SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21(7), 739–755 (JULY 1991)

Identifying Syntactic Differences Between Two Programs

WUU YANG
Computer Sciences Department, University of Wisconsin-Madison, 1210 Dayton St., Madison, WI 53706, U.S.A.

SUMMARY

Programmers frequently face the need to identify the differences between two programs, usually two different versions of a program. Text-based tools such as the UNIX® utility diff often produce unsatisfactory comparisons because they cannot accurately pinpoint the differences and because they sometimes produce irrelevant differences. Since programs have a rigid syntactic structure as described by the grammar of the programming language in which they are written, we develop a comparison algorithm that exploits knowledge of the grammar. The algorithm, which is based on a dynamic programming scheme, can point out the differences between two programs more accurately than previous text comparison tools. Finally, the two programs are pretty-printed 'synchronously' with the differences highlighted so that the differences are easily identified.

KEY WORDS cdiff Synchronous pretty-printing Syntax-directed comparator

INTRODUCTION

Programmers frequently face the need to identify the differences between two programs, usually two different versions of a program. During program development, programmers often need to know what parts of a program have been changed since a year, a week, or even an hour, ago. Software maintainers also need to identify how related versions of a program differ. This task is especially difficult when the programs being studied are older programs, or are written by other programmers. Accurately identifying the differences between program versions helps the maintainers understand the programs and eases the maintenance task. Though what constitutes a 'difference' depends on the user's purpose, finding the syntactic differences is a useful first step to finding other kinds of differences that the user may wish to identify. A syntax-directed differential program comparator is, thus, a useful tool for software development and maintenance.

Existing tools such as the UNIX utility diff are designed to compare text files rather than programs. These text comparison tools often produce unsatisfactory comparisons for programs because they cannot accurately pinpoint the syntactic differences between two programs. Moreover, the exact differences are hard to locate by looking at the output produced by these text tools. In addition, text-based

® UNIX is a registered trademark of AT&T.

Received 1 November 1990
Revised 29 January 1991

comparators sometimes produce irrelevant differences, e.g. they may consider two statements to be different due to some details, such as extra spaces or the placement of line breaks, which are usually considered by programmers to be irrelevant. Because programs have a rigid syntactic structure as described by the grammar of the programming language in which the programs are written, these irrelevant details can be filtered out by taking the grammar into account.

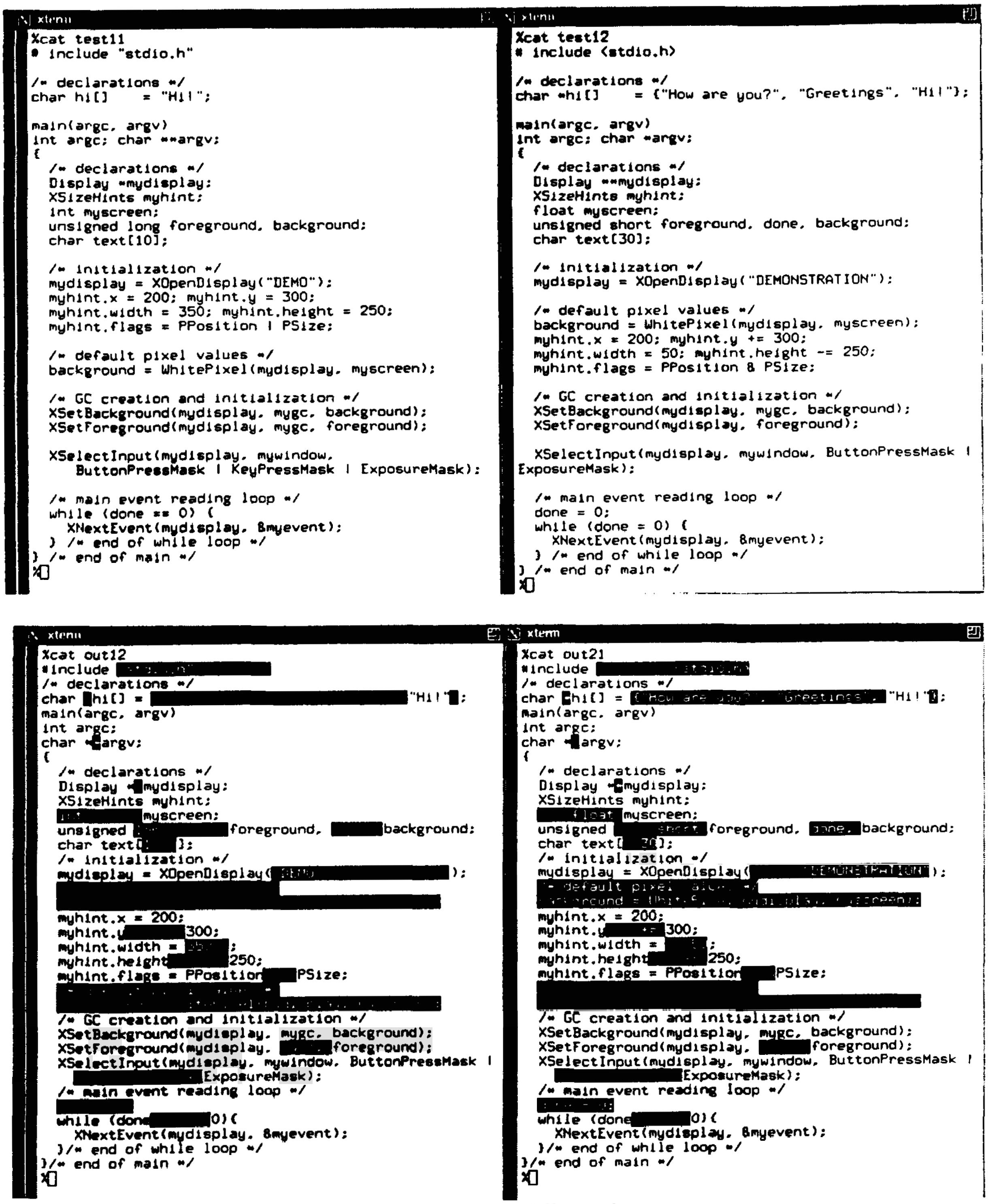
We develop a comparison algorithm that exploits knowledge of the grammar. Because the unit of comparison is individual tokens, not a whole line as used in text-based comparators, this algorithm can point out the syntactic differences more accurately than previous text comparison tools. Furthermore, irrelevant details are filtered out by a parser. The two programs are pretty-printed 'synchronously' with the differences highlighted, by reverse video, by underscores or by brighter intensity on the screen, so that the differences are easily identified.

