Classification and Postprocessing of Documents Using an Error-correcting Parser

H. Bunke, R. Liviero
Institut für Informatik und angewandte Mathematik, University of Bern
Neubrückstr. 10, CH-3012 Bern, Switzerland
bunke@iam.unibe.ch

Abstract: In this paper an error-correcting parsing algorithm and its application to a postprocessing task in the context of automatic check processing is described. The proposed method has shown very good results in terms of recognition accuracy and execution speed on both real and synthetic data.

1 Introduction

The recognition of machine printed characters has been intensively studied during the past years and significant progress has been made [1]. For example, there exist commercial OCR systems that achieve a correct recognition rate of over 99% today [2]. But depending on the particular application, such a high recognition rate may be still insufficient. In order to further improve recognition accuracy, contextual postprocessing is often very useful. Different contextual postprocessing methods have been proposed in the literature. They are based, for example, on n-gram statistics [3,4], or dictionary search [5,6]. A recent survey on contextual processing has been given in [7]. For earlier overviews see [8,9].

In the present paper we propose the application of finite state automata and error-correcting parsing to solve a particular postprocessing problem occurring in the context of automatic check reading. The proposed method is not only an aid to recover from OCR errors but also to classify a document, i.e. a check, based on its contents in the presence of OCR errors. In the next section we give a description of the formal concepts underlying the proposed approach. The particular problem and its solution will be described in Section 3. Experimental results will be presented in Section 4. Finally, Section 5 concludes the paper.

2 Theoretical Foundations

In this section we give a brief review of error-correcting parsing, which serves as theoretical foundation of the method described in Section 3. We will restrict ourselves to regular languages as this is sufficient for the considered application. The algorithm presented in this section is a restricted version of the parser introduced in [10]. Similar algorithms have been described in [11,12]. The following definitions are taken from [13].

We consider a finite alphabet $X = \{x_1, \ldots, x_n\}$ of symbols. The set of all words over X, including the empty word ϵ, is denoted by X^*. A (deterministic) *finite state automaton* (fsa) over X is a 5-tuple $A = (Q, X, \delta, q_0, F)$ where Q is the finite set of states, $q_0 \in Q$ is the initial state, $F \subset Q$ is the set of final states, and $\delta : Q \times X \to Q$ is the transition function. The transition function can be extended to $\delta : Q \times X^* \to Q$ be defining $\delta(q, \epsilon) = q$ for any $q \in F$, and $\delta(q, xa) = \delta(\delta(q, x), a)$ for any $x \in X^*$ and $a \in X$. The language $L(A)$ accepted by a fsa A is defined by $L(A) = \{x | x \in X^* \wedge \delta(q_0, x) \in F\}$. This means that $L(A)$ consists of all words x over X for which there exists a sequence of state transitions, defined by δ, from q_0 to a final state. It is well known that the class of languages accepted by fsa's is identical to the class of languages of Chomsky-type 3 [13]. These languages are also called *regular*.

Before describing our error-correcting parsing algorithm, we need to introduce the concept of *string distance*. The errors produced by OCR devices can be classified into three types, namely deletion, insertion, and substitution of a symbol. These three types of errors are also called edit operations. In order to model the fact that certain errors are more likely to occur than others, each edit operation s gets assigned a cost $c(s)$, which is a non-negative integer. Given a sequence $S = s_1, s_2, \ldots, s_n$ of n edit operations, its cost is defined as $c(S) = \sum_{i=1}^{n} c(s_i)$. Now for any two words x and y over an alphabet X, the string distance $d(x, y)$ is defined as the minimum cost taken over all sequences of edit operations that transform x into y. Formally, $d(x, y) = min\{c(S) | S$ is a sequence of edit operations transforming x into $y\}$. Algorithms for the actual computation of $d(x, y)$ have been described in [14-16].

