On the Levenshtein Automaton and the Size of the Neighborhood of a Word

Hélène Touzet

On the Levenshtein Automaton and the Size of the Neighbourhood of a Word

Hélène Touzet

CRIStAL (UMR CNRS 9189 University of Lille) and Inria, France
helene.touzet@univ-lille1.fr

Abstract. Given a word P and a maximal number of errors k, we address the problem of counting the number of strings whose Levenshtein distance to P does not exceed k. We give an algorithm that scales linearly with the size of P and that is based on a variant of the classical Levenshtein automaton.

1 Introduction

The problem of measuring the similarity between two strings arises in many areas such as computational molecular biology, natural language processing, spelling correction, plagiarism detection, music information retrieval. A common metric for it is the *Levenshtein distance*, also simply called the Edit Distance. This distance is defined as the smallest number of substitutions, insertions, and deletions of symbols required to transform one of the words into the other.

In this paper, we investigate the basic problem of the size of the neighbourhood of a given string: count how many strings are within a bounded distance of a fixed reference string. This problem has been exposed in [6, 2], among others. As far as we know, there is no efficient algorithm for solving it. We propose a dynamic programming algorithm that runs linearly in the length of the pattern. This algorithm heavily relies upon the *Universal Levenshtein Automaton*, which is an advanced automaton for the Levenshtein distance problem introduced in [4, 5]. This automaton is *universal* in the sense that it does not depend on the two strings that have to be compared.

The paper is organized as follows. In Section 2, we present the Universal Levenshtein Automaton. We review its main principles, and revisit them by proposing some new features, such as the introduction of a *Nondeterministic Universal Levenshtein Automaton*. In Section 3, we present the algorithm to compute the cardinality of the neighbourhood of a word.

2 Definition and Construction of the Deterministic Universal Levenshtein Automaton

2.1 Preliminaries and notations

Let Σ be a finite alphabet, and let P and V be two strings over Σ. The *Levenshtein distance* between P and V, denoted $Lev(P, V)$, is the smallest number of edit

operations needed to transform P into V, where the allowed operations are *substitution* of one symbol with another, *deletion* of a symbol and *insertion* of a symbol. A sequence of operations transforming P into V is called an *edit script*. For any pair of strings P and V, the distance between P and V can be computed by dynamic programming in time proportional to the product of the length of P and V with the Wagner-Fischer algorithm [9].

Assume now that we are given a fixed threshold for the number of errors k. We consider the decision problem associated to the Levenshtein distance: for any pair of strings P and V, decide whether the Levenshtein distance is lesser than or equal to k. In this decision problem, we can also introduce the additional hypothesis that the pattern P is fixed, and consider the *neighbourhood* of P, noted $\mathcal{L}\text{ev}(P,k)$: $\mathcal{L}\text{ev}(P,k) = \{V \in \Sigma^*;\ Lev(P,V) \leq k\}$.

When P is fixed, it is highly desirable to preprocess P, so that the decision problem can be solved more efficiently than with the dynamic programming algorithm. There has been an abundance of literature on this subject. A standard approach is to start from the *Levenshtein automaton*, depicted in Figure 1, which recognizes the set of strings which are at most at distance k to P. This Levenshtein automaton is a Nondeterministic Finite Automaton (NFA). A run can be simulated using dynamic programming or bit parallelism [1, 3]. However, it is of little help to count the number of accepted strings, because distinct paths in this NFA can recognize the same string. Another possibility is to transform the Levenshtein automaton into an equivalent Deterministic Finite Automaton (DFA) [8, 7]. This is a tedious task, that should be performed for each target string P.

In [4, 5], a universal Levenshtein automaton was introduced. The term *universal* conveys its one-time construction and independency of the two input strings. Thus it can be applied to any pair of strings P and V of arbitrary length over any arbitrary alphabet Σ. This new automaton is based upon insightful observations of the nondeterministic Levenshtein automaton for a fixed word, and extends the work done for the determinization algorithm described by the same authors in [7].

The remainder of this section is devoted to a thorough presentation of the Universal Levenshtein Automaton. First, in Section 2.2, we explain how to convert the two input strings, P and V, into a single string that contains the required information to determine their distance. Then, in Subsections 2.3 and 2.4, we explain how to construct the Universal Levenshtein Automaton.

Notation: For a word $V \in \Sigma^*$, $|V|$ is the length of this word, and $|V|_\Sigma$ is the number of distinct symbols of Σ present in V. For each positions i, j in V ($1 \leq i \leq |V|$), V_i is the ith letter of V and $V[i..j]$ is the subtring of V starting at position i and ending at position j. ε denotes the empty string.

