A Simple Method of Data Correction

KAZIMIERZ SUBIETA
Institute of Computer Science, Polish Academy of Sciences, P.O. Box 22, 00-901 Warszawa PKiN, Poland

The paper presents a simple algorithm for correcting lexical errors in data. The method employs the concept of a finite automaton. It effectively works in the experimental DBMS LINDA.

1. INTRODUCTION

The automatic correction of lexical errors has been intensively developed in the field of programming languages as an aid for the compiling process, see e.g. Refs. 2 and 4. It also proves to be useful for data input in Data Base Management Systems (DBMS), reducing the agony with data debugging. We present a simple algorithm for the automatic correction of lexical errors in input data, implemented and effectively working in the experimental DBMS LINDA.[1]

The situation when we can employ the automatic correction of lexical errors is as follows. At the input there is a new lexical element l. From the information contained, e.g. in a schema, it is known that this element should belong to a fixed set $S = \{l_1, l_2, \ldots, l_k\}$ of lexical elements. If $l \in S$, then element l is accepted. If $l \notin S$, then the system should indicate an error. Our method relies on defining the notion of 'fuzzy' belonging to a set: if there

Step	Words	Situation
1	W A L K W A L L	A A X
2	W A L K W A L L	A X A X
3	W A L K W A L L	A X A A
4	W A L K W A L L	A X X ∇
5	W A L K W A L L	∇ X ∇

Figure 1. Situations arising during the comparison of a pattern word WALK with a tested word WALL.

No.	Situations	States 1	2	3	4	5	6
1	A A A	1					
2	A A X	1					
3	A A ∇	1					
4	A X A	2346					
5	A X X	24					
6	A X ∇	24					
7	A A A A	1	2	3	4	5	5
8	A A A X	1	2		4	5	5
9	A A A ∇	1	2		4	5	5
10	A A X A	2346		3	4		5
11	A A X X	24			4		5
12	A A X ∇	24			4		5
13	A A ∇	4			4		5

No.	Situations	States 1	2	3	4	5	6
14	A X A A	1	2	3		5	
15	A X A X	1	2			5	
16	A X A ∇	1	2			5	
17	A X X A	2346		3			
18	A X X X	24					
19	A X X ∇	24					
20	A X ∇	4					
21	A ∇						
22	∇ X X X						
23	∇ X X ∇	3		3			
24	∇ X ∇	1	2			5	
25	∇ ∇	4			4		

Remark. The situations 21–25 stop the action of the automaton.

Figure 2. The table of transitions of the non-deterministic automaton.

Downloaded from https://academic.oup.com/comjnl/article/28/4/372/379180 by guest on 10 April 2024

Beginning situations	STATE
A A A A X A A A ∇	1
A X A	2
A A X X X ∇	3

Situations	SIGNAL	STATE											
		1	2	3	4	5	6	7	8	9	10	11	12
A A A A X A A ∇	1	1	0	3	9	8	11	12	8	9	0	11	12
A A A A	2	1	0	3	4	5	6	7	8	9	10	11	12
A A A X X A X ∇	3	3	8	11	0	11	11	0	11	0	0	11	0
A A X A	4	2	5	11	10	6	6	10	11	0	10	11	0
A A ∇	5	11	8	11	0	11	11	0	11	0	0	11	0
A A X A X X A ∇	6	1	9	9	9	12	0	12	12	9	0	0	12
A X A A	7	1	4	9	4	7	10	7	12	9	10	0	12
A A X X X X X ∇	8	3	0	0	0	0	0	0	0	0	0	0	0
A X X A	9	2	10	0	10	10	10	10	0	0	10	0	0
A X ∇	10	11	0	0	0	0	0	0	0	0	0	0	0
A ∇ ∇ X X X	11	0	0	0	0	0	0	0	0	0	0	0	0
∇ X X ∇	12	13	13	0	13	13	13	13	0	0	13	0	0
∇ X ∇	13	13	13	13	13	13	0	13	13	13	0	0	13
∇ ∇	14	0	13	13	0	13	13	0	13	0	0	13	0

Figure 3. The table of transitions of the deterministic automaton.

exists an element $l_i \in S$ such that l and l_i are similar in a proper sense, then the wrong element l may be replaced by l_i. Hence the core of our method is a procedure comparing two character strings and giving the information about their similarity.

The data may be organised in such a manner that the automatic correction is much easier. In DBMS such a solution is usually not justified since special error-recovery-oriented organisation is a burden for other data-base functions (e.g. updating). In the LINDA system the data organisation is not influenced by the error recovery method.

