x;int y;` と分離する．構造体の型と変数が同時に宣言されている場合も複数宣言にあたる．無名構造体の宣言の場合は，適当な仮構造体名を付けて分離する．
- 関数プロトタイプ宣言の補完
  関数にプロトタイプ宣言がない場合，これを補完する．
- 宣言の辞書順並べ換え
  宣言及び関数を辞書順に並べ換えることで，意味的な差異のない宣言の「移動」を除去する．
- ステートメントの並べ換え
  依存関係のないステートメントに限り，辞書順に並べ換える．上同様，意味的な差異のないステートメントの「移動」を除去する．

正規化により，意味に影響を与えない差分を抽出することを防ぐことができる．正規化を行うにあたり，宣言の並べ換えは，複数宣言の分離及び関数プロトタイプ宣言の補完を行った後に行う必要がある．また，正規化したプログラムは構文要素の位置情報が失われているため，そのままで差分として出力すると，プログラマの直感に合う差分とはならない．従って，出力の際に，正規化したプログラムの差分から元のプログラムに基づく差分へと逆変換する必要がある．ここで挙げた以外にも行うべき正規化はあると予想されるが，正規化及びその逆変換の困難性から，これだけとした．

3. 差分の判定

差分要素である「削除」「挿入」「移動」「変更」の判定方法は，文献 1) に基づいている．文献 1) の比較対象は関数で，差分の抽出単位はステートメントだが，これを構文要素（ステートメント及び宣言）とすることで本手法に利用する．

3.1 差 分 抽 出

最小編集距離と最小コストに基づいて差分を抽出する．編集距離とは，一方から他方を得るために行う操作の回数を差し，最小の場合を最小編集距離と呼ぶ[2]．また，コストとは，構文要素の重みであり，その構文要素の重要性を表す．最小編集距離から得られた差分の中から，コストが最小となるものを選択することで，差分の挿入，削除の判定を行う．

比較する二つの構文要素のリストをA，Bとし，それぞれの構文要素の数をm，nとしたとき，$(m+1)\times(n+1)$ の最小編集距離行列 D を求めることで，最小編集距離となる挿入・削除操作のリストを得る．ここで，最小編集距離行列 D の (i,j) の要素 $D[i,j]$ は，リストAの先頭から長さ i の部分リストをリストBの先頭から長さ j の部分リストと等しいリストへ変更するときの最小編集距離である．よって，左上の要素 $D[0,0]$ から右下の要素 $D[m,n]$ へ進む経路を探索することで，リストAからリストBへの操作のリストを得ることができる．経路探索の際，要素 $D[i,j]$ から次に進むことのできる要素は以下の通りである．

- $D[i,j+1] > D[i,j]$ ならば要素 $D[i,j+1]$
- $D[i+1,j] > D[i,j]$ ならば要素 $D[i+1,j]$
- $D[i,j] = D[i+1,j+1]$ ならば要素 $D[i+1,j+1]$

また，要素 $D[0,0]$ から要素 $D[i,j]$ への経路が複数存在する場合，その経路上の挿入・削除の対象となる構文要素の重みの総和をその差分のコストとし，最小となるものを選択する。構文要素の重みは以下の通りに決定する。

- 手続き文・代入文・宣言： 参照する変数の数
- 制御文： 制御文が囲む手続き文・代入文の重みの総和
- ラベル文： 1

最終的に得られる経路から，差分のリストを得る．なお，最終的に複数の経路が得られる場合があるが，いずれも最小編集距離と最小コストの差分であり，プログラマの直感に合うため，任意の一つを選択する．

3.2 移動の判定

最小編集距離かつ最小コストとなるように得た差分には，移動操作を受けたステートメントが挿入・削除の差分としてそれぞれ単独に含まれる可能性がある．それらの差分から，等しい字句で構成されるステートメントの組を抽出し，移動の差分とする．前もって正規化を行っている場合は，宣言，ステートメントが辞書順に並べ換えられているため，変数宣言や意味に影響を与えないステートメントの移動を抽出することはなく，抽出した移動要素を再び判別し直す必要はない．なお，等しい字句で構成されるステートメントの組は1対1とは限らない．この場合，どちらに対応するのかを判定することは困難ではあるが，ここでは，ソースプログラム中で，出現位置の近いステートメントの組を移動と判定し，選択する．