2.2 Bit vector representation

Let P and V be the two strings. In the two-dimensional dynamic programming table of the Wagner-Fischer algorithm, all edit scripts with at most k errors have

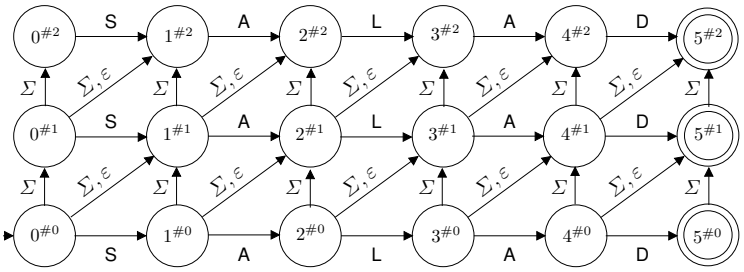


Fig. 1. Levenshtein automaton for the word SALAD. In general, this automaton has $m+1$ columns and $k+1$ levels. The $i^{\#x}$ notation for each state corresponds to i symbols read in the pattern and x errors recorded. Horizontal transitions represent identities, vertical transitions represent insertions, and the two types of diagonal transitions represent substitutions (Σ) and deletions (ε), respectively.

a path that stays around the diagonal. The width of this diagonal is $2k + 1$. To compute the values in this portion of the table, it is sufficient to know which coordinates (i, j) of the cells are such that $P_i = V_j$. This information captures the similarity between the two strings P and V. We show an example in Figure 2 (left), where grey cells represent coordinates such that $P_i = V_j$, and white cells represent other coordinates. At this point, one can forget the two input words, P and V.

The idea of the Universal Levenshtein Automaton is to work directly on these configurations of white and grey cells. For this, each horizontal line of the diagonal is represented by a bit vector of length $2k + 1$: 0 for white cells, and 1 for grey cells. These bit vectors will serve as the new alphabet, rather than Σ. This encoding captures the local structural properties of the input words, and guarantees alphabet independence.

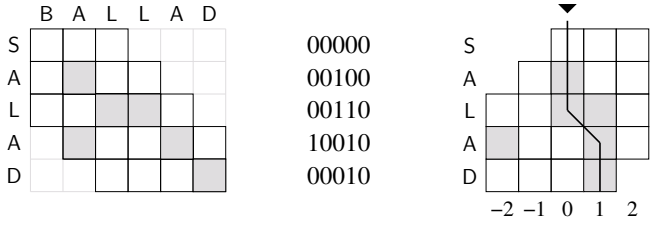


Fig. 2. Bit vector representation for the string SALAD with respect to BALLAD

Definition 1. *Let $P \in \Sigma^*$ and let $s \in \Sigma$. The* characteristic vector $\chi(s, P)$ *is the bit vector of length $|P|$ such that the ith bit is 1 if $s = P_i$, and 0 otherwise.*

Definition 2. *Let $P \in \Sigma^m$, $V \in \Sigma^n$ such that $n \leq m+k$. Let $P' = \$^k P \$^{2k}$ and $V' = V\$^{m-n+k}$, where \$ is a new symbol not present in Σ. The* k-encoding of V with respect to P *is the sequence of $m+k$ bit vectors of length $2k+1$ such that the jth element is the characteristic vector $\chi(V'_j, P'[j-k..j+k])$.*

The new symbol \$ is a sentinel character that serves two purposes. It is added to the prefix and to the suffix of P to standardize the length of bit vectors. Additionnaly, it is added to the suffix of V to deal with edit scripts that end with one or several deletions of symbols of P. These operations will be treated as substitutions with \$. Note that the k-encoding of V with respect to P is not defined when the length of V exceeds the length of P by more than k characters. Indeed, there is no point in asking $Lev(P, V) \leq k$ in this case.

Example 3. 2-encodings with respect to the pattern BALLAD.

	B	A	L	L	A	D		
S A L A D	00000	00100	00110	10010	00010	00011	00111	01111
B A L D	00100	00100	00110	00001	00001	00011	00111	01111
B A L L	00100	00100	00110	00110	00001	00011	00111	01111
B A L L A D S	00100	00100	00110	01100	00100	00100	00000	01111

Finally, we define $\overline{\mathcal{L}\text{ev}}(P, k)$ as the set of bit vector sequences u in $(\{0,1\}^{2k+1})^*$ such that there exists V in $\mathcal{L}\text{ev}(P, k)$ whose k-encoding wrt P is u.

