A Probabilistic Algorithm for Updating Files over a Communication Link

Alexandre V. Evfimievski*

Abstract

Consider two persons, **P** and **Q**, connected by a bidirectional communication link. Let **P** have a binary string x and let **Q** have y. The string y can be obtained from x by a small number k of editing operations: inserting/deleting a bit and copying/moving/deleting a block (substring). Our problem is to communicate the string y to **P** using a small number of bits in both directions. We present a probabilistic algorithm for that task and prove that for any x, y and $\varepsilon > 0$ the algorithm communicates y (a string that **Q** has) to **P** with probability $1-\varepsilon$ using $poly(\log|x|, \log|y|, \log\varepsilon^{-1}, k)$ bits. The running time is $poly(|x|, |y|, \log\varepsilon^{-1})$.

1 Introduction

Consider two persons, **P** and **Q**; assume that **P** knows a binary string x and **Q** knows a binary string y (that is assumed to be close to x, see below). The persons can send bits to each other (Figure 1).

P wants to know the string y; the communication protocol should require as few bits as possible using the fact that **P** already knows the string x that y is close to. The distance between y and x is measured by the number of edit operations needed to transform x into y (see Definition 2.1); let us mention that distance in our sense is not symmetric. We present a probabilistic algorithm and estimate the number of transmitted bits and running time.

The key tool is a probabilistic communication algorithm that compares two strings using a logarithmic number of transmitted bits. This algorithm was invented by Rabin and Yao [4]. However, we need some additional construction to use this tool for our problem.

The edit distances that use block operations were considered by Hannenhalli and Pevzner [1] who constructed (non-communicational) polynomial algorithms for computing these distances.

The problem discussed in this paper was communicated to me by A. Shen. His and N. Vereshchagin's help was very useful.

*Department of Mathematical Logic and Theory of Algorithms, Moscow State University, Russia

2 The main theorem

Definition 2.1. *A binary string y is called k–similar to a binary string x if x can be transformed to y using at most k operations of the following types:*

- *bit insertion:* $AB \to A0B$ *or* $A1B$;
- *bit deletion:* $A0B \to AB$ *or* $A1B \to AB$;
- *block transposition:* $AXBC \to ABXC$;
- *block deletion:* $AXB \to AB$;
- *block duplication (copying):* $AXBC \to AXBXC$.

Theorem 2.1. *There exist probabilistic algorithms for* **P** *and* **Q** *such that for any strings x, y and for any $\varepsilon > 0$ the algorithms transmit the string y from* **Q** *to* **P** *with error probability ε, and if y is k–similar to x then the number of communicated bits is $poly(k, \log|x|, \log|y|, \log\varepsilon^{-1})$ and running time is $poly(k, |x|, |y|, \log\varepsilon^{-1})$.*

More exactly, the upper bound for the communication complexity (see (4.2)) is following:

$$1000k^2 \log|y| \cdot (16 + \log|x| + 5\log|y| + \log\varepsilon^{-1}).$$

3 Construction of the algorithm

Without loss of generality we may assume that the length of y is a power of two: $|y| = 2^r$, $r \in \mathbf{N}$ (If this is not the case, **Q** adds a necessary number of zeros to y, informing **P** about that number). First, **Q** sends the length of y to **P**, and **P** allocates memory where y will be formed. Second, one of the persons (say, **Q**) chooses and sends to another one a prime number p in the interval

$$(3.1) \qquad 501 \cdot |x||y|^5\varepsilon^{-1} \le p \le 100 \cdot 501 \cdot |x||y|^5\varepsilon^{-1}.$$

It can be easily done in polynomial time by a probabilistic algorithm for testing primality (see [2, 3]) due to high density of the distribution of prime numbers.

The main procedure in the algorithm is "an attempt to transmit a substring z of y from **Q** to **P**", called

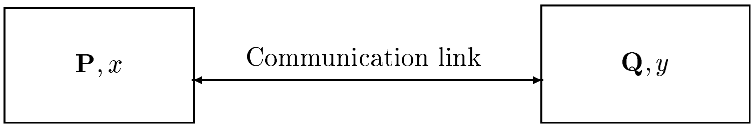


Figure 1: Before the transmission of y to $\mathbf{P}$.

try(z). We assume that before executing try(z) both $\mathbf{P}$ and $\mathbf{Q}$ know the length of z, its position in y and the prefix of y that precedes z (the $\mathbf{P}$'s idea about this prefix may be wrong, but this happens with a small probability).

