RAIRO-Theor. Inf. Appl. 45 (2011) 235–248
DOI: 10.1051/ita/2011017

Available online at:
www.rairo-ita.org

NORMALIZATION OF EDIT SEQUENCES FOR TEXT SYNCHRONIZATION

Rafael C. Carrasco[1] and Alexander Sánchez Díaz[2]

Abstract. It often occurs that local copies of a text are modified by users but that the local modifications are not synchronized (thus allowing the merged text to become the source for later editions) until later when, for instance the network connection is reestablished. Since text editions usually affect a small fraction of the whole content, the history of edit operations provides a compact representation of the modified file. In this paper, we define a normal form for these records which will permit for the comparison of all text files that have been obtained by editing a common source S when the difference between each output file O_i and the source file is given as a sequence L_i of edit operations. We show that the normalized sequence is unique for all the equivalent text editions and provide efficient procedures with which to compute this normal form and to obtain the edit sequence L_M transforming S into a merged file M which integrates all the local modifications. We also discuss how these normalization can be integrated into the operational transformation paradigm for optimistic replication.

Mathematics Subject Classification. 68U99.

1. Introduction

The replication of data enables independent access to shared information, such as that contained in wiki pages or distributed databases by maintaining multiple copies of the data in separate computers. Optimistic approaches to data replication provide a higher availability of the resources because they allow the replicas to be updated independently. In these frameworks, the consistency of the data

Keywords and phrases. Edit distance, text synchronization, reconciliation of replicas.

[1] Departamento de Lenguajes y Sistemas Informàticos, Universidad de Alicante, 03071 Alicante, Spain. carrasco@dlsi.ua.es

[2] Departamento de Computación. Universidad Agraria de La Habana. La Habana, Cuba. alezsd@yahoo.com

Article published by EDP Sciences

must be attained later, during the reconciliation process, and the convergence to an identical state is only guaranteed at quiescence. A complete review of the optimistic replication techniques has been published by Saito and Shapiro [18].

Depending on the nature of the information that is propagated, there are two main families of approaches for the updates performed in optimistic replication: state transfer, in which the current object is propagated, and operation transfer [5], in which only the modifications regarding a previous state are shared. There are several fast methods with which to compare content – see, for example, [13,23] – and there is also a standard procedure with which to compare files that come from an original common source [9].

Delta compression methods [20] can be used in the state transfer approach to reduce the amount of information transmitted over the network, since they encode the differences between a source file and a target file. For example, the *rsync* algorithm [24] seeks the minimal number of blocks that must be transmitted in order to update a replica according to the primary copy. However, when no file acts as primary for the distributed data, a mechanism must be defined which decides what modifications must be preserved in the synchronization. Non-conflicting operations are usually propagated automatically [1] but whenever unsafe synchronizations are detected a conflict is reported. For example, the algorithm *Unison* reports a conflicts to the user when a file has been modified on both sources [15].

However, network bandwidth is optimized if conflicts are minimized. The use of vector time pairs [14] to track file modifications and synchronization history together has been implemented in the *Tra* synchronizer [4] in order to avoid false conflicts and to allow for partial synchronizations. An algebraic framework which provides a proof system for reasoning about operations performed on a file system also exists [16].

The log-based model for the reconciliation of replicas denominated as IceCube [8] exploits dynamic constraints in order to merge logs of actions performed on a common initial state. These constraints prune the space of reorderings that must be explored in order to minimize conflicts and the model can also be applied to binary objects. In this approach, non-commutative operations appear in a canonical order while commutative operations are ordered arbitrarily but consistently.

In the case of collaborative editors, operation-transfer methods permit a more flexible resolution of conflicts during replication and, in particular, the definition of strategies with which to preserve the users' intention [21,22]. For example, *operation-based merging* [11] defines a precedence relation on the primitive transformations and partitions the primitive transformations performed into non-conflicting blocks, so that there cannot be any cycles among blocks; it then detects conflicts based on prior knowledge concerning what operations commute (both locally or globally).

In this paper, we address the generation of modification logs containing edit operations performed on each text and its application to detect conflicts and to update replicas. Edit sequences, rather than single edit operations, will be propagated when frequent or persistent partitions in the network occur. We then

define a normalized form for the sequences of edit operations together with certain algorithms with which to obtain and maintain it. We shall prove that this normalization is unique and, that it thus allows equivalent sequences to be identified by means of a simple comparison. Furthermore, the amount of information that must be propagated is reduced because cancelling operations are automatically removed.