The programs to be compared are first transformed into a variant of a parse tree[1] by a parser. A node in the tree denotes either a token, such as a variable name, or a non-terminal that represents a substructure, such as an expression. (The word 'token' is used as a synonym for the phrase 'terminal symbol', which is frequently used in compiler texts. The word 'symbol' denotes either a token or a non-terminal.) A dynamic programming scheme is applied to match nodes of the two trees. A node of a tree that does not have a corresponding node in the second tree is considered to be absent from the second tree and hence is considered to be a difference between the programs. A node of a tree that does have a corresponding node in the second tree but that contains a different symbol to the corresponding node is considered to have been changed; this node is also considered to be a difference.

The synchronous pretty-printer, like a traditional pretty-printer,[2,3] walks through the trees, prints out the tokens as the nodes containing the tokens are visited, and inserts appropriate spaces and blank lines. By 'synchronous', we mean that two trees are traversed at the same time. The traversals are arranged in such a way that corresponding nodes are always visited at the same time. Highlighted spaces for tokens that are absent from a tree (but are present in the other tree) are printed when the trees are traversed. The number of highlighted spaces is equal to the length of the token. Thus, there is a line-by-line and character-by-character correspondence when the output is examined by the programmer. This feature greatly enhances the readability of the resulting comparisons.

We have implemented a differential program comparator for the C programming language[4] based on a matching algorithm and a synchronous pretty-printing technique. Preliminary results from the implementation show that differences pointed out by our algorithm are very accurate. Figure 1 shows the comparison of two programs produced by the program comparator. The top two windows show the two original files; the bottom two windows show the comparison, in which differences can be identified immediately. The highlighted spaces and blank lines are produced by the synchronous pretty-printer. The matching algorithm and the synchronous pretty-printing technique can be easily adapted for other programming languages or text files that conform to a context-free grammar, and hence are generally applicable as a software tool.

One requirement of the differential comparator is that the programs must be 'syntactically correct', that is, the programs must conform to the grammar, for otherwise the parser cannot produce a tree representation. On the other hand, a



```
%cat test11
# include "stdio.h"

/* declarations */
char hi[]    = "Hi!";

main(argc, argv)
int argc; char **argv;
{
  /* declarations */
  Display *mydisplay;
  XSizeHints myhint;
  int myscreen;
  unsigned long foreground, background;
  char text[10];

  /* initialization */
  mydisplay = XOpenDisplay("DEMO");
  myhint.x = 200; myhint.y = 300;
  myhint.width = 350; myhint.height = 250;
  myhint.flags = PPosition | PSize;

  /* default pixel values */
  background = WhitePixel(mydisplay, myscreen);

  /* GC creation and initialization */
  XSetBackground(mydisplay, mygc, background);
  XSetForeground(mydisplay, mygc, foreground);

  XSelectInput(mydisplay, mywindow,
     ButtonPressMask | KeyPressMask | ExposureMask);

  /* main event reading loop */
  while (done == 0) {
    XNextEvent(mydisplay, &myevent);
  } /* end of while loop */
} /* end of main */
%
```

```
%cat test12
# include <stdio.h>

/* declarations */
char *hi[]    = {"How are you?", "Greetings", "Hi!"};

main(argc, argv)
int argc; char *argv;
{
  /* declarations */
  Display **mydisplay;
  XSizeHints myhint;
  float myscreen;
  unsigned short foreground, done, background;
  char text[30];

  /* initialization */
  mydisplay = XOpenDisplay("DEMONSTRATION");

  /* default pixel values */
  background = WhitePixel(mydisplay, myscreen);
  myhint.x = 200; myhint.y += 300;
  myhint.width = 50; myhint.height -= 250;
  myhint.flags = PPosition & PSize;

  /* GC creation and initialization */
  XSetBackground(mydisplay, mygc, background);
  XSetForeground(mydisplay, foreground);

  XSelectInput(mydisplay, mywindow, ButtonPressMask |
ExposureMask);

  /* main event reading loop */
  done = 0;
  while (done = 0) {
    XNextEvent(mydisplay, &myevent);
  } /* end of while loop */
} /* end of main */
%
```

```
%cat out12
#include [illegible]
/* declarations */
char hi[] = [illegible]"Hi!";
main(argc, argv)
int argc;
char **argv;
{
  /* declarations */
  Display **mydisplay;
  XSizeHints myhint;
  [illegible]myscreen;
  unsigned [illegible]foreground, [illegible]background;
  char text[[illegible]];
  /* initialization */
  mydisplay = XOpenDisplay([illegible]);
  [illegible]
  myhint.x = 200;
  myhint.y[illegible]300;
  myhint.width = [illegible];
  myhint.height[illegible]250;
  myhint.flags = PPosition[illegible]PSize;
  [illegible]
  /* GC creation and initialization */
  XSetBackground(mydisplay, mygc, background);
  XSetForeground(mydisplay, [illegible]foreground);
  XSelectInput(mydisplay, mywindow, ButtonPressMask |
    [illegible]ExposureMask);
  /* main event reading loop */
  [illegible]
  while (done[illegible]0){
    XNextEvent(mydisplay, &myevent);
  }/* end of while loop */
}/* end of main */
%
```

```
%cat out21
#include [illegible]stdio.h>
/* declarations */
char *hi[] = {"How are you?", "Greetings", "Hi!"};
main(argc, argv)
int argc;
char *argv;
{
  /* declarations */
  Display **mydisplay;
  XSizeHints myhint;
  float myscreen;
  unsigned short foreground, done, background;
  char text[30];
  /* initialization */
  mydisplay = XOpenDisplay("DEMONSTRATION");
  /* default pixel values */
  background = WhitePixel(mydisplay, myscreen);
  myhint.x = 200;
  myhint.y += 300;
  myhint.width = [illegible];
  myhint.height[illegible]250;
  myhint.flags = PPosition[illegible]PSize;
  [illegible]
  /* GC creation and initialization */
  XSetBackground(mydisplay, mygc, background);
  XSetForeground(mydisplay, [illegible]foreground);
  XSelectInput(mydisplay, mywindow, ButtonPressMask |
    [illegible]ExposureMask);
  /* main event reading loop */
  [illegible]
  while (done[illegible]0){
    XNextEvent(mydisplay, &myevent);
  }/* end of while loop */
}/* end of main */
%
```

Figure 1. Output produced by the differential comparator

'semantic error' such as an undeclared variable is tolerated. We avoid semantic checking because we want to make the tool applicable to incomplete programs as well as complete ones. In addition, semantic checking is infeasible in certain situations. For instance, in the C language, the existence of preprocessor commands renders semantic checking infeasible. Consider the case that a name is used as a variable in one region delimited by an #ifdef–#endif pair and is used as a function name in another region delimited also by an #ifdef–#endif pair. The C compiler will not complain as long as one of the two regions is skipped by the C preprocessor. But we need to consider both regions at the same time when we compare the entire C source files. Hence semantic checking cannot be enforced in this case.