To perform try(z) the person $\mathbf{Q}$ first chooses at random an element α of the field $\mathbf{Z}/p\mathbf{Z}$ and computes over this field the "fingerprint value"

$$\beta = z_0 + z_1\alpha + z_2\alpha^2 + \ldots + z_{|z|-1}\alpha^{|z|-1} = \\ = \sum_{j=0}^{|z|-1} z_j\alpha^j, \quad \text{where } z = \overline{z_0 z_1 \ldots z_{|z|-1}}.$$

Then, $\mathbf{Q}$ sends to $\mathbf{P}$ the numbers α and β. The person $\mathbf{P}$ searches through all substrings of length $|z|$ in the string x and in the prefix of y that has been transmitted earlier, and computes the fingerprint value for all of them. If for all tested substrings the fingerprint value differs from β then $\mathbf{P}$ informs $\mathbf{Q}$ that the transmission failed. If $\mathbf{P}$ has found a substring with the same fingerprint, it copies this substring into the place for z (i.e. copies the substring found adding it to the transmitted part of y) and informs $\mathbf{Q}$ that the transmission was successful (Figure 2).

If the transmission fails, we repeat it recursively with shorter strings (two halves of z). So we get the following procedure send(z) that transmits a substring from $\mathbf{Q}$ to $\mathbf{P}$:

```
procedure send(z);
begin
   if |z| = 1 then transmit z as a bit;
   else begin
      try(z); if failure then begin
         send(the left half of z);
         send(the right half of z);
      end;
   end;
end.
```

Our algorithm maintains (with high probability) the following invariant relation: when transmitting some substring z of y we have already transmitted the prefix of y that precedes z. The transmission of the whole string y corresponds to the command send(y).

So the algorithm is straightforward; however, the proof of upper bounds requires some technical lemmas. Now let us put the lemmas aside and turn to the proof of the theorem.

4 Proof of the theorem

In order to prove the theorem, we must find an upper bound for the number of bits transmitted through the communication link; the restriction for the probability of error must also be taken into account. First we estimate the number of operations try(z) performed by the algorithm.

According to Lemma 5.2 (Section 5), the string y can be divided into $M \leq 112k^2$ blocks so that every block coincides either with a block in x or with a block in y which is on the left from it. Thus, if the operation try(z) is executed with a substring z which is a part of such a block, then it will be successful (with the probability 1), though not necessarily correct. Consequently, the operation send(z) can make recursive calls only when z crosses a boundary between these blocks.

What substrings z will arise when the algorithm works? Only the substrings whose length is a power of 2 and the position of the first bit is a multiple of the length. If $|y| = 2^n$ then for every $i = 1, \ldots, n$ no more than M such substrings of length 2^i can cross a boundary between blocks. It means that the total number of substrings z where send(z) makes recursive calls is at most $M \log |y|$. Every such operation send(z) starts with calling try(z) and then makes two recursive calls. Hence, the total number $N_{\mathtt{try}}$ of calls of try(z) can be estimated as follows:

$$N_{\mathtt{try}} \leq 3M \log |y| \leq 336k^2 \log |y|.$$

We will not consider the case $|z| = 1$ separately: it is clear that to send a bit is extremely easy.

Now let us consider the work of try(z). During its execution $\mathbf{Q}$ transmits to $\mathbf{P}$ the numbers α and β, which have the length $\lceil \log p \rceil$; $\mathbf{P}$ transmits only one bit. During the execution of the whole algorithm the number of transmitted bits is at most

$$N_{bits} \leq \log |y| + \lceil \log p \rceil + N_{\mathtt{try}} \cdot (2\lceil \log p \rceil + 1) \leq \\ \leq 1000k^2 \log |y| \cdot \lceil \log p \rceil.$$

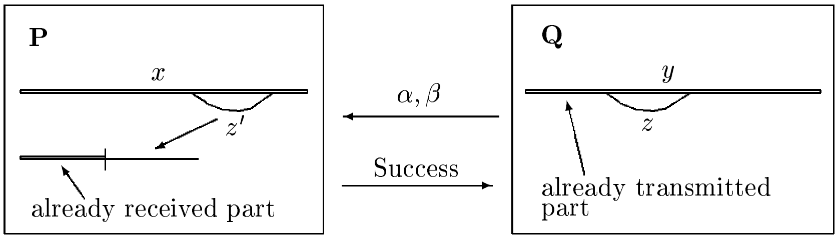


Figure 2: Performing of the procedure $\mathtt{try}(z)$.