3.3 変更の判定

差分の中には，変更操作を受けた構文要素の組が含まれる．これらの組の抽出は難しく，ここでは変更操作を受けた可能性が高い構文要素の組を抽出する．差分の中で，等しいもしくは変更操作を受けた構文要素に囲まれた差分群のうち，意味的に類似している構文要素の組を変更の差分とする．この定義では，正規化による構文要素の並べ換えにより，本来抽出されるべき変更差分が抽出されない可能性がある．しかし，移動操作かつ変更操作を受けた構文要素を変更差分と誤判定することを防ぐため，このような定義とする．

プログラムのある部分の差分において，削除の組のリストをA，挿入の組のリストをBとし，それぞれのステートメントの数をm, nとしたとき，$m \times n$の類似率行列Sを求める．ここで，類似率行列Sの(i, j)要素$S[i, j]$は，リストAのi番目の構文要素とリストBのj番目の構文要素の類似率である．類似率は，以下の式で表される．

$$\frac{\text{構文要素}i\text{と構文要素}j\text{で一致する字句数}}{\text{構文要素}i\text{の字句数} + \text{構文要素}j\text{の字句数}} \quad (1)$$

類似率行列Sから，以下の手順で変更を判定する．

(1) 類似度を判定する閾値をsとする．また，類似率行列Sの中で最大の要素を$S[i, j]$とする．

(2) $S[i, j] >= s$なら，ステートメントiからステートメントjへ変更とみなす．$S[i, j] < s$なら終了．

(3) 類似率行列Sの部分行列である要素$S[0, 0]$から要素$S[i-1, j-1]$を対角とする部分類似率行列S_1と，要素$S[i+1, j+1]$から要素$S[m, n]$を対角とする部分類似率行列S_2について，以上の操作を繰り返す．

閾値sは0から1の間の値であり，この値により，変更の可能性のある差分の抽出量が変化する．閾値sが1なら，変更の差文は出力されず，閾値sが0なら対応する挿入・削除の組はすべて変更と判定される．この閾値は，経験的に0.7から0.8程度とすると直感にあった変更差分が出力されるが，その決定にはさらなる検討が必要である．

3.4 差分の段階的抽出

より正確な差分を抽出するため，関数単位で以下の手順で差分を抽出する．

(1) 大域変数及び各関数ごとに差分（挿入，削除）を求める．

(2) 各関数ごとの差分（挿入，削除）から関数内での差分（移動）を求める．

(3) 残った関数ごとの差分（挿入，削除）から関数間及び関数・大域変数間での差分（移動）を求める．

(4) 残った関数ごとの差分（挿入，削除），関数間での差分（移動）をもとに，関数の削除，追加，関数名変更を求める．

(5) 残った差分（挿入，削除）から各関数・大域変数ごとに差分（変更）を求める．

(6) 差分（変更）をもとに識別子名変更を求める．

関数名の変更については，関数内部のステートメントのうち，ある閾値以上の数のステートメントが移動と判定された場合，その可能性が高いとし，出力する．変更の判定同様，この閾値の値も，検討が必要である．また，識別子の変更については，その識別子が使われているステートメントがすべて変更と判定された場合，その可能性が高いとし，出力する．

4. 実　　装

前節までの提案に基づいてツールを実装する．ツールの実装にあたり，ソースプログラムをStreamCodeに変換し，StreamCodeに対して正規化を行う．正規化したStreamCodeの差分を求め，それに対応するソースプログラムの構文を抜き出すことで差分を求める．また，ツールの作成には文字列操作が比較的簡単にできるPerlを使用した．

4.1 StreamCode

StreamCodeは，|Sapid|[3]により提供され，ソースプログラムを構文に基づいて分解し，各要素を特別な

形式で一列に並べたものである．各要素は構文要素に対応するだけでなく，その構文要素のテキスト表現も含み，各テキスト表現を組み合わせることで元のソースプログラムを再現できる．

StreamCode は，ソースプログラムについての以下の情報から構成される．

- 構文
- 記号表
- 構文要素の表現

「構文」は抽象構文木に相当する情報を表す．また，「構文要素の表現」は各構文要素がソースプログラムでどのようなテキスト表現になっているかを表す．このとき，改行やコメントなどの情報も含むため，StreamCode から元のソースプログラムを完全に再現することが可能である．

StreamCode は以下の構文から成る命令の列である．ただし，(,),<,>,*,?,_ はメタ文字であり，実際には表記されない．また，_は一つの空白を表す．

＜オペレータ＞(_＜オペランド＞)*(_:＜表現＞)?