The task of a parser is to decide, for a given word x and language L, if $x \in L$. The fsa A of any language $L = L(A)$ can be used as a parser in a straightforward way by starting with the initial state and traversing the states of A according to δ and the actual input word x. After reading x we are in some state $\delta(q_0, x)$. Now if $\delta(q_0, x) \in F$ then we conclude $x \in L(A)$, otherwise $x \notin L(A)$. In error-correcting parsing, we are given a fsa A and a string x that does not necessarily belong to $L(A)$. If $x \in L(A)$ then the error correcting parser is supposed to report this fact. Otherwise, if $x \notin L(A)$, the error-correcting parser has to find string y such that $d(x, y) = min\{d(x, z) | z \in L(A)\}$ and outputs $d(x, y)$. In other words, it has to find that element of $L(A)$ that has the smallest edit distance to the input x.

The pseudo code of an error-correcting parsing algorithm for Chomsky-type 3 languages is given in Fig. 1. This algorithm constructs a list $L(i)$ for each input symbol x_i. Additionally, there is an initial list $L(0)$. Each list contains a number of elements of the form (q, c) where $q \in Q$ and c is the cost of a sequence of edit operations. More precisely, if $(q, c) \in L(i)$ then there exists a string y such that $d(x_1 \dots x_i, y) = c$ and $\delta(q_0, y) = q$. In other words, if $L(i)$ contains (q, c) then we know that after reading $x_1 \dots x_i$ state q can be reached if we apply a sequence of edit operations with cost c to $x_1 \dots x_i$. Furthermore, we know that there is no other such sequence of edit operations with a smaller cost. This property implies that after $L(n)$ has been constructed, the cost c of the element $(q, c) = min_{c'}\{(q', c')|q' \in F\}$ is our desired output. In the formulation of the algorithm given in Fig. 1, we assume constant cost *ins, del* and *subst* for any insertion, deletion, and substitution, respectively. These costs are global variables to the algorithm. However, the algorithm can be easily extended to the case where each insertion, deletion, and substitution may have assigned its individual cost.

The algorithm uses two functions. The function $next(q)$ returns the set of successor states of $q \in Q$ under any symbol. More precisely,

$$next(q) = \{q'|\delta(q, a) = q', a \in X\}.$$

Let $L(i)$ be a list and (q, c) a list element. Then $add((q, c), L(i))$ constructs a new list $L'(i)$ as specified below

$$L'(i) = \begin{cases} L(i) \cup (q, c) & \text{if } L(i) \text{ contains no list element } (q, c') \text{ for any } c' \\ (L(i) - (q, c') \cup (q, c) & \text{if } L(i) \text{ contains a list element } (q, c') \text{ with } c < c' \\ L(i) & \text{otherwise} \end{cases}$$

Obviously, $add((q, c), L(i))$ keeps track of the minimum cost necessary to reach a certain state after reading $x_1 \dots x_i$. It can be easily concluded that the time and space complexity of the error-correcting parsing algorithm are $O(n \cdot m)$, where n is the length of the input word and M denotes the number of states of the fsa. The correctness of the algorithm follows, as a special case, from the correctness of the algorithm described in [10].

As an example, consider the alphabet $X = \{0, 1, \dots, 9\}$, the input string $x = x_1x_2x_3 = 854$, and the fsa shown in Fig. 2, where $Q = \{A, B, C, D, E\}$, $q_0 = A$, $F = \{B, E\}$. The transition function is represented by the labeled arcs where 5-7 and 0-9 are abbreviations for 5,6,7 and $0, 1, \dots, 9$, respectively. The lists produced by the algorithm are shown in Table 1. The elements of these lists have been consecutively numbered, see column 2. Column 3 shows the list elements (q, c). From the elements $(E, 1)$ and $(B, 2)$ in $L(3)$ we conclude that the desired result is equal to 1, i.e., $min\{d(x, z)|z \in L(A)\} = 1$.

It is easy to augment the algorithm shown in Fig. 1 by pointers that allow to extract the word y in the language that has the minimum edit distance to the input, i.e. the word y with $d(x, y) = min\{d(x, z)|z \in L(A)\}$. The pointers just indicate for each list element (q, c) from which other element it has been generated by means of which edit operation. The pointers that are generated in the example are shown in the last column of Table 1. The notation DEL(i), INS(i) etc. means that the actual list element has been generated from the list element with running number i by means of a deletion, insertion etc. edit operation. If we trace back the pointers from element $(E, 1)$ in $L(3)$ to $(A, 0)$ in $L(0)$ then we see that for any string $y \in \{804, 554, 654, 754\}$ we have $d(x, y) = 1$. In conclusion, it needs at least one edit operation to transform $x = 854$ into an element $y = L(A)$, and there are four words in $L(A)$, namely 804, 554, 654, and 754 that have an edit distance to x that is equal to 1.