Using the upper bound for p (from (3.1)) we obtain:

$$\begin{aligned} & p \le 50100|x||y|^5\varepsilon^{-1} \quad \Rightarrow \\ (4.2) \qquad & \Rightarrow \quad N_{bits} \le 1000k^2 \log|y| \cdot \lceil \log p \rceil \le \\ & \le 1000k^2 \log|y| \cdot (16 + \log|x| + 5\log|y| + \\ & + \log\varepsilon^{-1}) \le poly(k, \log|x|, \log|y|, \log\varepsilon^{-1}); \end{aligned}$$

polynomial upper bound for the running time is obvious.

It remains to estimate the probability of error. In order to determine whether a substring z coincides with a substring z', the algorithm calculates their fingerprints, i.e., polynomials like

$$z_0 + z_1\alpha + z_2\alpha^2 + \ldots + z_{|z|-1}\alpha^{|z|-1}$$

at a random point $\alpha \in \mathbf{Z}/p\mathbf{Z}$, and compares their values. If $z \ne z'$ then the algorithm will make an error only in the case when α is a root of the polynomial

$$(4.3) \qquad \begin{aligned} & (z_0 - z'_0) + (z_1 - z'_1)\alpha + (z_2 - z'_2)\alpha^2 + \ldots + \\ & + (z_{|z|-1} - z'_{|z|-1})\alpha^{|z|-1}. \end{aligned}$$

The polynomial is nonzero and its degree is less than $|z|$; so the probability of error is at most $\frac{|z|}{p}$. During the execution of one operation $\mathtt{try}(z)$ the algorithm makes at most $|x| + |y|$ comparisons, and the number of operations is at most $N_{\mathtt{try}}$, hence the probability of error is bounded by

$$N_{\mathtt{try}} \cdot (|x| + |y|) \cdot |y| p^{-1} \le 500k^2|x||y|^3p^{-1}.$$

Taking into account that $k \le |y|$ and $p > 500 \cdot |x||y|^5\varepsilon^{-1}$ (see (3.1)), we obtain the following upper bound:

$$500k^2|x||y|^3p^{-1} < 500|x||y|^5 \cdot (500|x||y|^5\varepsilon^{-1})^{-1} = \varepsilon.$$

5 Technical Lemmas

DEFINITION 5.1. *By a copying–attaching–operation (CA–operation) we denote a transformation of a binary string ABC to ABCB (for any strings A, B and C). In the obtained string ABCB we call the first block B as the source block (S–block) and the last block B as the copied block (C–block).*

LEMMA 5.1. *If a string y is k–similar to a string x then there exists a sequence of no more than $4k$ CA–operations that transforms x to a string xuy (for some string u). Every bit position in the string xuy can be a part of an S–block for at most 2 CA–operations of this sequence.*

Proof. All the block operations from the definition of k–similarity can be translated to the language of copying–attaching (using no more then 4 CA–operations instead of every operation) as follows:

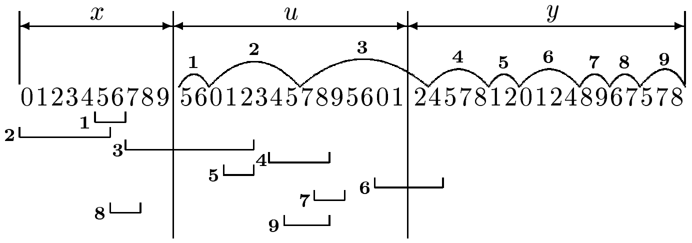
Deletion:	Transposition:	Copying:
ABC	ABCD	ABCD
ABC A	ABCD A	ABCD ABC
ABC AC	ABCD AC	ABCD ABCB
	ABCD ACB	ABCD ABCBD
	ABCD ACBD	

Operations with bits can be considered as operations with blocks (we assume for simplicity that all strings considered contain both ones and zeros).

The resulting string xuy is a concatenation of substrings, each of them is 1–similar to the previous one. Every bit position in xuy belongs to one of these substrings and can be in an S–block only for CA–operations forming the next substring, i.e., only twice.