It is possible to introduce error-recovering ability into the parser so that even syntactically incorrect programs can be parsed and compared. Since we do not need to do any error correction as done by a compiler—we simply require that a reasonable tree representation can be built by the parser—error recovery is easier in our case than in a compiler.

The remainder of this paper is organized into five sections, as follows. The internal form of a program, which is a variant of a parse tree, is discussed in the next section. Then the tree-matching algorithm and the synchronous pretty-printing technique are described. Experience with the comparator for the C language and some performance measurements are also presented. The last section discusses related work and concludes this paper.

PROGRAM REPRESENTATION

A program is represented internally as a variant of a parse tree, which is built by a parser. The tree representation is designed to guide the tree-matching algorithm. Before we discuss the tree representation, we consider the properties of the matching algorithm. Given two trees, the matching algorithm can find a set of pairs of nodes, one from each tree, such that a node can appear in at most one pair, nodes in the same pair contain identical symbols, and the parent–child relationship as well as the order between sibling nodes are respected. That is, the following two conditions are satisfied: (1) two nodes can match only if their parents match; (2) suppose v_1 matches v_2, w_1 matches w_2, v_1 and w_1 are siblings, and v_2 and w_2 are siblings. Then v_1 comes before w_1 if and only if v_2 comes before w_2.

The tree representation should reflect the syntactic structure and the hierarchical structure of programs so as to guide the tree-matching algorithm. The syntactic structure is used to prevent two incompatible structures, such as a data declaration and a function declaration, from being matched. The hierarchical structure is used to enforce that two structures can match only if they are at the same nesting level. The designer of the comparator has the freedom to design the appropriate representation to achieve the desired comparisons. Below we provide five guidelines for building the internal tree representation.

1. For a left-recursive production rule (or similarly right-recursive rules) in the grammar, such as

```
⟨list_exp⟩ ::= ⟨list_exp⟩"," ⟨exp⟩
⟨list_exp⟩ ::= ⟨exp⟩
```

we wish to build a flattened tree rather than a skewed tree for a structure denoted by the non-terminal ⟨list_exp⟩. For instance, the parse tree for a list of three expressions 'exp 1, exp 2, exp 3' is a skewed tree as shown in Figure 2(a). But for the tree-matching algorithm, we need a flattened tree as shown in Figure 2(b). (This point will become clear after we present the tree-matching algorithm.)

2. One problem with parse trees is that their size might be too big.[5] The problems with a big tree representation are that more memory space is needed and a longer time will be spent in matching. Therefore, the second guideline for designing the internal tree representation is to eliminate as many insignificant non-terminals as possible.

Certain non-terminals are undesirable as far as the tree-matching algorithm is concerned. For instance, the parse trees that represent the two expressions x + y and (x + y) are quite different. Owing to the pair of parentheses, there are many non-terminals in the parse tree for the second expression that do not appear in the parse tree for the first expression. These extra non-terminals prevent the first expression from being matched against the subexpression x + y of the second expression. Therefore, all the non-terminals that are incurred by the pair of parentheses should be eliminated.

There is a delicate balance between eliminating non-terminals and achieving the desired comparisons. On one hand, we wish to eliminate as many non-terminals as possible in order to reduce the size of the internal tree representation. On the other hand, certain non-terminals should be kept in the internal tree representation in order to reflect the essential syntactic structure of the programs, so that unrelated structures will not be erroneously matched. For instance, in C, there are two kinds of top-level declarations: data declarations and function declarations, denoted by the non-terminals ⟨top-level data declaration⟩ and ⟨top-level function declaration⟩, respectively. In order to avoid matching a subtree for a data declaration against a subtree for a function declaration, the two non-terminals should not be eliminated. The designer of the comparator can guide the matching algorithm by retaining the appropriate non-terminals in the internal tree representation.

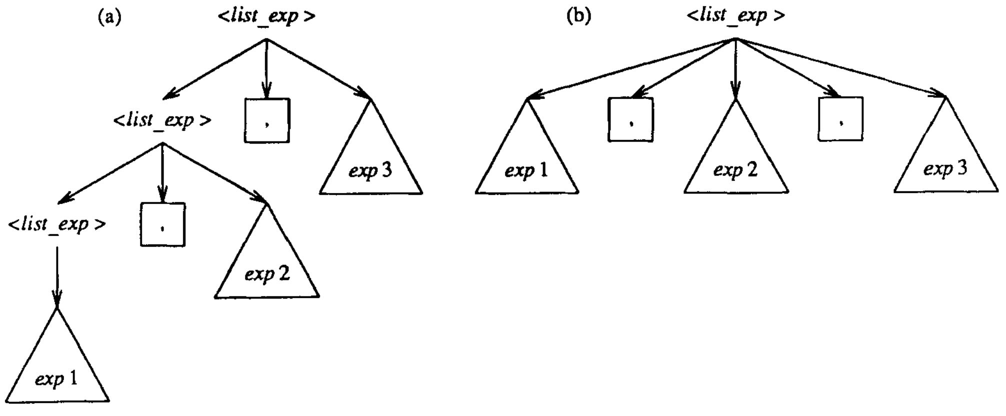


Figure 2. (a) The skewed tree for a list of three expressions; (b) the corresponding flattened tree

3. As in language-based editors, the comparison between two programs should respect the hierarchical structure of the programs. For instance, suppose the following two program fragments are to be compared:

```
while (p) {                 while (p) {
  x = y + z;                  x = y + z;
  a = b + c;                }
}                           while (p) {
                              a = b + c;
                            }
```

   The while statement on the left can match either the first while statement on the right or the second while statement, but not both. Although the shortest editing distance to change the left fragment to the right fragment would be the insertion of two lines—the third and fourth lines of the right fragment—our goal is not to find the minimum editing distance. Rather, we attempt to find the minimum *syntactic* distance (or equivalently, the maximum syntactic similarity). Suppose that the while statement on the left matches the first while statement on the right. The differences between the two program fragments should be that the assignment statement to a is deleted from the left fragment and the second while loop is added.

   Because the comparison should respect the hierarchical structure of the programs, the internal tree representation should reflect the hierarchical structure of the programs.
4. Sometimes the same syntactic structure is denoted by more than one non-terminal in the grammar in order to make the task of building the parser easier. Such non-terminals should be treated as identical symbols. For instance, in the grammar for C,[6] a while statement is denoted by either the non-terminal ⟨balanced while stmt⟩ or the non-terminal ⟨unbalanced while stmt⟩. In the internal tree representation, all while statements should be represented by the same non-terminal.
5. Unlike a parse tree or an abstract syntax tree, the internal tree representation contains nodes that represent comments and preprocessor commands. These comment and preprocessor command nodes are handled in the same way as other token nodes during comparison.