4.2 ツールの構成

このツールは大きく分けて，ソースプログラムを StreamCode に変換し，StreamCode を正規化する正規化部，正規化した StreamCode の差分をとる差分抽出部，差分内容を表すタグを StreamCode に埋め込み，ソースプログラムを再構成し，埋め込まれたタグの部分を抜きだし出力する出力部の三つに大別される．ただし，前節までで述べた差分抽出法のうち，依存解析及び変換が困難，判定方法が確立していない，本人の能力不足という理由から，以下の部分は実装されていない．

- ステートメントの辞書順に並べ換え
- 正規化の逆変換
- 関数名，識別子名変更

4.2.1 正規化部

入力されたソースプログラム二つを，それぞれ StreamCode に変換し，正規化を行う．正規化の内容一つにつき一つのツールがあり，各ツールをパイプでつないで組み合わせることですべての正規化を行う．正規化部のソースプログラムの流れを以下に示す．

(1) ソースプログラムを StreamCode に変換
(2) 空白，コメントの除去
(3) 複数宣言の分離
(4) 関数プロトタイプ宣言の補完
(5) 宣言を辞書順に並べ換え
(6) StreamCode を構文要素単位に分離

“StreamCode を構文要素単位に分離” は，2 節では触れなかったが，StreamCode を差分抽出の最小単位に

```
struct s
   int x;
   int y;
} z;
if(i==0){
   j=0;
}else{
   j=1;
}
```

図 1 元のソースプログラム
Fig. 1 source program

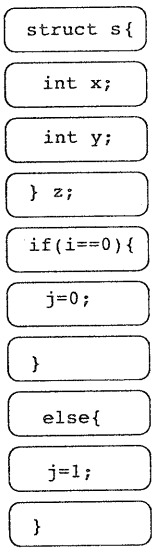

図 2 構文要素リスト
Fig. 2 syntactic list

分離する作業である．ここでの構文要素とは，「意味を保持した最小の StreamCode 群」である．単体の宣言，文は一つの構文要素として扱う．構造体や関数の宣言のブロックは，宣言開始部，宣言終了部，その内部，に分けられる．制御文は，条件部分，条件部分の終了を示す部分，その内部，に分けられる。

図 1を構文要素に分割した例を図 2に示す．図 2はソースプログラムとの対応をわかりやすくするため，構文要素に分割した StreamCode に対応するソースプログラムの断片を表している．

図 3を StrcamCode に変換し，正規化した後，再構成したものを図 4に示す．

4.2.2 差分抽出部

4.2.1 節で得られた，正規化した二つの StreamCode

```
#include<stdio.h>
typedef struct list_cell{
    int x;   /* cell */
    struct list_cell *next; /* pointer to next cell */
}list_cell_type;
void new_list_cell(list_cell_type **cell){
    *cell = (list_cell_type *)malloc(sizeof(list_cell_type));
    (*cell)->x = 0;
    (*cell)->next = NULL;
}
void add_list_cell(list_cell_type **root, int x){
    new_list_cell(root);
    (*root)->x = x;
}
void print_list(list_cell_type *root){
    int i = 0;
    while(root != NULL){
        printf("%3d %3d\n",i++, root->x);
        root = root->next;
    }
}
void main(int argc, char *argv[]){
    list_cell_type *root = NULL;
    list_cell_type *tail;
    FILE *fp;
    int x,n;
    struct {
        int x;
        int y;
    } s;
    /* Opening a File */
    if (argc > 1) {
        if ((fp = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "Can't open file: %s\n", argv[1]);
            exit(1);
        }
    }else{
        fp = stdin;
    }
    n=0;
    while(fscanf(fp,"%u",&x) != EOF){
        if(root == NULL){
            add_list_cell(&root, x);
            tail = root;
        }
        else{
            add_list_cell(&(tail->next), x);
            tail = tail->next;
        }
        n++;
    }
    fclose(fp);
    print_list(root);
}
```