Remark 4. In [4], the authors use a different encoding. They have bit vectors of length $2k+2$ bits, instead of $2k+1$. The last bit is used to identify the transition between non-accepting and accepting states. Here, we get rid of this additional bit by adding \$ symbols to the suffix of V. In Subsection 2.4, it will allow us to obtain a smaller automaton.

2.3 Construction of the Nondeterministic Universal Levenshtein Automaton

In [4], the authors directly build the *Deterministic Universal Levenshtein Automaton* from the Levenshtein automaton for a fixed word. They consider a symbolic triangular area of a state to simulate multiple active states. Here we take a different approach, and introduce a *Nondeterministic Universal Levenshtein Automaton*. We find this intermediate step useful to facilitate understanding. Moreover it allows us to give an effective algorithm to build the Deterministic Universal Levenshtein Automaton, which is not so easy to infer from [4][1].

Let us come back to the table in Figure 2. An edit script between P and V can be seen as a path in the white and grey grid. The portions of the path that stay in the same lane, correspond to a series of identity and substitution operations. In this context, traversing a white cell costs one error. Everytime you change lanes, you have to pay either for an insertion (moving to the left) or a

[1] The authors write: "We describe only the basic idea."

deletion (moving to the right). To capture this idea, we introduce states of the form (x, y), meaning "I am in the lane y, and have made x errors so far".

We now examine which are the outgoing transitions for the state (x, y). If we prefer not to have ε-transitions, the automaton should read a bit vector from its input sequence at each time step. So each transition should consume exactly one symbol v of V. In a first approach, an edit script can be decomposed into a series of one the two following basic events: an insertion of v, or a series of ℓ deletions in P ($0 \leq \ell \leq k - x$), followed by either a substitution or an identity with v. In this decomposition, we make the usual hypothesis that a deletion is not followed by an insertion. To reduce the nondeterminism, we add another local condition: a deletion is not followed by a substitution. Indeed, it is always possible to intervert the two operations, and to apply the substitution before the deletion. Doing so, we obtain three types of edit events.

- ins: insertion of v,
- sub: substitution of v,
- ℓ del+id: ℓ deletions in P ($0 \leq \ell \leq k$), followed by an identity with v.

ins makes the automaton transits from (x, y) to $(x + 1, y - 1)$, sub from (x, y) to $(x + 1, y)$ and ℓ del+id from (x, y) to $(x + \ell, y + \ell)$. Figure 3 shows all these transitions.

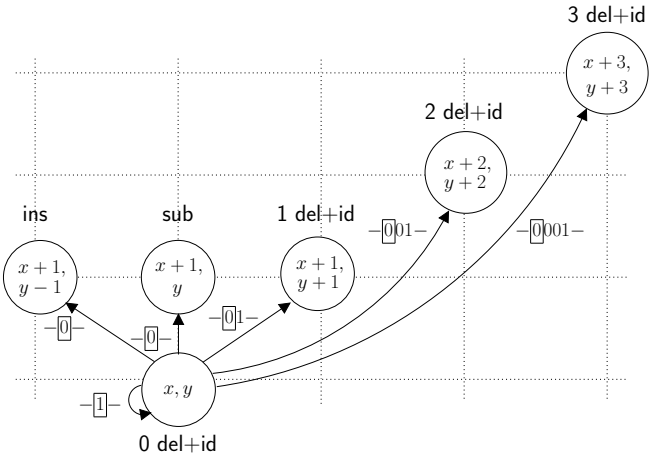


Fig. 3. Outgoing transitions for the state (x, y), $k = 3$. For each bit vector, the bit in position $k + y + 1$ corresponding to the lane y is framed, and $-$ is used to denote any sequence of bits.

We are now ready to formally define NULA(k), the *Nondeterministic Universal Levenshtein Automaton* for k errors.