MATCHING TWO TREES

The problem of matching two trees is a natural generalization of that of matching two sequences of tokens. There exist several algorithms for sequence matching. Some are based on dynamic programming schemes to find longest common subsequences.[7–15] Others attempt to find a 'satisfactory' matching under different criteria.[16,17] The tree-matching algorithm presented in this paper is a generalization of the longest common subsequence algorithm of Hirschberg,[18] which is based on a dynamic programming scheme.

The general dynamic programming scheme for matching two sequences is shown in Figure 3, which is adapted from Reference 18. The Sequence_Matching algorithm in Figure 3 is based on the observation that the longest common subsequence of the

```
Algorithm: Sequence_Matching( $A$, $B$ )
1.  $m$ := the length of the sequence $A$.
2.  $n$ := the length of the sequence $B$.
3.  Initialization. $M[ i, 0] := 0$ for $i = 0, ..., m$.
                    $M[ 0, j ] := 0$ for $j = 0, ..., n$.
4.  for $i := 1$ to $m$ do
5.      for $j := 1$ to $n$ do
6.          $M[ i, j ] := \max ( M[ i, j-1], M[ i-1, j], M[ i-1, j-1] + W[ i, j ] )$
7.              where $W[ i, j ] = 1$ if $A_i = B_j$ and $W[ i, j ] = 0$ otherwise.
8.      od
9.  od
10. return( $M[ m, n ]$ ).
```

Figure 3. The Sequence_Matching algorithm

two sequences $A_1, \ldots, A_m$ and $B_1, \ldots, B_n$ can be computed from the longest common subsequences of $A_1, \ldots, A_m$ and $B_1, \ldots, B_{n-1}$, of $A_1, \ldots, A_{m-1}$ and $B_1, \ldots, B_n$, and of $A_1, \ldots, A_{m-1}$ and $B_1, \ldots, B_{n-1}$. The entry $M[i, j]$ denotes the length of a longest common subsequence of the two prefixes $A_1, \ldots, A_i$ and $B_1, \ldots, B_j$. The function max returns the maximum of its arguments. When the Sequence_Matching algorithm terminates, $M[m, n]$ contains the length of a longest common subsequence of the two sequences.

We may view the two sequences as two trees A and B of height 2, whose leaves are $A_1, \ldots, A_m$ and $B_1, \ldots, B_n$, respectively. Thus, the Sequence_Matching algorithm can be considered to work for a restricted class of trees, namely trees whose height is 2. We generalize the Sequence_Matching algorithm to find a maximum matching between two trees by extending the meaning of the W matrix.

A *matching* between two trees is defined to be a set of pairs of nodes, one from each tree, such that (1) two nodes in a pair contain identical symbols, (2) a node can match at most one node in the other tree, and (3) the parent–child relationship as well as the order between sibling nodes are respected. A maximum matching is a matching with the maximum number of pairs.

In the Sequence_Matching algorithm, $W[i, j]$ is either 0 or 1, depending on whether A_i and B_j are identical tokens. For general trees A and B, A_i and B_j may not be leaf nodes; instead they may be roots of first-level subtrees of A and B. In this case, $W[i, j]$ will denote the number of pairs in a maximum matching of the subtrees rooted at A_i and B_j. The entry $M[i, j]$ correspondingly denotes the number of pairs in a maximum matching between two forests of trees rooted at $A_1, \ldots, A_i$ and $B_1, \ldots, B_j$, respectively. $W[i, j]$ is computed recursively with the subtrees rooted at A_i and B_j as arguments.

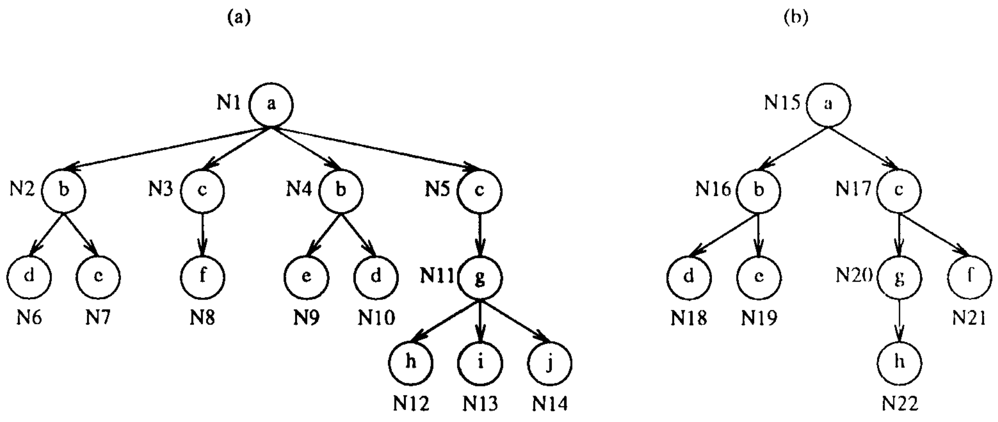
In the Simple_Tree_Matching algorithm in Figure 4, the roots of A and B are compared first. If the roots contain distinct symbols, then the two trees do not match at all. If the roots contain identical symbols, then the Simple_Tree_Matching algorithm recursively finds the number of pairs in a maximum matching between first-level subtrees of A and B, i.e. the W matrix. Based on the W matrix, a dynamic programming scheme is applied to find the number of pairs in a maximum matching between the two trees A and B. (On line 12 of the Simple_Tree_Matching algorithm, 1 is added to $M[m, n]$ to account for the fact that the roots of the trees A and B match.)

```
Algorithm: Simple_Tree_Matching( $A$, $B$ )
1.  if the roots of the two trees $A$ and $B$ contain distinct symbols then return(0).
2.  $m$ := the number of first-level subtrees of $A$.
3.  $n$ := the number of first-level subtrees of $B$.
4.  Initialization. $M[i, 0] := 0$ for $i = 0, \ldots, m$.
                    $M[0, j] := 0$ for $j = 0, \ldots, n$.
5.  for $i$ := 1 to $m$ do
6.      for $j$ := 1 to $n$ do
7.          $M[i, j] := \max( M[i, j-1], M[i-1, j], M[i-1, j-1] + W[i, j] )$
8.              where $W[i, j]$ = Simple_Tree_Matching( $A_i$, $B_j$ )
9.                  where $A_i$ and $B_j$ are the $i$th and $j$th first-level subtrees of $A$ and $B$, respectively.
10.     od
11. od
12. return( $M[m, n] + 1$ ).
```