図3 ソースプログラム：正規化前

Fig. 3 source program:before normalization

```
#include<stdio.h>
typedef struct list_cell{int x;
struct list_cell *next;}list_cell_type;
void add_list_cell(list_cell_type**, int);
void new_list_cell(list_cell_type**);
void print_list(list_cell_type*);
void add_list_cell(list_cell_type **root, int x){
new_list_cell(root);
(*root)->x = x;
}
void main(int argc, char *argv[]){
struct argument0{int x;
int y;};
FILE *fp;
int n;
int x;
list_cell_type *root = NULL;
list_cell_type *tail;
struct argument0 s;
if (argc > 1) {
if ((fp = fopen(argv[1], "r")) == NULL) {
fprintf(stderr, "Can't open file: %s\n", argv[1]);
exit(1);
}
}else{
fp = stdin;
}
n=0;
while(fscanf(fp,"%u",&x) != EOF){
if(root == NULL){
add_list_cell(&root, x);
tail = root;
}
        else{
add_list_cell(&(tail->next), x);
tail = tail->next;
}
n++;
}
fclose(fp);
print_list(root);
}
void new_list_cell(list_cell_type **cell){
*cell = (list_cell_type *)malloc(sizeof(list_cell_type));
(*cell)->x = 0;
(*cell)->next = NULL;
}
void print_list(list_cell_type *root){
int i = 0;
while(root != NULL){
printf("%3d %3d\n",i++, root->x);
root = root->next;
}
}
```

図4 ソースプログラム：正規化後

Fig. 4 source program:after normalization

群から，差分を3.4節の順序に従って抽出する．これも正規化部同様，各段階につき一つのプログラムが対応しているが，こちらはパイプを使わず，正規化したStreamCodeのファイル，差分情報をもったファイルを中間コードとして使用している．差分情報とは，以下の形式からなる．ただし，括弧はメタ文字で，実際には表示しない．

(差分のfile1における開始位置),(同終了位置)(差分マーク)(差分のfile2における開始位置),(同終了位置)

差分情報は構文要素ごとに対応しており，差分の位置は，StreamCodeの命令系列における位置であり，(差分の開始位置)番目のStreamCodeから(差分の終了位置)番目のStreamCodeが差分である．差分マークは，差分の内容を表し，a（挿入），d（削除），m（移動），c（変更）の4種類のいずれかである．例えば，“123,126c105,108”という差分情報は，「file1のStreamCodeの123番目から126番目からなる構文要素が，file2のStreamCodeの105番目から108番目からなる構文要素へ変更された」ことを意味する．

4.2.3 出力部

差分抽出部で得られた差分情報をStreamCodeの表現の中にタグとして埋め込む．タグは#差分マーク#（例：#a#，#d#，#c#）の形式で，これを構文要素の差分開始部，差分終了部に埋め込む．ただし，移動と判定された構文要素のファイル間の対応関係を残すため，識別番号つきのタグを使用する（例：#m0#，#m2#）．タグで囲まれた部分が差分となるが，StreamCodeから再構成されたソースプログラムには，タグが含まれているため，その部分を対応するソースプログラムの行数とともに表示する．なお，表示は，各差分要素（挿入，削除，移動，変更）ごとに行う．

4.3 実行例

入力として図5と図6のソースプログラムを与える．図5は開発途中のものであり，これに変更を加えて図6を得たとする．この二つの版のソースプログラムの差分を従来のUNIXの差分抽出ツールdiffを用いて求めた結果を図7に，本節で実装したツールを用いて求めた結果を図8に示す．図8の左側の数字は，構文要素の，正規化したStreamCodeから再構成したソースプログラム中での行番号を表す．記号“<”，“>”は，それぞれ，古い版，新しい版に出現していることを表す．また，移動の部分では，古い版の矢印のもとの数字の行から，新しい版の矢印の先の数字の行へと，構文要素が移動したことを表している．