3 Problem Description and Proposed Solution

The application area considered in this paper is the automatic reading of checks. Over thirty different types of checks, each having an individual layout format, are commonly used for money transfer in Switzerland. An example is shown in Fig. 3. Although a large number of such checks are submitted daily, their processing at banks and post offices is only partly automated. That is, only the so-called coding line on a check is read by machine. The coding line is in the lower right part of a check. Its location is predefined and is the same for all different types, i.e. layout formats, of checks. For a graphical illustration see Fig. 3.

The coding line of each check follows a predefined format. This format, however, depends on the particular type of check. The definition of the format of the coding line of the check in Fig. 3 is given in Table 2. The format definition of another type of check is given in Table 3. The ultimate goal of automatically reading the coding line on a check is not only to correctly recognize the sequence of characters on the coding line, but also to infer the meaning of each character. That is, one wants to assign an interpretation to each character in the sense of the definition shown in Table 2 or 3. For example, when processing the check in Fig. 3, we intend to derive a result similar to Table 4.

Apparently, if the type of a check were known, the inference of the meaning of each character on the coding line would be more or less trivial because the format of the coding line for a given type of check is precisely defined. In reality, however, the type of a check is not known as only the coding line on a check - and nothing else - is captured by the scanning device. Therefore, in order to infer the meaning of each character on the coding line, we first have to determine the type of the actual check using only the sequence of characters on the coding line. Solving this task is not trivial as at least two subproblems are encountered. First, the formats of the coding lines of different types of checks may be similar to each other, and secondly, there may be OCR errors resulting in the insertion, deletion, or substitution of characters on the coding line.

An outline of the complete process of check processing is shown in Fig. 4. The OCR module digitizes the coding line on a check, extracts the individual characters, and feeds them into an OCR program. The postprocessing module compares the sequence of characters output by the OCR module to the format definitions of the coding lines and determines the type of check that fits best. This process yields the meaning of each character on the coding line (as shown in Table 4) as a by-product. Our actual comparison procedure is an error-correcting parser [10] that is controlled by a regular grammar, which describes the coding line formats of all different types of checks. In the present paper, we concentrate on the postprocessing module.

The legal symbols occurring on the coding line are from the alphabet $X = \{0, 1, \ldots, 9, <, >, +, \mathit{space}\}$. The coding line of each type of check consists of a sequence of logical units, where a logical unit is one of the following (see also Tables 2 and 3):

(1) a sequence of fixed length $l \geq 1$ of arbitrary symbols from a subset of X;

(2) one out of a finite number of constant sequences of symbols;

(3) a range of integer values;

(4) a date;

(5) a parity digit.

Obviously each of the logical units from the above list can be represented by a fsa in a straightforward way. For example, a sequence of length two of arbitrary digits can be represented by the fsa shown in Fig. 5. The fsa in Fig. 2 represents the range $[500 \ldots 809]$. The set of dates in format MMDD is defined by the fsa in Fig. 6. Also sequences of numbers with parity digits can be easily represented by fsa. Consequently, any coding line can be represented by concatenation a number of fsa's, each defining one of the types (1) to (5) from the list above.

The fsa's that represent the coding lines of the different types of checks in our system have been generated from their definitions. Given these fsa's and the sequence of symbols output by the OCR-module (see Fig. 4), the error-correcting parser described in Section 2 can be applied. It determines the most similar type of check for a given input coding line based on the minimum edit distance. Thus the actual check can be classified into one of the types defined a priori. Evaluating the pointers set by the algorithm, the meaning of the characters on the actual coding line can be determined (see Table 4).

4 Experimental Results

The error-correcting parser described in Section 2 has been implemented in C under MS-DOS and UNIX and runs on both personal computers and workstations. As the printing quality of the characters on the coding lines of the checks under consideration is generally quite good, the error rate of the OCR-module (see Fig.