1. THE METHOD

We assume that the following mistakes in the new element should be recognised: one character is wrong, one character is superfluous, one character is missing, or two neighbouring characters are permuted. We need a procedure which has two strings as parameters and returns 0, if these strings are essentially different; and a value not equal to 0, if these strings are similar or identical. The procedure is based on employing a non-deterministic, finite automaton with six states.

Starting at the beginning of both words we compare the ith character of the pattern word with $(i-1)$th, ith and $(i+1)$th characters of the checked word. As the result of the comparison we obtain a code of a situation. This code contains three or four characters (see Fig. 1). The upper letter A means that the ith symbol of the pattern is a letter and the upper symbol ∇ denotes the end of the pattern. The lower letters code the result of the comparison of the ith letter of the pattern with the $(i+1)$th, ith and $(i+1)$th letters of the tested word, respectively, where A, the same, X, different. v, as before, denotes the end of the tested word.

Codes of situations are signals for the automaton. Its states have the following interpretation: 1, without an error; 2, one letter is wrong; 3, one letter is superfluous; 4, one letter is missing; 5, two neighbouring letters are permuted; 6, it is possible that the two letters are permuted. At the beginning the automaton is in state 1.

Downloaded from https://academic.oup.com/comjnl/article/28/4/372/379180 by guest on 10 April 2024

The situations are given at its input causing changes of its states according to Fig. 2. Since it is a non-deterministic automaton, at a given moment it may be in more than one state or in none. The last case indicates that the words are dissimilar. If after the whole sequence of situations the automaton is in any state at all, this indicates that the words are similar or identical.

The implementation of the automaton presented in Fig. 2 is quite straightforward;[3] however, it is more effective if this automaton is deterministic. Such an automaton may possess 64 states, but only 12 of them are essential, since others are not accessible from the beginning state, or indicate the end of action. The table of the deterministic automaton derived from the above is presented in Fig. 3. State 0 indicates that words are dissimilar and state 13 indicates that words are similar. Both states stop the action.

2. THE PASCAL FUNCTION

The function is based at the automaton presented in Fig. 3. It is written in PASCAL for publication and carefully tested on the computer ODRA 1305 (compatible with ICL 1900). For efficiency this function should be written in the assembly language. The parameters of the function are D (dictionary word) and N (new word). It returns **true** if these words are similar, and **false** otherwise. The function requires one space after each word (the space ends a word). Empty words are not allowed. Before the text of the function, PASCAL declarations of used data structures are given. For the table of the automaton TRANS the non-standard option **value** is used.

```
. . . . . .
type
TYPEWORD = array [1..N1] of char;
TYPEAUT = array [1..14, 1..12] of [0..13];
var TRANS : TYPEAUT;
value TRANS = (
  1, 0, 3, 9, ..., 12,
  1, 0, 3, 4, ... , 12,
  .  .  .  .
  0, 13, 13, 0, ... , 0);
  {According to table from Fig. 3.}
. . . . . .
function SIMILAR (D,N : TYPEWORD): boolean;
var I, SIGNAL, STATE : integer;
begin
  {Beginning state}
if D[1] = N[1] then STATE := 1
else if D[1) = N[2] then STATE := 2else STATE : = 3;
I := 2;
repeat
  {Coding of the signal}
  if D[I] = '∇' then SIGNAL := 3 else SIGNAL : = 0;
  if N[I−1] ≠ '∇' then
    begin
    if D[I] ≠ N[I−1] then SIGNAL := SIGNAL+5;
    if N[I] ≠ '∇' then
      begin
      if D[I] = N[I] then SIGNAL := SIGNAL+1
      else SIGNAL := SIGNAL+3;
      if D[I] = N[I+1] then SIGNAL := SIGNAL+1
      end
    else SIGNAL := SIGNAL+5
    end
  else SIGNAL := SIGNAL+11;
  {Transition according to the signal}
  STATE := TRANS [SIGNAL, STATE];
  {Next character of D}
  I := I+1
until (STATE = 0) or (STATE = 13);
SIMILAR := STATE = 13
end ; {SIMILAR}
```

REFERENCES

1. B. Kopeć, M. Kuta, W. Rzeczkowski and K. Subieta, Data base management system LINDA, Institute of Computer Science PAS Report 375, Warsaw (1979) [in Polish].
2. H. L. Morgan, Spelling correction in systems programs, *Communications of the ACM* **13**, 90–94 (1970).
3. K. Subieta, Correction of single errors in words based on patterns dictionary, *Informatyka*, **11** (2) (1976) [in Polish].
4. R. Wagner, Order-*n* correction for regular languages, *Communications of the ACM* **17**, 265–268 (1974).

Downloaded from https://academic.oup.com/comjnl/article/28/4/372/379180 by guest on 10 April 2024