図5と図6を比較すると，大域変数の宣言の順番が変わっている．変数宣言“int c,col;”が別々に宣言さ

```
#include <stdio.h>

char SPC = ' ';
char TAB = '\t';
char NL = '\n';
int TABSIZE = 8;

/*
 * Expand: convent tabs to equivalent number of blanks
 */
void main(int argc, char *argv[])
{

    FILE *fp;
    int  c,col;

    col = 0;

    if (argc < 2) {
        fp = stdin;
    } else {
        if ((fp = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "Can't open file: %s\n", argv[1]);
            exit(1);
        }
    }

    while((c = fgetc(fp)) != EOF) {
       if (c == NL) {
           printf("(%d)\n", col);
           col = 0;
           continue;
       }
       putchar(c);
       col = col + 1;
    }
    fclose(fp);
}
```

図5　入力ソースプログラム：変更前

Fig. 5　source program:before change

```
#include <stdio.h>

char TAB = '\t';
char SPC = ' ';
char NL = '\n';
int TABSIZE = 8;

/*
 * Expand: convent tabs to equivalent number of blanks
 */

void main(int argc, char *argv[])
{
    FILE *fp;
    int  c;
    int  col;

    /* Opening a File */
    if (argc > 1) {
        if ((fp = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "Can't open file: %s\n", argv[1]);
            exit(1);
        }
    }else{
        fp = stdin;
    }

    col = 0;
    while((c = fgetc(fp)) != EOF) {
        if (c == TAB) {
            do {
                putchar(SPC);
                col = col + 1;
            }while (col % TABSIZE != 0);
        }else{
            if (c == NL) {
                col = 0;
                continue;
            }
            col = col + 1;
            putchar(c);
        }
    }
    fclose(fp);
}
```

図6　入力ソースプログラム：変更後

Fig. 6　source program:after changed

れている．最初のif文の変更と，それに伴いthen節とelse節の中身が入れ替わっている．代入文“col = 0”がif文の前から後へ移動している．TABに関するif文，do-while文が追加されているなどの変更がある．

図8では，削除，挿入，変更，移動の順に差分を示している．図8の差分は，構文要素の位置情報が失われているが，無駄な差分が少なく，より本質的な部分が差分として出力されていることがわかる．

5. 評　　価

提案した手法を評価するために，現実的なプログラム

```
3d2
< char SPC = ' ';
4a4
> char SPC = ' ';
15,17c15,16
<       int  c,col;
<
<       col = 0;
---
>       int  c;
>       int  col;
19,21c18,19
<       if (argc < 2) {
<           fp = stdin;
<       } else {
---
>       /* Opening a File */
>       if (argc > 1) {
25a24,25
>       }else{
>           fp = stdin;
27a28
>       col = 0;
29,32c30,41
<           if (c == NL) {
<               printf("(%d)\n", col);
<               col = 0;
<               continue;
---
>           if (c == TAB) {
>               do {
>                   putchar(SPC);
>                   col = col + 1;
>               }while (col % TABSIZE != 0);
>           }else{
>               if (c == NL) {
>                   col = 0;
>                   continue;
>               }
>               col = col + 1;
>               putchar(c);
34,35d42
<           putchar(c);
<           col = col + 1;
```

図7　出力：UNIXのdiffコマンド

Fig. 7　output:UNIX diff

```
======
delete
======
11<{

13<} else

21<printf("(%d)\n", col);

===
add
===
15>else{

17>}

20>if (c == TAB) {
21>do {
22>putchar(SPC);
23>col = col + 1;
24>}while (col % TABSIZE != 0);
25>}else{

32>}

======
change
======
11<if (argc < 2)
---
10>if (argc > 1)

====
move
====
10--->18
col = 0;

12--->16
fp = stdin;

26--->30
col = col + 1;
```

図8　出力：本手法

Fig. 8　output:my tool

表1　差分の行数（UNIX の diff）
Table 1　line number of differences(diff in UNIX)

	nkf-1.5	nkf-1.6	合計
変更	100(17)	144(24)	244(41)
削除	4(0)		4(0)
挿入		69(9)	69(9)
合計	104(17)	213(33)	317(50)