4) can be expected fairly low. Consequently, we have defined an error threshold T for our error-correcting parser. As soon as the cost c of a pair (q, c) in any of the lists $L(i)$ exceeds this threshold, i.e. $c > T$, the item (q, c) is not included in $L(i)$. Practically, this prevents any item which will not contribute to the final solution from being considered, and thus speeds up the algorithm. Theoretically, it reduces the time complexity of the parser from $O(n \cdot m)$ to $O(n \cdot T)$ (see Section 2). The concrete value of T has been varied in our experiments as will be described below.

A number of experiments were done aiming at the classification of a check into its type based on the output of the OCR-module (see Fig. 4). The 14 most frequent check types, i.e. 14 different fsa's, were used in these experiments. As OCR-module, a commercial product was used. Particularly, we were interested in error rate and reliability depending on the error threshold T. Let $N = N_1 + N_2 + N_3$ where N denotes the total number of checks, and N_1, N_2, N_3 are the number of rejected, correctly, and incorrectly classified checks, respectively. From these numbers, we define the rejection rate $R = N_1/N$, the correct recognition rate $C = N_2/N$, the substitution error rate $E = N_3/N$, and the reliability rate $L = N_2/(N - N_1)$. We will say that the word x has the distance $i \geq 1$ to the language $L(A)$, $d(x, L(A)) = i$, where A is a fsa, if (1) $x \notin L(A)$, (2) there exists $y \in L(A)$ with $d(x, y) = i$, and (3) there is no $z \in L(A)$ with $d(x, z) < d(x, y)$. If $x \in L(A)$ then x has distance zero to $L(A)$.

In our first experiment, we used 2'455 coding lines that came from real checks. The result of this experiment is shown in Table 5. There were 99,27% of all checks correctly classified with $T = 0$. This means that one or more OCR error occurred in 0,73% of all checks such that a word $x \in L(A)$ was transformed into another word $x' \notin L(A)$. As $E = 0$, no word $x \in L(A)$ was transformed into another word $x' \in L(A')$, $A' \neq A$. With $T = 1$, all distorted words x with distance 1 to $L(A)$ have been correctly classified. The remaining words x' were rejected because $d(x', L(A)) > 1$ for any fsa A. Finally, setting $T = 2$, all words were correctly classified. For $T = 2$, the execution speed is over 100 documents per second on a pc.

It can be concluded from the first experiment that the error-correcting parser proposed in this paper is a very suitable tool for the classification of checks in a real world scenario. In order to reveal the limitations of the method, we did another experiment with more difficult data. That is, we artificially synthesized coding lines with varying degrees of distortion. In the process of data generation, a correct coding line $x \in L(A)$ was produced first. Then one edit operation (i.e., deletion, insertion, or substitution) after the other was applied according to some predefined probability until a word y with $d(y, L(A)) = d$ was obtained, where d was the desired edit distance, i.e. degree of distortion.

In the second experiment, a set of 2'173 syntactically correct coding lines was generated first. Then each correct coding line x was transformed into a distorted version y with $d(x, y) = d = 1, 2, \ldots, 5$. The results of this experiment, depending on d and