Definition 5. *Let k be a positive number. The* Nondeterministic Universal Levenshtein Automaton *for k, denoted* NULA(k)*, is the NFA represented as follows.*

- *the input alphabet is* $\{0,1\}^{2k+1}$,
- *the set of states* Q_k *is* $\{(x,y) \in \mathbb{N} \times \mathbb{Z};\ 0 \leq x \leq k, -x \leq y \leq x\}$,
- *the transition function* $\Delta_k : Q_k \times \{0,1\}^{2k+1} \to P(Q_k)$ *is constituted of three types of transitions.*

  insertion transitions*:* $(x+1, y-1) \in \Delta_k((x,y),u)$ *for all states* $(x,y) \in Q_k$ *such that* $x < k$ *and all* $u \in \{0,1\}^{k+y}0\{0,1\}^{k-y}$

  substitution transitions*:* $(x+1, y) \in \Delta_k((x,y),u)$ *for all states* $(x,y) \in Q_k$*, such that* $x < k$ *and all* $u \in \{0,1\}^{k+y}0\{0,1\}^{k-y}$

  deletion+identity transitions*:* $(x+\ell, y+\ell) \in \Delta_k((x,y),u)$ *for all states* $(x,y) \in Q_k$*, all* ℓ *such that* $0 \leq \ell \leq k-x$ *and all* $u \in \{0,1\}^{k+y}0^{\ell}1\{0,1\}^{k-y-\ell}$
- *the start state is* $(0,0)$,
- *all states are accepting.*

Figure 4 shows NULA(2). The automaton is very regular and easy to construct. The reader can check that all 2-encodings of Example 3 are accepted. In general, NULA(k) can be seen as an extension of NULA($k-1$): just add $2k+1$ new states of the form (k,y) and incoming transitions. Thus NULA(k) has $(k+1)^2$ states.

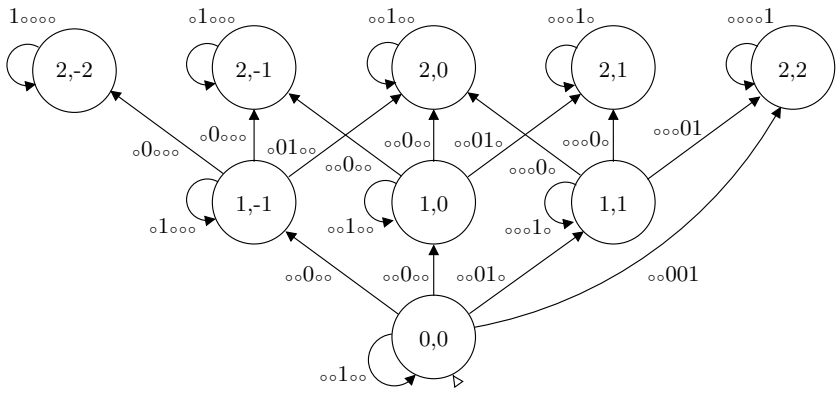


Fig. 4. NULA(2), the Nondeterministic Universal Levenshtein Automaton for $k = 2$ errors. All states on the same horizontal level carry the same number of errors x, and all states in same same column correspond to the same phase y in the pattern. Each looping arrow is an identity transition, each vertical arrow a substitution transition, each north-east arrow a deletion transition, and each north-west arrow an insertion transition. For the sake of readability, the symbol $\circ$ in a bit vector is either 0 or 1. $(0,0)$ is the start site and all states are accepting.

We prove that NULA(k) effectively recognizes the expected language. Recall that the *right language* of a state $q \in Q_k$, denoted $\mathcal{L}(q)$, is the set of all sequences of bit vectors u such that NULA(k) when started in q will accept u.

Proposition 6. *Let $(x, y) \in Q_k$. Given $P \in \Sigma^m$ and $V \in \Sigma^n$, we have*

- *when $y = 0$: $u \in \mathcal{L}(x, y)$ if, and only if, $Lev(P, V) \leq k - x$,*
- *when $y > 0$: $u \in \mathcal{L}(x, y)$ if, and only if, $Lev(P[1 + y..m], V) \leq k - x$,*
- *when $y < 0$: $u \in \mathcal{L}(x, y)$ if, and only if, $Lev(P, V[1 - y..n]) \leq k - x$,*

where u is the k-encoding of V with respect to P.

Proof. The proof is by induction on the length of u.

Corollary 7. *Let $P \in \Sigma^*$, $V \in \Sigma^*$. $Lev(P, V) \leq k$, if, and only if, the k-encoding of V with respect to P is accepted by* NULA(k).

2.4 Construction of the Deterministic Universal Levenshtein Automaton

We now explain how to build the Deterministic Universal Levenshtein Automaton from the NFA introduced in the preceding section. The principle is to use the standard powerset construction that converts a NFA into a DFA. In this construction, we show that it is possible to reduce the number of states by defining *subsumed states*. This notion is already present in [7], where it is defined for the Levenshtein automaton for a fixed word. Here, we adapt it to the states of NULA(k).