LEMMA 5.2. *If the string y is k–similar to the string x then there exists a sequence of at most $112 \cdot k^2$ CA–operations that transforms x to xy.*

Proof. According to Lemma 5.1 there exists some intermediate string u, such that x can be transformed to $w = xuy$ by means of $m \le 4k$ CA–operations. Every CA–operation has its S–block $[p, q]$ and its C–block $[p', q']$ (a block $[p, q]$ begins at a bit position p and ends at a position q in the string w); in this case $p \le q < p' \le q'$, $q - p = q' - p'$ and both substrings $\overline{w_p w_{p+1} \ldots w_q}$ and $\overline{w_{p'} w_{p'+1} \ldots w_{q'}}$ are the same. Since the part uy is a result of CA–operations, it is a sequence of m concatenated C–blocks (Figure 3).



CA–operations:
(in the case when digits are bits)
1. [5,6] $\rightarrow$ [10,11];
2. [0,5] $\rightarrow$ [12,17];
3. [7,14] $\rightarrow$ [18,25];
4. [16,19] $\rightarrow$ [26,29];
5. [13,14] $\rightarrow$ [30,31];
6. [23,26] $\rightarrow$ [32,35];
7. [19,20] $\rightarrow$ [36,37];
8. [6,7] $\rightarrow$ [38,39];
9. [17,19] $\rightarrow$ [40,42].

Figure 3: A sample string xuy obtained from x by means of a sequence of CA–operations. Here digits in the string denote some binary substrings (identical digits correspond to identical substrings); S–blocks are shown below the string, C–blocks are shown above it.

Our task is to eliminate u by changing the sequence of CA–operations so that new sequence will transform x to xy. We will do it step–by–step. At every step we delete the substring that is the rightmost C–block in u; other C–blocks in u (and corresponding CA–operations) remain the same, but some CA–operations forming y are changed.

First, look whether the boundary between u and y crosses some C–block. If yes, we replace the corresponding CA–operation by two CA–operations so that the boundary between u and y will be the boundary between new C–blocks.

STEP of the transformation:

Before the step we have the rightmost C–block $[p', q']$ in u intersected with some S–blocks $[p_j, q_j]$; The corresponding C–blocks are situated in the string y (Figure 4).

Substep 1: Every S–block intersecting with the rightmost C–block $[p', q']$ is divided into parts; the division points are left and right boundaries of $[p', q']$ and all S–block boundaries situated inside $[p', q']$. The corresponding C–block (situated in y) is divided into the same parts, and one CA–operation is replaced by several consecutive CA–operations (Figure 5).

Substep 2: Now every two S–blocks inside $[p', q']$ are disjoint or coincide. For every group of coinciding S–blocks we take one of them, such that the corresponding C–block is the leftmost one. Then we change CA–operations corresponding to other S–blocks of the group by using this leftmost C–block as S–block for all these CA–operations (Figure 6).

Substep 3: All S–blocks remaining inside $[p', q']$ are disjoint. The C–block $[p', q']$ corresponds to the S–block $[p, q]$ which is situated on the left of $[p', q']$ (somewhere in u or in x). Since the blocks $[p, q]$ and $[p', q']$ contain identical substrings, we can use $[p, q]$ as a place for S–blocks from $[p', q']$. After this change no S–blocks remains inside $[p', q']$, therefore, this piece of u becomes useless. We delete this substring and exclude the CA-operation "$[p, q] \rightarrow [p', q']$" from the sequence of CA–operations (Figure 7).

After one step of the transformation the number of C–blocks inside u decreases by 1, hence at most m steps are required to eliminate the whole u.

Now it remains to estimate the number of blocks after the transformation. By s we denote the number of points in the string u (boundaries between consecutive bits), each of which contains at least one, left or right, boundary of some S–block (if there are several boundaries in the same point, we count it only once). By t we denote the maximal number of S–blocks containing the same bit in the string u. We can see that t does not increase during the transformation. Indeed, block division does not change t; when we set some S–blocks out of u, t can only decrease; when we set some disjoint S–blocks to $[p, q]$ instead of $[p', q']$, t may increase by 1, but we immediately compensate it by the elimination of the CA–operation "$[p, q] \rightarrow [p', q']$".