T, are shown in Tables 6, a) to e). As an example, we comment on the numbers given in Table 6e). All other tables have a similar interpretation. For error threshold $T = 0$, 0,28% of all distorted words y still belong to $L(A)$, while 99,72% don't belong to $L(A')$ for any fsa A'. As $E = 0$, no $x \in L(A)$ was transformed into $y \in L(A')$ with $A \neq A'$. For error threshold $T = 1$, more words are correctly classified and less are rejected. Rejection of y may be caused by the fact that $d(y, L(A')) > 1$ for any fsa A', or by a tie, i.e. by the existence of more than one fsa's A' with $d(y, L(A')) = 1$. The fact that E is greater than zero now is an indication that some $x \in L(A)$ are transformed in y such that $d(y, L(A')) = 1$ for some $A' \neq A$. The rates for $T = 2$, 3, 4 have a similar interpretation. Finally, for $T = 5$ we observe that 87,62% of all words are correctly classified. These words are characterized by the fact that if $x \in L(A)$, then the distorted version y of x fulfills $d(y, L(A)) = 5$ and $d(y, L(A')) > 5$ for any other fsa A'. Furthermore 9,89% of all strings are rejected because of ties, i.e., if $x \in L(A)$, then for its distorted version y, we have $d(y, L(A)) = d(y, L(A')) = 5$ for at least one $A' \neq A$. Moreover, 2,49% of all strings are incorrectly classified because the distorted version y of $x \in L(A)$ has a distance d smaller than 5 to some other language $L(A'), A' \neq A$. The identical numbers of E for $T = 4$ and $T = 5$ are not a coincidence. If y is a distorted version of x with i errors, then E for $T = i - 1$ and $T = i$ must be necessarly the same. The reason is that for $T = i - 1$, E is equal to the percentage of words that have a distance greater than i to A and a distance less than i to some A', $A' \neq A$. Increasing the error threshold from $i - 1$ to i doesn't result in any change of E because there is no string with a distance greater than i to A. Note that for any $T > 5$, the entities C, R, E, and L will be the same as for $T = 5$. Therefore, it is sufficient to give only values up to $T = 1, 2, 3, 4$ and 5 in Tables 6, a, b, c, d, and e, respectively.

5 Summary, Discussion and Conclusions

A postprocessing module for automatic check processing was proposed in this paper. It is based on an error correcting parser fo regular languages. The method has been tested on a large number of real and syntesized data, and has shown very good performance, in terms of classification and error-correcting accurracy, and computational efficiency. In an experiment with over 2'000 real checks, a correct classification rate of 100% has been achieved with an appropriate error threshold $T = 2$.

One additional strength of the method is that it can be easily adapted to new types of coding lines. Earlier (commercial) postprocessing modules were mainly "handcrafted", i.e. heuristically designed[1]. A serious drawback of this approach is that the whole postprocessing module has to be redesigned from scratch if a new type of check is to be taken into account, or an old one is redefined. By contrast, in the

[1] According to various personal communications.

present system, all format definitions can be kept in a database and automatically converted into their corresponding fsa[2]. Thus, any updates or modifications of the coding line format definitions can be handled by our system at almost zero cost.

A theoretical alternative to the method proposed in this paper is not to represent a coding line by means of a fsa, but by the finite set of all its possible instances, i.e. words, and to use an algorithm for string edit distance computation [14-16] instead of the error-correcting parser. As the number of different coding lines is finite for any type of check, this method is equivalent to the one proposed in this paper from the theoretical point of view. In practice, however, it can be expected much slower because of the large number of different prototype strings that are to be tested.

Finally, we would like to mention that the parser described in Section 2 is not restricted to the application described in Section 3. It is rather a general tool that may have applications in many other OCR contextual postprocessing tasks.

References

[1] Pavlidis, T. and Mori, S. (eds.): Optical Character Recognition, Special Issue of Proceedings of the IEEE, Vol. 80, No. 7, July 1992, 1027-1209

[2] Rice, S.V., Kanai, J. and Norther, T.A.: An evaluation of OCR accuracy, in UNLV Inform. Sci. Research Inst., Annual Reprot, 1993, 9-39

[3] Riseman, E.M. and Hanson, A.R.: A contextual postprocessing system for error correction using binary n-grams, IEEE Trans. on Computers, Vol. C-23, May 1974, 480-493

[4] Hull, J.J. and Srihari, S.N.: Experiments in text recognition with binary n-gram and viterbi algorithms, IEEE Trans. PAMI, Vol. PAMI-4, Sept 1982, 520-530

[5] Downtown, A.C. and Tregido, R.W.S.: The use of a trie structured dictionary as a contextual aid to recognition of handwritten british postal addresses, Proc. 1st ICDAR, Saint-Malo, France, 1991, 542-550

[6] Leroux, M., Salome, J.C. and Badard, J.: Recognition of cursive words in a small lexicon, Proc. 1st ICDAR, Saint-Malo, France, 1991, 774-782

[7] Kukich, K.: Techniques for automatically correcting words in text, ACM Comp. Surveys, Vol. 24, No. 4, 1992, 377-439

[8] Elliman, D.G. and Lancaster, I.T.: A review of segmentation and contextual analysis techniques for text recognition, Pattern Recognition, Vol. 23, No. 3/4, 1990, 337-346

[2]This feature is included in our present implementation.