Definition 8. *Let (x, y) and (x', y') be two states of Q_k. We say that (x, y)* subsumes *(x', y'), denoted $(x', y') \sqsubset (x, y)$, if $x < x'$ and $y+x-x' \leq y' \leq y+x'-x$.*

It is clear from the definition that the relation $\sqsubset$ is a well-founded partial order.

Proposition 9. *Let $q \in Q_k$, $q' \in Q_k$, such that $q' \sqsubset q$. Then $\mathcal{L}(q') \subseteq \mathcal{L}(q)$.*

Proof. Consequence of Proposition 6. □

This proposition implies that all subsumed states can be removed from a subset of Q_k without modifying its right language. In other words, for any subset Q' of Q_k, Q' and $\textsc{Reduced}(Q')$ cannot be distinguished, where $\textsc{Reduced}(Q')$ is defined as the largest subset of Q' such that no two elements of $\textsc{Reduced}(Q')$ are subsumed. It allows us to prune the set of states considered during the construction of DULA(k). Figure 1 gives the corresponding algorithm and Definition 10 the formal definition of DULA(k).

Definition 10. *Let k be a positive number. The* Deterministic Universal Levenshtein Automaton *for k, denoted* DULA(k)*, is the DFA represented as follows.*

- *the input alphabet is $\{0, 1\}^{2k+1}$,*
- *the set of states is the set of reduced subsets of Q_k,*
- *the transition function δ is given by Algorithm of Figure 1,*
- *the start site is $\{(0, 0)\}$,*
- *all states are accepting.*

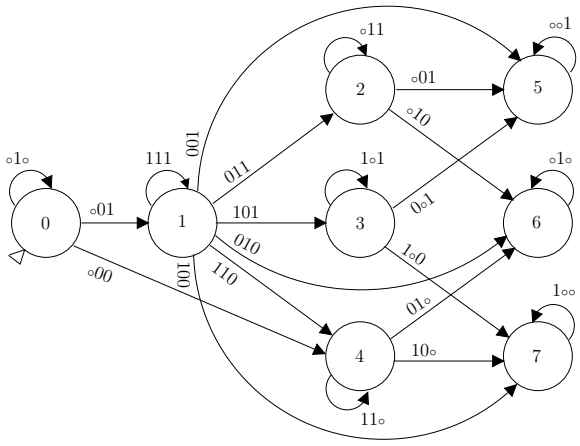
```
Add {(0,0)} to DULA(k) as an unmarked state;
while DULA(k) contains an unmarked state do
    Let T be that unmarked state;
    Mark T;
    for each bit vector u ∈ {0,1}^(2k+1) do
        S = {q' ∈ Q_k; ∃q ∈ T q' ∈ Δ(q,u)};
        S' = REDUCED(S);
        Define δ(T,u) = S';
        if S' is not in DULA(k) already then
            Add S' to DULA(k) as an unmarked state;
        end
    end
end
```

Algorithm 1: Construction of DULA(k) from NULA(k)

Figure 5 shows DULA(1), that has 8 states. DULA(2), the automaton obtained from NULA(2) visible on Figure 4, has 50 states, and is not represented here. For each value of k, the number of states of DULA(k) is computed with $R(k)$.

$$R_k(x,y) = \sum_{(x',y')\in Q_k, y<y', (x,y)\not\sqsubset(x',y'), (x',y')\not\sqsubset(x,y)} R_k(x',y')$$
$$R(k) = \sum_{(x,y)\in Q_k} R_k(x,y)$$

$R_k(x,y)$ is the number of reduced subsets of Q_k such that the state (x,y) is the element with the smallest y in the subset. The total number of reduced subsets for Q_k, $R(k)$ is obtained by summing over all possible states of Q_k. For k ranging from 1 to 10, R equals 8, 50, 322, 2187, 15510, 113633, 853466, 6536381, 50852018, 400763222. We conjecture that $R(k)$ is in $O(7^k)$.



0	no error	(0,0)
1	sub ins del	(1,-1) (1,0) (1,1)
2	sub del	(1,0) (1,1)
3	ins del	(1,-1) (1,1)
4	sub ins	(1,-1) (1,0)
5	del	(1,1)
6	sub	(1,0)
7	ins	(-1,1)

Fig. 5. DULA(1), Deterministic Universal Levenshtein Automaton for $k = 1$. It has 8 states, numbered from 0 to 7, that are all accepting. In the table, we report the subset of Q_1 corresponding to each state, as well as the semantics of the state.