The value of s can increase, but no more than by 1 at every step of the transformation (except for the first step, when it can increase by 2). At Substep 1 only two boundaries of the C–block $[p', q']$ can be added to the set of "boundary points" and increase s by at most 2; other new boundaries appear in the points containing boundaries earlier. Moreover, if this is not the first step then the right boundary of $[p', q']$ is in the same point as the left boundary of the C–block deleted during the previous step; therefore, we have already counted this point. At Substep 2 s does not change; Substep 3 also can not increase s because, first, every new boundary point inside $[p, q]$ corresponds to some deleted boundary point inside $[p', q']$, and second, two possible new boundary points moved from the left and

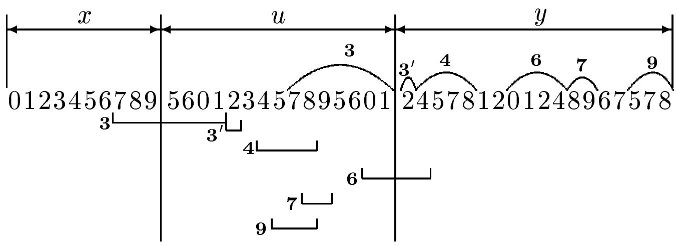


Figure 4: The sample string from Figure 3 before the Step. CA–operation 3 has been divided; blocks that will not change during the Step are not shown.

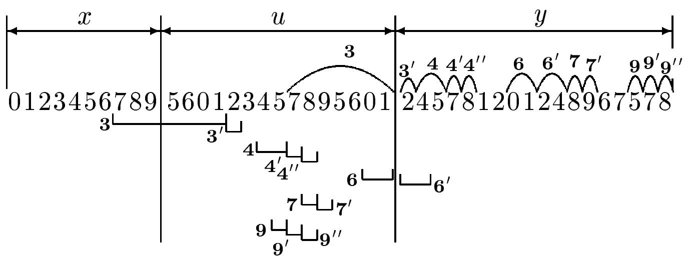


Figure 5: The sample string after Substep 1.

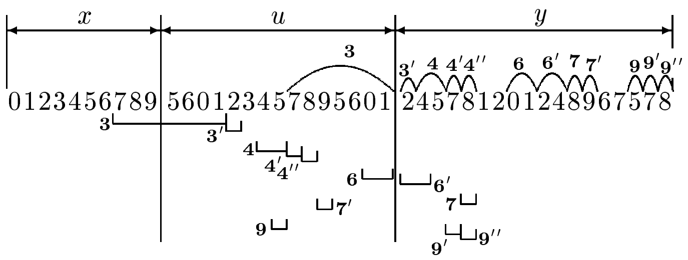


Figure 6: The sample string after Substep 2.

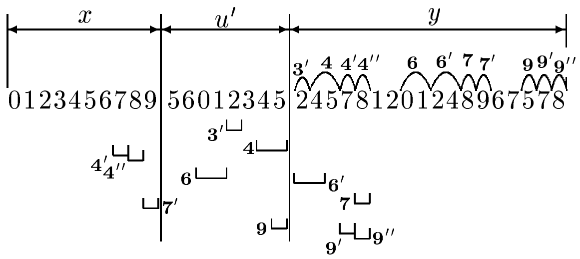


Figure 7: The sample string after the whole Step.

right boundaries of $[p', q']$ appear in points containing boundaries earlier.

Before the beginning of the transformation t was at most 2 (see Lemma 5.1) and s was at most $2m$. Since the number of steps is at most m, after every step the following will be true:

$$s \leq 3m + 1; \qquad t \leq 2.$$

At every step after Substep 1 at most $s \cdot t$ S–blocks will be inside the C–block $[p', q']$. Only these blocks can be added to the whole set of S–blocks at this step, hence, this increasing will be at most $6m + 2$.

After the whole process of the transformation the number of CA–operations M can be estimated as

$$M \leq m + m \cdot st \leq 3m + 6m^2 \leq 7m^2 \leq 112k^2 \qquad (m > 3).$$

References

[1] Sridhar Hannenhalli, Pavel Pevzner, *Transforming men into mice (polynomial algorithm for genomic distance problem)*. Tech. Report CSE–95–012, Dep. of Computer Science and Engineering, Penn. State Univ., May 1995

[2] M.O. Rabin, *Probabilistic algorithms for testing primality*. J. Number Theory, 12(1980), p. 128–138

[3] R. Solovay, V. Strassen, *A fast Monte–Carlo test for primality*. SIAM J. Comput., 6(1977), p. 84–85; Erratum 7(1978), p. 118

[4] A.C. Yao, *Some complexity questions related to distributed computing*. Proc. 11th Ann. ACM Symp. on Theory of Computing (1979), p. 209–213