[9] Srihari, S.N.(ed.): Computer Text Recognition and Error Correction, Tutorial, IEEE Computer Society Press, Silver Spring, MD, 1985

[10] Aho, A.V. and Peterson, T.G.: A minimum distance error-correcting parser for context-free languages, SIAM J. Computing 1, 1972, 305-312

[11] Wagner, A.: Order-n correction for regular languages, CACM, Vol. 17, No. 5, 1974, 265-268

[12] Myers, E. and Miller, W.: Approximate matching of regular expressions, Bulletin of Math. Biology, Vol. 51, No. 1, 1989, 5-37

[13] Hopcroft. J. and Ullman, J.: Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979

[14] Wagner, R.A. and Fischer, M.J.: String-to-string correction problem, Journal of the ACM, Vol. 21, No. 1, 1974, 168-173

[15] Ukkonen, E.: Algorithms for approximate string matching, Inform. and Control, Vol. 64, 1985, 100-118

[16] Bunke, H.: A fast algorithm for finding the nearest neighbor of a word in a dictionary, Proc. 2nd ICDAR, Tonkuba City, 1993, 632-637

```
algorithm error-correcting parser
input:    a fsa $A = (Q, X, \delta, q_0, F)$ and an input word $x = x_1 \ldots x_n$
output: $d = min\{d(x,z) | z \in L(A)\}$
method:
/*initialization*/
$L(0) := \{(q_0, 0)\}$;
for $i = 1$ to $n$ do $L(i) = \emptyset$;
/*main loop*/
for $i = 1$ to $n$ do {
    repeat {
        for all $(q, c) \in L(i)$ do {
            add$[(q, c + ins), L(i+1)]$; /*insertion of $x_{i+1}$*/
            for all $q' \in next(q)$ do {
                add$[(q', c + del), L(i)]$; /* deletion of symbol $a$ from automaton*/
                if $\delta(q, x_{i+1}) = q'$ then add$[(q', c), L(i+1)]$ /*match, i.e. $a = x_{i+1}$*/
                else add$[(q', c + sub), L(i+1)]$}}} /*substitution of $a$ by $x_{i+1}$*/
    until no more elements can be added to $L(i)$}
$d := min_c\{(q, c) | (q, c) \in L(n), q \in F\}$
end error-correcting parser
```