3 Application to the Neighbourhood Counting Problem

We now turn to the problem of computing the cardinality of $\mathcal{L}\text{ev}(P,k)$. This value depends on the length of the word, the input alphabet and the internal structure of the word. Consider for example the three-letter alphabet {A,B,L}.

$\mathcal{L}\text{ev}(\text{AAA},1) = \{$ AAA, AA, AAB, AAL, ABA, ALA, BAA, LAA, AAAA, BAAA, LAAA, ABAA, ALAA, AABA, AALA, AAAB, AAAL$\}$

$\mathcal{L}\text{ev}(\text{LAB},1) = \{$ LAB, LA, AB, LB, AAB, BAB, LBB, LLB, LAA, LAL, ALAB, BLAB, LLAB, LAAB, LBAB, LALB, LABA, LABB, LABL$\}$

In the first case, the neighbourhood has 17 elements, and in the latter case 19 elements. The combinatorics is even more complex for greater values of k.

We show how to solve this problem efficiently with the help of DULA(k). Indeed, it is enough to intersect DULA(k) with the set of all sequences of bit vectors that are a valid encoding for some string V with respect to P. We designate by Encod(P,k) this latter language.

3.1 A DFA for Encod(P,k)

Definition 11. *Let P be a string of Σ^*. Encod(P,k) is the set $\{u \in (\{0,1\}^{2k+1})^*; \exists V \in \Sigma^*$ s.t. u is the k-encoding of V wrt to $P\}$.*

From Definition 2, we know that the elements of Encod(P,k) are strings of $k+m$ bit vectors.

Definition 12. *Let $V \in (\Sigma \cup \{\$\})^*$. Define $B(V) = \{\chi(s,V);\ s \in \Sigma\}$.*

$B(V)$ is the set of all bit vectors u of length $|V|$ that satisfies the three following properties.
- If $V_i = V_j$, then $u_i = u_j$.
- If $u_i = 1$ and $u_j = 1$, then $V_i = V_j$.
- If $V_i = \$$, then $u_i = 0$.

Example 13. B(ABL)=$\{000, 001, 010, 100\}$, B(ABB)=$\{000, 011, 100\}$, B(AA\$)= $\{000, 110\}$.

Encod(P,k) is recognized by the following DFA.
- the input alphabet is $(\{0,1\}^{2k+1})^*$,
- the set of states is $\{0,\ldots,m+k\} \cup \{\$_{m-k+1},\ldots,\$_{m+k}\}$,
- the transition function γ is defined by

$$\gamma(i-1,u)=i,\ 1 \le i \le m+k, u \in B(P'[i-k..i+k])$$
$$\gamma(i-1, 0^{k+1+m-i}1^{i+k-m})=\$_i,\ m-k+1 \le i \le m+k$$
$$\gamma(\$_{i-1}, 0^{k+1+m-i}1^{i+k-m})=\$_i,\ m-k+1 < i \le m+k$$

- the start state is 0,
- there are two accepting states: $m+k$ and $\$_{m+k}$.

Each state i recognizes the encodings of strings of Σ^+ of length i, and each state $\$_i$ recognizes the encodings of strings of $\Sigma^+\$^+$ of length i. The transition from $i-1$ to $\$_i$ corresponds to the first occurrence of \$ in the string. Figure 6 shows the DFAs obtained for AAA, LAB and $k=1$. We also give the DFA for BALLAD and $k=2$, for which several encodings were provided in Example 3.

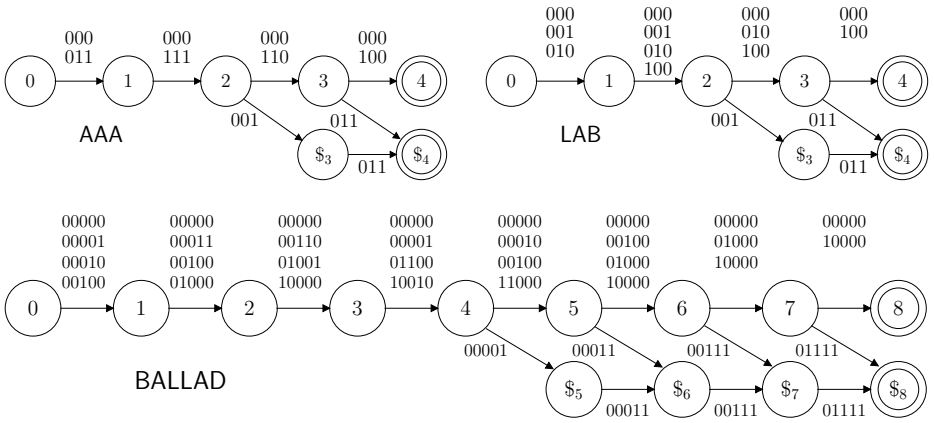


Fig. 6. DFAs for Encod(AAA, 1), Encod(LAB, 1) and Encod(BALLAD, 2)