Figure 4. The Simple_Tree_Matching *algorithm*

Example

We illustrate the Simple_Tree_Matching algorithm by applying it to the two trees shown in Figures 5(a) and (b). The nodes are numbered for ease of reference. First, the roots, nodes N1 and N15, which contain identical symbol a, are compared. Then each first-level subtree of N1 is matched against each first-level subtree of N15. The



(c)

	N16	N17
N2	3	0
N3	0	2
N4	2	0
N5	0	3

(d)

	0	1 (N16)	2 (N16-N17)
0	0	0	0
1 (N2)	0	3	3
2 (N2-N3)	0	3	5
3 (N2-N4)	0	3	5
4 (N2-N5)	0	3	6

Figure 5. (a) Tree A; (b) tree B; (c) W matrix for the first-level subtrees; (d) M matrix for the first-level subtrees

Simple_Tree_Matching algorithm is applied recursively to match the subtrees rooted at N2 and N16, which yields a value 3 for the three pairs {N2, N16}, {N6, N18} and {N7, N19}. The value resulting from matching the subtrees rooted at N2 and N17 is 0 because N2 and N17 contain distinct symbols. The value resulting from matching the subtrees rooted at N3 and N16 is also 0. The value resulting from matching the subtrees rooted at N3 and N17 is 2. The value resulting from matching the subtrees rooted at N4 and N16 is 2. Note that N9 and N19 match and N10 and N18 match, but the two pairs cannot be in the same matching because N9 comes before N10, whereas N19 comes after N18. The value resulting from matching the subtrees rooted at N5 and N17 is 3 for the three pairs {N5, N17}, {N11, N20} and {N12, N22}. Now we have computed the *W* matrix. Next the dynamic programming scheme is applied to compute the *M* matrix. When the algorithm terminates, *M*[4, 2] is 6. Therefore, the number of pairs in a maximum matching is 7. In this example, there is only one maximum matching, which consists of the following seven pairs: {N1, N15}, {N2, N16}, {N6, N18}, {N7, N19}, {N5, N17}, {N11, N20} and {N12, N22}. □

The Simple_Tree_Matching algorithm guarantees that two nodes can match only if their parents contain identical symbols. However, sometimes we wish to break the rule: we may want to match two nodes when their parents contain 'comparable' but not identical tokens. For instance, suppose that the following program fragments are to be compared:

```
while (w > 0) {                 for (i = 1; i < 10; i ++) {
  x = 1;                          x = 1;
  y = 2;                          y = 2;
  z = 3;                          z = 3;
}                               }
```

In this example, we may wish to match the three assignment statements inside the for loop against those in the while loop and consider the difference between the two program fragments to be that different looping constructs are used. But in the internal tree representation the roots of the subtrees representing for loops and while loops are distinct. By using the Simple_Tree_Matching algorithm, we are prevented from matching statements inside for loops against those inside while loops. In order to overcome this difficulty, we postulate that the symbols at the roots of for loops and of while loops are *comparable*, although they are not identical. Consequently, we distinguish *matching* nodes from *corresponding* nodes: two nodes may correspond only if they contain comparable or identical symbols, whereas two nodes match if they correspond and they contain identical symbols. By making such a distinction, we can achieve the desired matching in the above example. The Simple_Tree_Matching algorithm is, therefore, extended as in Figure 6.

The Tree_Matching algorithm in Figure 6 computes the *weight* of a maximum-weight *correspondence* between two trees. A correspondence, like a matching, is a set of pairs of corresponding nodes, one from each tree, such that (1) corresponding nodes contain comparable or identical symbols, (2) a node can correspond to at most one node in the other tree, and (3) the parent–child relationship as well as the order between sibling nodes are respected. The weight of a correspondence is the sum of the weights of each pair of corresponding nodes. The weight of a pair of

```
Algorithm: Tree_Matching( $A$, $B$ )
1.  if the roots of the two trees $A$ and $B$ do not contain comparable or identical symbols then return(0).
2.  $m$ := the number of first-level subtrees of $A$.
3.  $n$ := the number of first-level subtrees of $B$.
4.  Initialization. $M[i, 0] := 0$ for $i = 0, \dots, m$.
                    $M[0, j] := 0$ for $j = 0, \dots, n$.
5.  for $i := 1$ to $m$ do
6.     for $j := 1$ to $n$ do
7.        $M[i, j] := \max(M[i, j-1], M[i-1, j], M[i-1, j-1] + W[i, j])$
8.           where $W[i, j] = Tree\_Matching(A_i, B_j)$
9.                 where $A_i$ and $B_j$ are the $i$th and $j$th first-level subtrees of $A$ and $B$, respectively.
10.    od
11. od
12. if the roots of $A$ and $B$ contain identical symbols
13.    then $weight := M[m, n] + 1$
14.    else $weight := M[m, n]$.
15. return( $weight$ ).
```

Figure 6. The Tree_Matching *algorithm*

corresonding nodes is 1 if they contain identical symbols or 0 otherwise. (This explains lines 12 to 14 of the Tree_Matching algorithm.)

The weight assignment is flexible. The Tree_Matching algorithm can be fine-tuned by adjusting the weight assignment. Depending on the desired comparisons, different weights can be assigned to different kinds of corresponding or matching pairs. For instance, in our implementation of a differential comparator for the C language, a pair of matching nodes that contain the token comma ‘,’ has weight 2, whereas a pair of matching nodes that contain the same string constants has weight 6. This weight assignment is used because we prefer matching strings to matching commas. Consider the following two declarations.

```
char*ReservedSymbol[] = {"extern","auto"};
char*ReservedSymbol[] = {"static","extern"};
```

There are two ways to match the initialization parts of the two declarations. Either we can match the string extern or we can match the comma ‘,’. By assigning higher a weight to the pair of matching strings, the strings, rather than the comma, are matched. This kind of preference can be achieved quite easily by adjusting the weight assignment.

The weight assignment can also be used to guide the matching algorithm in yet another way. For instance, suppose that we want to compare the following two fragments.

```
x = y + z;                  x = y + z;
                            x = y + z + w;
```

The assignment statement on the left can match either of the two assignment statements on the right, yielding the same weight. But in this case, it is preferable to match the statement on the left against the first assignment because the two statements are identical. Note that the subtree for the assignment on the left is the same as the subtree for the first assignment on the right. Thus, we may add extra weight when the two subtrees are identical. That is, line 15 of the Tree_Matching algorithm is replaced by the following line:

```
15. **if** $A$ and $B$ are identical trees **then return**(*weight* + 1)
    **else return**(*weight*).
```

The test whether A and B are identical trees can be carried out by maintaining a one-bit flag in the nodes. When the trees are compared, the flags of the roots are set if the roots match and their first-level subtrees are pairwise identical. Thus, the identity test uses only a fraction of the total run time and space.