Figure 1: The error-correcting parsing algorithm in pseudo-code

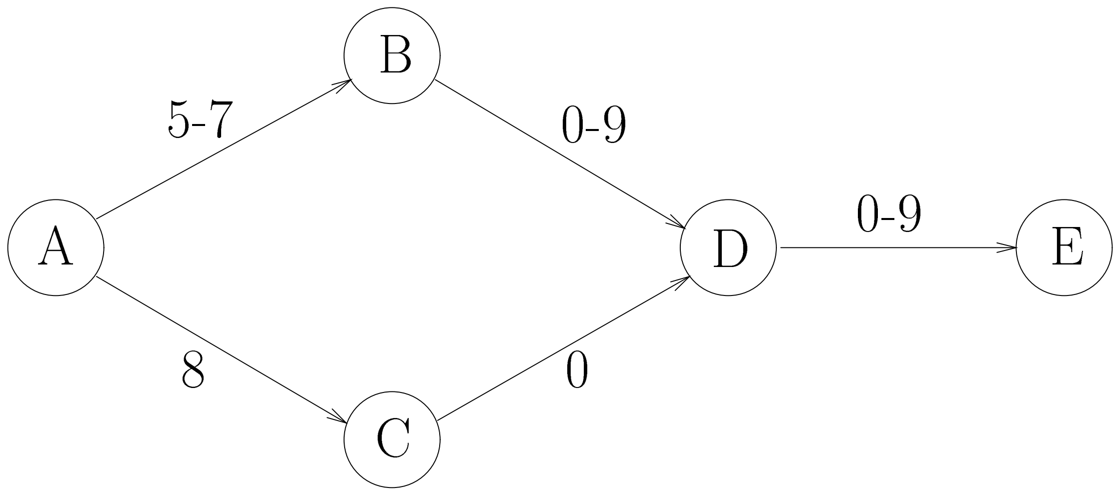

Figure 2: Example of finite state automaton

Figure 3: Example of a check

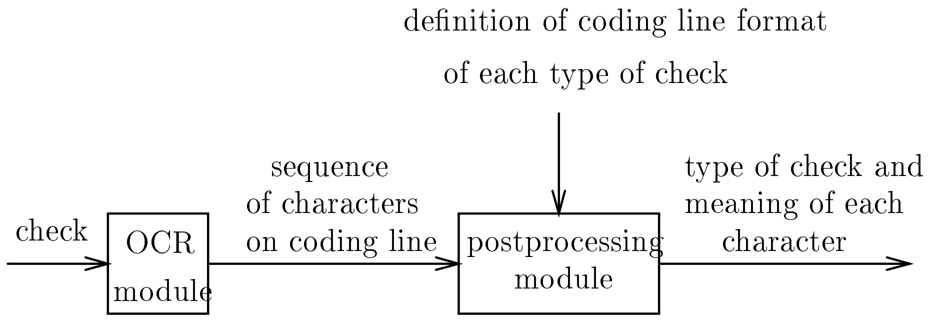
definition of coding line format
of each type of check
check
OCR module
sequence of characters on coding line
postprocessing module
type of check and meaning of each character

Figure 4: Overview of check processing system

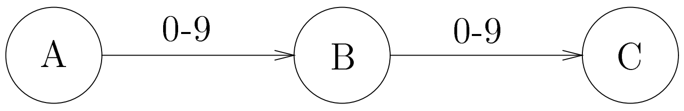

Figure 5: A fsa representing all sequences of digits of length two

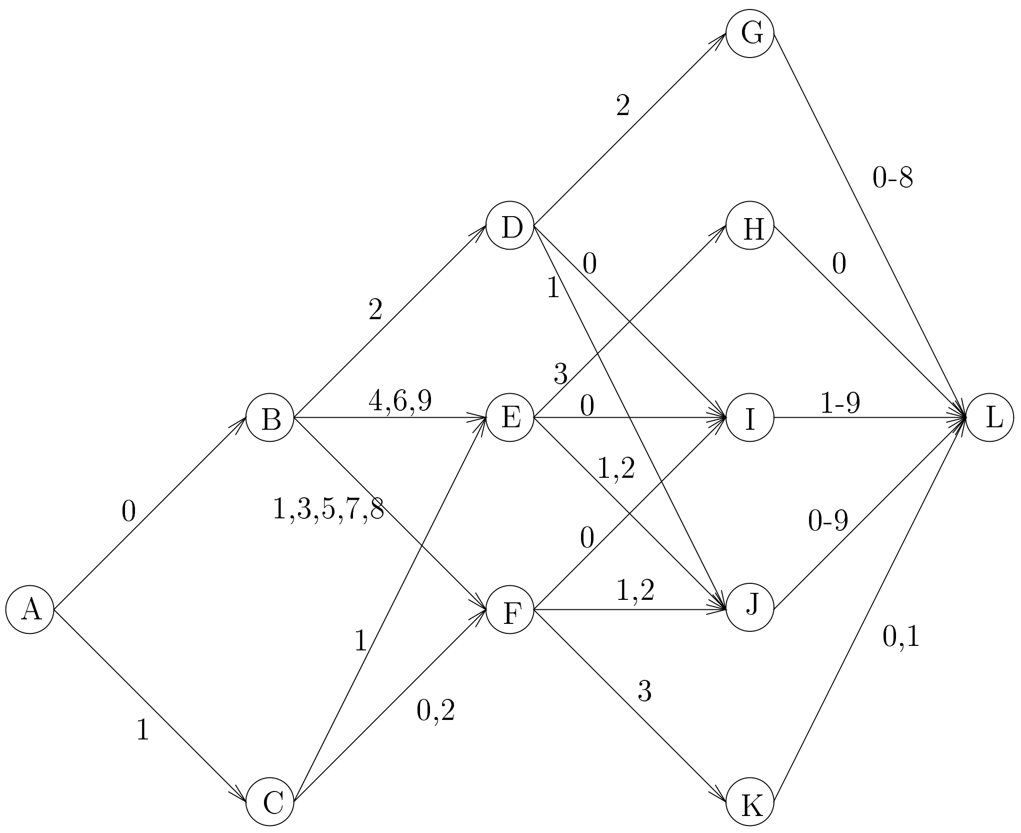

Figure 6: A fsa representing the set of dates in format MMDD.