3.2 Back to the counting problem

Considering that $\overline{\mathcal{L}\text{ev}}(P,k) = \text{DULA}(k) \cap \text{Encod}(P,k)$, we will exploit the product automaton of DULA(k) and Encod(P,k). This product automaton needs not to be constructed explicitely. It will serve us to design the recurrence formula to compute the size of $\mathcal{L}\text{ev}(P,k)$.

From the product automaton, we can deduce what is the size of the language recognized by $\overline{\mathcal{L}\text{ev}}(P,k)$. This is simply the total number of distinct paths leading from the start state to an accepting state. What we still have to do is to bring the problem back to the initial alphabet Σ. For that, we need a function $\alpha : \{0,1\}^{2k+1} \times \Sigma^{2k+1} \to \mathbb{N}$ that computes the number of symbols s of Σ such that $\chi(s,V) = u$, for each bit vector u and each word V over Σ.

$\alpha(u,V) = 1$, whenever at least one bit of u is 1
$\alpha(u,V) = |\Sigma| - |V|_\Sigma$ otherwise (in this case, $u = 00\cdots00$)

α is used to assign a multiplicity to each transition of the product automaton. In this context, the total number of underlying strings of Σ^* is the sum of all multiplicities of all distinct paths. Define S as follows.

$$
\begin{aligned}
S &: \textsc{Reduced}(Q_k) \times (\{0,\ldots,m+k\} \cup \{\$_{m-k+1},\ldots,\$_{m+k}\}) \to \mathbb{N} \\
S(0,0) &= 1 \\
S(q',i+1) &= \textstyle\sum_{u,q,q'=\delta_k(q,u)} \alpha(u, P'[i-k..i+k]) \times S(q,i), \quad 0 \le i < m+k \\
S(q',\$_{m-k}) &= \textstyle\sum_{u,q,q'=\delta_k(q,u)} S(q, m-k-1) \\
S(q',\$_{i+1}) &= \textstyle\sum_{u,q,q'=\delta_k(q,u)} S(q,i-1) + S(q,\$_i), \quad m-k \le i < m+k
\end{aligned}
$$

m is the length of P. $S(q,i)$ is the number of distinct paths leading from the start site to the state (q,i) in the product automaton. The final result is obtained by summing over the accepting states.

Proposition 14.

$$|\mathcal{L}ev(P,k)| = \sum_{q \in \text{REDUCED}(Q_k)} S(q, m+k) + S(q, \$_{m+k})$$