The time complexity of the three algorithms is $O(S_1S_2)$, where S_1 and S_2 are the numbers of nodes of the trees, respectively. Note that entries in the W matrix are used only once and at different times. Therefore, it is not necessary to allocate space for the whole W matrix; one variable is enough. Also note that it is not necessary to allocate space for the whole M matrix; at any time only two rows of M are needed. We also need storage for the two programs in tree form. Thus, the space requirement is $O(S_1 + S_2)$.

Once the weight of a maximum-weight correspondence is determined, it is straightforward to recover a maximum-weight correspondence. For the sake of brevity, we do not explain the recovery process.

SYNCHRONOUS PRETTY-PRINTING

Pretty-printing has been studied by several people.[2,3,19–21] A parse tree or some internal representation is traversed. Tokens of the nodes are printed when the nodes are visited. White spaces and blank lines are added at appropriate places to make the output 'pretty'. All these pretty-printing algorithms are concerned with only *one* program.

In order to highlight the differences betwween two programs, we developed a 'synchronous' pretty-printing technique, that is, two programs are pretty-printed simultaneously. The synchronous pretty-printer is best implemented by coroutines. The two trees are traversed in pre-order. The traversals are arranged in such a way that corresponding nodes will be visited at the same time.

When a node u of a tree A that does not have a corresponding node in the other tree B is visited, the token at node u is printed and highlighted on the output for A. At the same time, a sequence of highlighted spaces whose length is equal to that of the token is printed on the output for B. The traversal of A advances to the next node while the traversal of B remains at the same node.

When a pair of corresponding nodes is visited, if they contain distinct tokens, each token and some spaces are printed and highlighted on the output for the program in which the token occurs. The number of highlighted spaces is equal to the length of the token that occurs in the other tree. If they contain identical tokens, the token is printed, but not highlighted, on the outputs for both programs. The traversals of both A and B advance to the next nodes.

It is possible for the traversals to reach a point where neither node being examined has a corresponding node in the other tree. In this case, the traversal of one of the trees proceeds by itself until a node that does have a corresponding node is reached. Then the traversal of the other tree proceeds.

When non-terminal nodes are visited, blank lines and white spaces are added as in an ordinary pretty-printer. We need to be careful that the same number of blank lines and white spaces is added to both outputs. Owing to the insertion of highlighted blanks, there is a line-by-line and character-by-character correspondence. When the output is examined by the programmer, the differences can be located immediately.

AN IMPLEMENTATION FOR THE C LANGUAGE

We have implemented a differential comparator for the C language. The program, called cdiff, is based on the tree-matching algorithm and the pretty-printing technique described in the previous two sections. Figure 1 shows the comparison produced by cdiff. This section discusses our experience with cdiff.

C source files are usually a mix of preprocessor commands and C program text. Preprocessor commands serve as directives for conditional compilation, macro substitution, source line numbering, and file inclusion. Because preprocessor commands, like comments, can appear in almost any place in C source files, it is questionable whether C's grammar can be modified to accommodate preprocessor commands. We decide to treat processor commands as comments in order to avoid the related parsing problem.

This decision leads to the restriction that a program must still conform to the grammar when preprocessor commands are treated as comments. The restriction is manifested in conditional compilation and macro substitution. As discussed in the Introduction, conditional compilation renders semantic checking infeasible. Another difficulty caused by conditional compilation is demonstrated by the following example:

```
#ifdef  VAX_machines
  if (varA > 100) {
#else
  if (varA + varB > 200) {
#endif
    varB = 3;
  }
```

When the preprocessor commands #ifdef, #else and #endif are treated like comments, there are two if statements, instead of one, which prevents the parser from recognizing the structure of the program. When #ifdefs are used in this way, it is usually because the program fragments in the regions controlled by the #ifdef are too different to parametrize. A more practical solution for this problem would be to enable selective expansion of preprocessor commands. Users can selectively expand appropriate parts of the program, particularly #ifdefs contained in code. The section of an #ifdef not expanded, and hence ignored in the comparison, could be suitably highlighted by the synchronous pretty-printer.

A second problem posed by preprocessor commands is due to macro definitions and substitutions. Macro definitions such as

```
#define BEGIN {
#define END }
```

are certain to cause the parser to fail because the symbols BEGIN and END, when used in the programs, are treated as identifiers rather than the special tokens '{' and '}'. This second problem turns out to be more severe in practice because many programmers exploit the ability of the preprocessor to define a dialect of C. In case the restriction does pose a difficulty, the preprocessor can be invoked before the two programs are compared.

Note that these problems are not inherent in the comparison and pretty-printing techniques. Rather, they are caused by the parsing difficulty for C source files. For other programming languages that do not have a closely coupled preprocessor, these difficulties will not arise.

Comments, which are usually discarded during the scanner phase of a compiler, are an important constituent of C source files. It is not acceptable to ignore comments when we compare C source files. To accommodate comments, we create a new kind of non-terminal node in the internal tree representation. A comment is considered to consist of a sequence of lines, each line being a lexical unit. An alternative is to treat a comment as a sequence of blank separated tokens. This alternative approach achieves a finer granularity for comparison at the expense of producing more nodes in the trees, and consequently, taking a longer time for comparison. It is for performance reasons that we use a line as a lexical unit. The finer granularity approach can be implemented as a command-line option. Preprocessor commands are handled in a similar way. During matching, nodes for comments as well as processor commands are handled like other nodes.

There are two sets of comparable symbols used in cdiff. One set contains the terminal symbols for identifiers, numeric constants, character constants and string constants, since these terminal symbols can potentially appear in the same place in the program. The other contains the terminal symbols that denote compound statements: if, while, do, for and switch, because we wish to match the statements nested inside these compound statements.

Currently cdiff outputs the pretty-printed programs into two files. Depending on the types of the terminals used to display the output, the differences are highlighted either by reverse video, by underscores, or by brighter intensity on the screen. The differences are best viewed when the two output files are displayed in two windows sitting side by side, as shown in Figure 1.

We have used cdiff to compare release 3.2 and release 3.3 of the synthesizer generator[22] in order to measure its performance. The data were collected on a DECstation 3100 with 12 Megabytes of main memory, running Ultrix V4.0. There are 34 pairs of files that cannot be compared due to macro definitions similar to the ones mentioned above. Figure 7 illustrates the total running time versus the file sizes for the remaining 209 pairs of files.