表2　差分の行数（実装したツール）
Table 2　line number of differences(my tool)

	nkf-1.5	nkf-1.6	合計
変更	25	25	50
移動	4	4	8
削除	19		19
挿入		111	111
合計	48	140	188

としてnkfを取り上げ，その差分を抽出した．nkfは，文字コードを，EUC，S-JIS，JISに変換するツールであり，ソースプログラムの大きさは約1500行である．比較したバージョンは1.5と1.6である．実行時間はHITACHI9000/VR400（CPU:PA7200（100MHz）×4，Memory1920MB）上で約30分であった．従来のdiffと今回実装したツールの出力の行数を表1，表2に示す．表1，表2において，変更の行数とは，変更の差分の中に含まれたステートメントを各版ごとに分け，それぞれ行数で表したものである．表2の移動も同様である．また，表1の括弧内の数字は，抽出された差分のうち，空白，コメントのみで構成された行数である．なお，行数は，差分の種類など付加的な情報を表す行は含まない．

実装したツールの差分は，diffに比べて129行（約41％）減少，コメントなどを考慮に入れても79行（約29％）減少しており，差分を減らす効果があることが分かる．また，全体の約74％を占めていた変更が約27％に減少しており，diffの変更の約64％が，プログラムの意味を考慮しない変更であることが分かる．

実装したツールは，その出力のうち，変更が約27％，移動が約4％を占めており，全体の約三割が変更または移動した可能性の高いステートメントである．プログラマが差分を理解するにあたり，このツールを利用することで，約三割のステートメントについては可能性の高い対応関係が示されるため，差分理解の支援に有効であることが分かる．

しかし，実装したツールは入力のソースプログラムが短いにもかかわらず，実行時間が長く，実用性の点で十分ではない．これは，最小編集距離行列の経路探索の際，最小編集距離行列の要素を一つ進むたびに冗長経路の確認，除去をしているためである．明らかに冗長な経路がない場合でも，毎回確認するため，実行時間が長くなる．さらに，差分の段階抽出により，同名関数内の「移動」の差分要素の判定が，関数ごとの「移動」差分判定（3.4節2）と，関数間の「移動」差分判定（3.4節3）で二度行われており，冗長である．実装したツールでは，構文要素が一致するかどうかの判定をすべての構文要素に対して行っているが，宣言が辞書順に並んでいることを利用すれば，すべての構文要素を比較する必要はない．また，「挿入」または「削除」が連続する箇所ではどの経路を通ってもコストは一定であり，この性質を利用すれば，冗長経路の確認の際，コストの計算をする必要がなく，システムの実行時間の短縮が期待できる．

6. おわりに

本論文では，プログラム差分の理解支援のために，意味を考慮した差分を抽出する方法を述べ，それに基づいてツールを実装し，ある程度実用性のある差分が得られることを確認した．しかし，差分抽出の方法については，これだけでは完全とは言い切れず，提案した正規化以外でも，必要な正規化がある．例えば，制御文の正規化は，正規化前の状態への変換が困難と判断し，本手法では取り扱っていない．また，switch文内のラベルの並べ換え，他ラベルの文との依存関係の除去は，依存解析が困難なため，取り扱っていない．正規化部分だけでなく，変更の判定基準については，さらなる検討が必要であり，識別子変更の判定基準についても検討が必要である．

実装したツールは不完全なところが多く，提案した差分抽出法を完全には実現していない．従って，今回実装できなかった識別子変更や正規化の逆変換などを補い，改めて評価する必要がある．また，本手法で示した，変更の判定基準，及び本手法で触れていない他の正規化の必要性について，検討していくことが今後の課題である．

参考文献

1) 吉田敦, 山本晋一郎, 阿草清滋: “意味を考慮した差分抽出ツール”, 情報処理学会論文誌, Vol.38, No.6, pp.1163-1171, 1997
2) Miller, W. and Myers, E. W., “A File Comparison Program”, Software-Practice and Experience, Vol.15, No.11, pp.1025-1040, 1985
3) 吉田敦, 山本晋一郎, 阿草清滋, “ソースプログラムに対する変更操作が可能な細粒度ソフトウェア

リポジトリの提案", ソフトウェア工学の基礎 V レクチャーノート/ソフトウェア学 20, 近代科学社, pp.189-197, 1998
4) "|Sapid| home page", http://streamed.sapid.org/