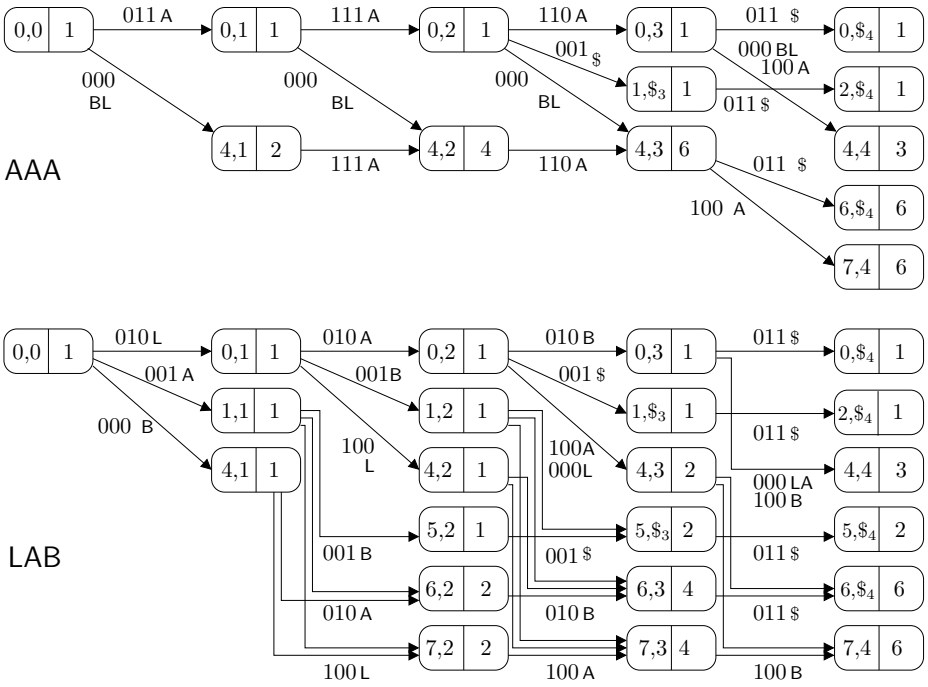


Fig. 7. Computation of S for AAA and LAB, $\Sigma = \{\mathsf{A}, \mathsf{B}, \mathsf{L}\}$ and $k = 1$. For each state (q, i) or $(q, \$_i)$ (left part of the box), we indicate the value of S (right part of the box). The labels on the transitions are the bit vectors. For each bit vector, we also mention the corresponding symbol(s) of Σ. The number of such symbols is α. The result is the sum of S over the last column.

Figure 7 shows the developments of S for $\mathcal{L}\text{ev}(\mathsf{AAA}, 1)$ and $\mathcal{L}\text{ev}(\mathsf{LAB}, 1)$, which were obtained from DULA(1) in Figure 5 on the one hand, and Encod(AAA,1), Encod(LAB,1) in Figure 6 on the other hand. S can be implemented by dynamic programming with a table of size $R(k) \times (m + 3k)$. Each element of the table is computed in constant time. So the algorithm has a time complexity of $O(m)$. In practice, for each possible word structure (e.g. AAA, AAB, ABL), the associated transitions in DULA(k) can be extracted during a preprocessing step. As for the space complexity, this is not necessary to store the full dynamic table. At any instant, the algorithm only requires elements from the current row (i) and the previous row ($i - 1$). Thus the space complexity is in $O(1)$.

Remark 15. In Section 2, we have put a lot of efforts into defining a universal automaton that is able to process any pair of strings. In this Section, we have specialized this automaton for a fixed pattern P and have obtained a DFA for $\mathcal{L}\text{ev}(P, k)$. Alternatively, we could have directly used the DFA presented in [7]. The construction of this DFA is also in linear time, for a fixed value of k. However it requires a complex table-based preprocessing, that is dependent on each pattern. In our approach, the computational burden is reported to the design of DULA(k), which is performed only once. This is a new route to recover the result established in [7].

4 Conclusion

We have shown how to count the number of strings present in the neighbourhood of some fixed reference word P. The algorithm produces a product automaton, which could also be used to generate the set of all strings in the neighbourhood of P, or to sample it. This generic approach could extend to other target regular languages, instead of the singleton language $\{P\}$. The downside of the method, however, is that it requires the computation of the DFA DULA(k), whose size increases exponentially with the number of errors k. It exceeds one million states with $k = 8$. Another route is to construct the product automaton with NULA(k), instead of DULA(k), and then determinize the resulting automaton with an optimization similar to Proposition 9. This could lead to a lower memory consumption for regular languages that do not need all transitions of DULA(k).

References

1. Baeza-yates, R., Navarro, G.: A faster algorithm for approximate string matching. Algorithmica pp. 1–23 (1996)
2. Becerra-Bonache, L., de la Higuera, C., Janodet, J.C., Tantini, F.: Learning balls of strings from edit corrections. J. Mach. Learn. Res. 9, 1841–1870 (2008)
3. Holub, J., Melichar, B.: Implementation of nondeterministic finite automata for approximate pattern matching. In: Automata Implementation. Lecture Notes in Computer Science, vol. 1660, pp. 92–99. Springer Berlin Heidelberg (1999)
4. Mihov, S., Schulz, K.: Fast approximate search in large dictionaries. Computational Linguistics 30(4), 451–477 (2004)
5. Mitankin, P.: Universal Levenshtein automata. Building and properties. Master's thesis, University of Sofia (2005)
6. Myers, G.: What's behind blast. In: Chauve, C., El-Mabrouk, N., Tannier, E. (eds.) Models and Algorithms for Genome Evolution, Computational Biology, vol. 19, pp. 3–15. Springer London (2013)
7. Schulz, K., Mihov, S.: Fast string correction with levenshtein automata. International Journal of Document Analysis and Recognition 5, 67–85 (2002)
8. Ukkonen, E.: Finding approximate patterns in strings. Journal of Algorithms 6(1), 132–137 (1985)
9. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM 21(1), 168–173 (1974)