The normalized edit sequences are introduced in Section 2 and a procedure by which compute them from log files is described in Section 3. Some basic methods with which to modify and update edit sequences are described in Section 4. In Section 5, the results are summarized and ongoing research topics are presented.

2. Basic concepts

We shall consider *insertions* and *deletions* to be the basic edit operations on text files. In the following, edit operations will be denoted with pairs in $E = \mathbb{N} \times \Sigma_{\#}$, where $\Sigma_{\#} = \Sigma \cup \{\#\}$ is the alphabet of characters permitted in the text, $\Sigma = \{a, b, c, \ldots\}$, extended with the special symbol $\#$ to represent deletions. A pair $(k, \gamma) \in E$ will be interpreted as the insertion of character γ after position k in the text if $\gamma \in \Sigma$ and as the deletion of the character at position $k+1$ if $\gamma = \#$. More precisely, the result $S \cdot (k, \gamma)$ obtained after the application of the edit operation (k, γ) to a source text $S = s_1 s_2 \cdots s_L$ is

$$S \cdot (k, \gamma) = \begin{cases} s_1 \cdots s_k \gamma s_{k+1} \cdots s_L & \text{if } \gamma \in \Sigma \text{ and } k \leq L \\ s_1 \cdots s_k s_{k+2} \cdots s_L & \text{if } \gamma = \# \text{ and } k < L \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{2.1}$$

An *edit sequence* is a finite sequence of edit operations. More specifically, the empty edit sequence will be denoted with ϵ and the output when it operates on text S is $S \cdot \epsilon = S$. The output when the edit sequence $X = x_1 \cdots x_N$, with $N > 1$, operates on text S is $S \cdot X = (S \cdot x_1) \cdot (x_2 \cdots x_N)$. For example, the output when $(2, a)(0, \#)(1, c)$ operates on *abcd* is *bcacd*.

Two edit sequences X and Y will be said to be *equivalent*, $X \equiv Y$, if they generate identical output $S \cdot X = S \cdot Y$ when they operate on any source text S. In order to compare edit sequences, it is convenient to define a normalized form. For this purpose, we will write $(i, \alpha) \prec (j, \beta)$ if either $i < j$ or $\alpha = \beta = \#$ with $i = j$, and state that the edit sequence $x_1 x_2 \cdots x_N$ is *normalized* if $x_{n-1} \prec x_n$ for all $1 < n \leq N$. In particular, the empty sequence ϵ, which has length $N = 0$, is normalized.

A key feature of normalized edit sequences is that they are unique because two edit operations in a normalized sequence cannot access the same position in the text – unless they belong to a run of identical deletions $(k, \#)$ encoding a block delete. Indeed, let $X = \epsilon$ and S be an empty text. The result $S \cdot Y$ will then be undefined – and, thus, different from $S \cdot X = S$ – unless $Y = \epsilon$ or Y starts with an insertion (k, σ) with $k = 0$ and $\sigma \in \Sigma$. However, in the last case, later edit operations in Y can only remove those characters that appear after the first

character in the text and, thus, $S \cdot Y \neq S$. Therefore, Y must be identical to $X = \epsilon$. Consequently, $Y \equiv X = x_1 x_2 \cdots x_M$ with $M > 0$ implies $Y \neq \epsilon$. Furthermore, the initial operation y_1 in $Y = y_1 y_2 \cdots y_N$ must be identical to x_1 because, first, an insertion (i, σ) or a run of identical deletions $(i, \#)$ modifies the content of the text at position $i + 1$ and this content (and the preceding content) cannot be changed later by operations of the type (j, γ) with $j > i$; and, second, a normalized edit sequence contains at most one type of operation – a single insertion (i, σ) or a run of deletions $(i, \#)$ – operating on position i. By induction, the remaining operations in X and Y must be also identical.

In the following section, sequences will be obtained as an intermediate result, but will not fully normalized since they may contain pairs of the type $(i, \sigma)(i, \#)$. In this case, $x_1, x_2, \cdots x_N$ will be called a *quasi-normalized* edit sequence where $x_{n-1} \precsim x_n$ for all $1 < n \leq N$ and $(i, \alpha) \precsim (j, \beta)$ denotes $i < j$ or $\alpha = \#$ with $i = j$.