There are two measurements of file sizes: one is the average number of lines of the two files; the other is the average number of nodes in the internal tree representations. (Because the two releases of the synthesizer generator are quite similar, each comparison involves two files of roughly the same size. It is reasonable to use the average size rather than the product of the sizes of the two programs in order to make the diagrams easier to understand.) The node-count measurement reflects programs' structural complexity more closely. On the other hand, the line-count measurement is more intuitive to the users.

From Figure 7, we can see that files whose sizes are less than 3600 lines (approximately 21,000 nodes) can be compared in less than 6 s. It can also be seen that all but one pair of files fell in this category. The largest files we compared, which contain more than 5200 lines, requires 10 s to compare. By contrast, diff takes 2·3 s to compare the pair of the largest files mentioned above, and less than 1·3 s for all other test cases. Although cdiff is slower than diff, the output of cdiff is more accurate and easier to understand.

The comparison time depends on the amount of differences between the files and the places where the differences occur. In cidff, we have incorporated some simple techniques to speed up comparison when the two files are just slightly different. For files that differ significantly, we expect the comparison time to increase (this is also true of cdiff).

The comparison time can be further reduced by augmenting the tree-matching algorithm with heuristics. A good heuristic is that two functions will be compared only if they have identical function names. In two files that contain 20 functions each, there are 400 pairs of functions that need to be compared. Adopting this heuristic limits the number of comparisons to no more than 20 pairs of functions. We expect this heuristic to reduce the comparison time dramatically, especially for large files. We plan to implement this heuristic as a command-line option for cdiff.

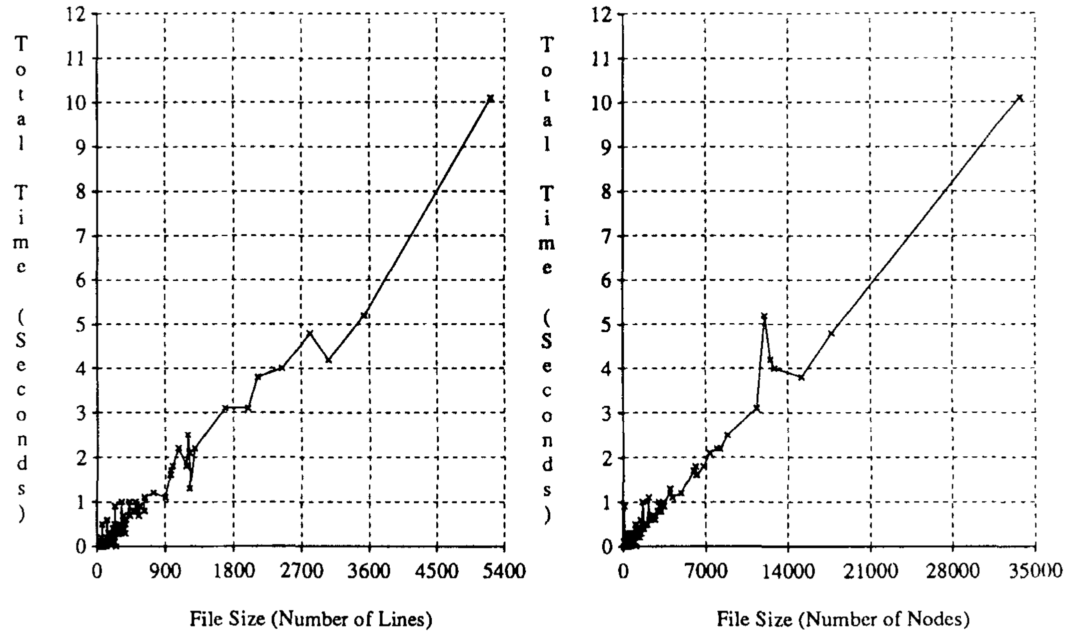


Figure 7. Performance of cdiff

CONCLUSION AND RELATED WORK

The tree-matching algorithm and the synchronous pretty-printing technique are applicable to all context-free languages, such as the specification language for yacc. It is our conclusion that the tree-matching algorithm combined with knowledge of the grammar produces more accurate comparisons and the synchronous pretty-printing technique makes comparisons easier to understand. Because context-free grammars have been idely used in programming languages and formal specifications, we believe that a syntax-directed comparator is a useful tool.

Hoffmann and O'Donnell[23] proposed an algorithm for matching two trees. In their algorithm, one tree is viewed as a pattern, which may contain wild-card symbols (i.e. symbols that can match any symbols or subtrees); the other tree is a plain tree that does not contain wild-card symbols. Their algorithm attempts to find a subtree of the plain tree that completely matches the pattern tree. Our approach differs from theirs in that we attempt to find a largest common subtree of the two trees under the requirement that the parent–child relationship and the order between sibling nodes be respected.

Zhang and Shasha,[24] Tai,[25] Lu[26] and Selkow[27] described algorithms for finding the least-cost editing sequence between two trees. The problem with the least-editing-cost approach is that it is not possible to make use of the notion of 'comparable' symbols. In these algorithms, a cost is associated with each editing operation such as changing a symbol to another, deleting a symbol or inserting a symbol. The cost of deleting (or inserting) a tree is the sum of the costs of deleting (or inserting, respectively) symbols at individual nodes. The cost of changing a symbol a to another symbol b is required to be less than the sum of the costs of changing a to a third symbol c and of changing c to b. Consider two trees whose roots contain incomparable symbols a and b as shown in Figure 8. Because a and b are incomparable, the desirable editing sequence to change tree 1 to tree 2 is to delete tree 1 and then to add tree 2. To make their algorithms produce the desired editing sequence, the cost of changing a to b should be greater than the cost of deleting tree 1 plus the cost of adding tree 2. Since the sizes of tree 1 and tree 2 may be arbitrarily large, the cost of changing a to b should be infinite. However, it is possible that there is a third symbol c that is comparable to both a and b.* Since the cost of changing a to b should be less than the cost of changing a to c plus the cost of changing c to b, either a cannot be comparable to c or c cannot be comparable to b. This difficulty

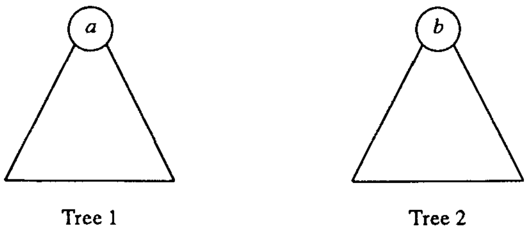


Figure 8. Two trees whose roots contain incomparable symbols, a and b

* In general, the comparability relation is not transitive. For instance, in C, an identifier is comparable to a type name since an identifier may be a (undeclared) type name. An identifier is also comparable to a decimal constant since the identifier may be a variable name. But a type can never be comparable to a decimal constant.

of cost assignment does not happen to weight assignment in the tree-matching algorithm.