list	running number	list element	pointer
L(0)	1	(A,0)	
	2	(B,1)	DEL(1)
	3	(C,1)	DEL(1)
	4	(D,2)	DEL(2), DEL(3)
	5	(E,3)	DEL(4)
L(1)	6	(A,1)	INS(1)
	7	(B,1)	SUB(1)
	8	(C,0)	MATCH(1)
	9	(D,1)	MATCH(2), DEL(8)
	10	(E,2)	MATCH(4), DEL(9)
L(2)	11	(A,2)	INS(6)
	12	(B,1)	MATCH(6)
	13	(C,1)	INS(8)
	14	(D,1)	MATCH(7), SUB(8)
	15	(E,1)	MATCH(9)
L(3)	16	(A,3)	INS(11)
	17	(B,2)	INS(12)
	18	(D,1)	MATCH(12)
	19	(E,1)	MATCH(14)
	20	(C,2)	INS(13)

Table 1: Result of error-correcting parsing

position	meaning	possible value
1-2	check subcategorie	one out of $\{01, 03, 11\}$
3-12	amount (same specified on check)	any sequence of digits
13	parity digit 1	parity check for position 1-12
14	delimiter	$>$
15-40	reference number	any sequence
41	parity digit 2	parity check for positions 15-40
42-43	delimiter	+ *space*
44-51	customer identification	any sequence of digits
52	parity digit 3	parity check for positions 44-51
53	delimiter	$>$

Table 2: Format definition of coding line on the check in Fig. 3.

position	meaning	possible value
1-2	check subcategorie	one out of $\{46, 47, 56, 57\}$
3	parity digit 1	parity check for positoin 1-2
4	delimiter	$>$
5-24	reference number	any sequence of digits
25-30	deadline	date in format YYMMDD
31	parity digit 2	parity check for positions 5-30
32-33	delimiter	+ *space*
34-41	customer identification	any sequence of digits
42	parity digit 3	parity check for positions 34-41
43	delimiter	$>$

Table 3: Format definition of coding line of another type of check.

type of information	value
check subcategorie	01
amount	187.50 Fr.
reference number	20011282367002209310248139
customer identification	01000064

Table 4: Result of automatic processing of Fig. 3.

	T=0	T=1	T=2
C	99.27	99.67	100.00
R	0.73	0.33	0.00
E	0.00	0.00	0.00
L	100.00	100.00	100.00

Table 5: Result of the first experiment (R = rejection rate, E = error rate, L = reliability)

	T=0	T=1
C	24.25	99.26
R	75.61	0.60
E	0.14	0.14
L	99.43	99.86

a)

	T=0	T=1	T=2
C	7.82	46.71	97.38
R	92.18	53.11	2.44
E	0.00	0.18	0.18
L	100.00	99.62	99.82

b)

	T=0	T=1	T=2	T=3
C	3.22	24.11	64.80	94.75
R	96.78	75.66	34.51	4.56
E	0.00	0.23	0.69	0.69
L	100.00	99.06	98.95	99.28

c)

	T=0	T=1	T=2	T=3	T=4
C	1.01	11.41	40.04	73.77	90.70
R	98.90	88.27	58.95	24.44	7.50
E	0.09	0.32	1.01	1.79	1.79
L	91.82	97.27	97.54	97.63	98.05

d)

	T=0	T=1	T=2	T=3	T=4	T=5
C	0.28	4.74	21.54	50.85	77.45	87.62
R	99.72	95.12	77.63	47.17	20.06	9.89
E	0.00	0.14	0.83	1.98	2.49	2.49
L	100.00	97.13	96.29	96.25	96.89	97.24

e)

Table 6: Results of the second experiment:
a) edit distance $d = 1$
b) edit distance $d = 2$
c) edit distance $d = 3$
d) edit distance $d = 4$
e) edit distance $d = 5$