Horwitz[28] proposed a set of methods for identifying the semantic and textual differences between two versions of a program. Her method requires a preprocessing step that can (conservatively) determine the equivalence classes of program components that have equivalent execution behaviours. Based on the equivalence classes, Horwitz's method proceeds to pair components under various optimization criteria, such as maximizing the number of pairs that have equivalent behaviours and texts, maximizing the number of pairs as well as the number of dependence edges between components, etc. In contrast, the algorithm presented in this paper is purely syntactic. The comparison is based solely on programs' texts and syntactic structures.

Highlighting has been used in PEDIT,[29] in which multiple versions of a program are edited at the same time, but only one version is displayed on the screen. A line that does not appear in the version being displayed but is in some other versions being edited is indicated in such a way that 'the adjacent section is highlighted'. The display algorithm in PEDIT differs from synchronous pretty-printing in that the multiple versions of a program being edited in PEDIT are actually stored in a single file with each line being tagged with a boolean expression. Synchronous pretty-printing, on the other hand, is essentially a merging operation that merges two trees into one. Another point of contrast is that synchronous pretty-printing causes highlighted blanks to be displayed.

It is possible to build a comparator generator that generates a differential comparator from the context-free grammar of the language, a specification of non-terminals to be retained in the internal tree representation, a specification of pairs of non-terminals that should be treated as the same non-terminals, a specification of pairs of non-terminals that should be considered to be comparable, and a specification of the weight of each possible kind of corresponding and matching symbol. This comparator generator can be built either from scratch or as a preprocessor for existing parser generators.

ACKNOWLEDGEMENT

The author wishes to thank Susan Horwitz and Thomas Reps for many helpful comments on this paper, for their support of this research, and for Reps' kind permission to run cdiff on two releases of the synthesizer generator. Phil Pfeiffer also helps improving the paper.

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the U.S. National Science Foundation under grants DCR-8552602 and DCR-8603356, by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590, as well as by grants from IBM, DEC, Xerox, Eastman Kodak, the Cray Research Foundation, Siemens, Hewlett-Packard and 3M.

REFERENCES

1. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.
2. D. C. Oppen, 'Prettyprinting', *ACM Trans. Programming Languages and Systems,* **2**, (4), 465–483 1980.
3. L. F. Rubin, 'Syntax-directed pretty printing—a first step towards a syntax-directed editor', *IEEE Trans. Software Engineering*, **SE-9**, (2), 119–127 (1983).

4. B. W. Kernigham and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Engleood Cliffs, NJ, 1978.
5. V. E. Waddle, 'Production trees: a compact representation of parsed programs', *ACM Trans. Programming Languages and Systems*, **21**, (1), 61–83 (1990).
6. S. P. Harbison and G. L. Steele, Jr., *C: A Reference Manual*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
7. D. Sankoff, 'Matching sequences under deletion/insertion constraints', *Proc. Nat. Acad. Sci. USA*, **69**, (1), 4–6 (1972).
8. P. H. Sellers, 'An algorithm for the distance between two finite sequences', *J. Combinatorial Theory, Series A*, **16**, 253–258 (1974).
9. R. A. Wagner and M. J. Fischer, 'The string-to-string correction problem', *J. ACM*, **21**, (1), 168–173 (1974).
10. R. Lowrance and R. A. Wagner, 'An extension to the string-to-string correction problem', *J. ACM*, **22**, (2), 177–183 (1975).
11. D. S. Hirschberg, 'Algorithms for the longest common subsequence problem', *J. ACM*, **24**, (4), 664–675 (1977).
12. N. Nakatsu, Y. Kambayashi and S. Yajima, 'A longest common subsequence algorithm suitable for similar text strings', *Acta Informatica*, **18**, 171–179 (1982).
13. W. Miller and E. W. Myers, 'A file comparison program', *Software—Practice and Experience*, **15**, (11), 1024–1040 (1985).
14. A. Apostolico and C. Guerra, 'The longest common subsequence problem revisited', *Algorithmica*, **2**, 315–336 (1987).
15. Z. Galil and R. Giancarlo, 'Speeding up dynamic programming with applications to molecular biology', *Theoretical Computer Science*, **64**, 107–118 (1989).
16. P. Heckel, 'A technique for isolating differences between files', *Comm. ACM*, **21**, (4), 264–268 (1978).
17. W. F. Tichy, 'The string-to-string correction problem with block moves', *ACM Trans. Computer Systems*, **2**, (4), 309–321 (1984).
18. D. S. Hirschberg, 'A linear space algorithm for computing maximal common subsequences', *Comm. ACM*, **18**, (6), 341–343 (1975).
19. G. A. Rose and J. Welsh, 'Formatted programming languages', *Software—Practice and Experience*, **11**, 651–669 (1981).
20. P. Mateti, 'A specification schema for indenting programs', *Software—Practice and Experience*, **13**, 163–179 (1983).
21. M. Woodman, 'Formatted syntaxes and Modula-2', *Software—Practice and Experience*, **16**, (7), 605–626 (1986).
22. T. Reps and T. Teitelbaum, 'The synthesizer generator', *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, 23–25 April, 1984; *ACM SIGPLAN Notices*, **19**, (5), 41–48 (1984).
23. C. M. Hoffmann and M. J. O'Donnell, 'Pattern matching in trees', *J. ACM*, **29**, (1), 68–95 (1982).
24. K. Zhang and D. Shasha, 'Simple fast algorithms for the editing distance between trees and related problems', *SIAM J. Computing*, **18**, (6), 1245–1262 (1989).
25. K. Tai, 'The tree-to-tree correction problem', *J. ACM*, **26**, (3), 422–433 (1979).
26. S. Lu, 'A tree-to-tree distance and its application to cluster analysis', *IEEE Trans. Pattern Analysis and Machine Intelligence*, **PAMI-1**, (2), 219–224 (1979).
27. S. M. Selkow, 'The tree-to-tree editing problem', *Information Processing Letters*, **6**, (6), 184–186 (1977).
28. S. Horwitz, 'Identifying the semantic and textual differences between two versions of a program', *Proceedings of the SIGPLAN 90 Conference on Programming Language Design and Implementation*, White Plain, New York, 20–22 June, 1990; *ACM SIGPLAN Notices*, **25**, (6), 234–245 (1990).
29. Vincent Kruskal, 'Managing multi-version programs with an editor', *IBM J. Res. Develop.*, **28**, (1), 74–81 (1984).