3. Normalization of edit sequences

As will be shown below, for every edit sequence X it is always possible to obtain a normalized edit sequence Y, such that $Y \equiv X$ through the application of a set of transpositions. However, as edit operations are not commutative, the procedure requires a careful analysis. For instance, if $i \geq j$ we can write for all $\gamma \in \Sigma_{\#}$ and $\sigma \in \Sigma$:

$$(i, \gamma)(j, \sigma) \equiv (j, \sigma)(i + 1, \gamma). \tag{3.1}$$

This equivalence allows any unnormalized edit sequence of the type $(i, \gamma)(j, \sigma)$ to be transformed into a new one which is normalized, because $i \geq j$ implies $(j, \sigma) \prec (i + 1, \gamma)$. In contrast, for pairs of the type $(i, \gamma)(j, \#)$ with $i > j$ and $\gamma \in \Sigma_{\#}$ it holds that

$$(i, \gamma)(j, \#) \equiv (j, \#)(i - 1, \gamma). \tag{3.2}$$

This equivalence leads either to a normalized edit sequence, if $(j, \#) \prec (i - 1, \gamma)$, or only to a quasi-normalized edit sequence if $i = j + 1$ and $\gamma \in \Sigma$. However, according to equation (3.1), $(j, \#)(j, \sigma)$ is equivalent to $(j, \sigma)(j + 1, \#)$ which satisfies $(j, \sigma) \prec (j + 1, \#)$, and the consecutive application of both equivalences thus leads to a normalized sequence[1]. Finally, for $i = j$ and $\sigma \in \Sigma$ we have

$$(i, \sigma)(j, \#) \equiv \epsilon, \tag{3.3}$$

signifying that the removal of an inserted character is equivalent to the (normalized) empty sequence.

The three equivalence relations shown above are used by Algorithm 1 to transform any unnormalized edit sequence $x_1 \cdots x_N$ into an equivalent quasi-normalized edit sequence. In this algorithm, which can be considered as a variation of the standard insertion sort, the empty sequences generated as a consequence of the

[1]We will not use directly the equivalence $(j+1, \sigma)(j, \#) \equiv (j, \sigma)(j+1, \#)$ in the sorting algorithms below because it could generate unnormalized sequences when the neighboring operations in the sequence are considered.

Algorithm 1 `quasiNormalize`(X)

Input: An edit sequence $X = x_1 \cdots x_N$
Output: A quasi-normalized edit sequence Y such that $X \equiv Y$.

for $n = 1, \ldots, N$ **do**
  $m \leftarrow n$
  while $m > 1 \wedge x_m \neq null \wedge (x_{m-1} = null \vee x_{m-1} \not\prec x_m)$ **do**
    $(i, \alpha) \leftarrow x_{m-1}$
    if $x_{m-1} = null$ **then**
      $x_{m-1} \leftarrow x_m$
      $x_m \leftarrow null$ {Skip over *null*}
    else if x_m is an insertion **then**
      $x_{m-1} \leftarrow x_m$
      $x_m \leftarrow (i + 1, \alpha)$ {Apply eq. (3.1)}
    else if $x_{m-1}x_m \equiv \epsilon$ **then**
      $x_{m-1} \leftarrow null$
      $x_m \leftarrow null$ {Apply eq. (3.3)}
    else
      $x_{m-1} \leftarrow x_m$
      $x_m \leftarrow (i - 1, \alpha)$ {Apply eq. (3.2)}
    end if
    $m \leftarrow m - 1$
  end while
end for
$Y \leftarrow \text{clean}(X)$ {Remove all identity operations (nulls) in X}
return Y

application of the equivalence (3.3) are replaced by a distinguished *identity* operation – represented as a *null* reference in the implementation.

It is not difficult to realize that, after iteration n in this algorithm, the prefix $x_1 \cdots x_n$ is quasi-normalized if all identity operations in $x_1 \cdots x_n$ are removed (although for the sake of efficiency, these removals take place after the last iteration and the main loop skips over identity operations). When x_{n+1} is an insertion (j, σ), it is moved by applying the equivalence (3.1) until (j, σ) is placed after an operation (i_1, γ_1) such that $(i_1, \gamma_1) \prec (j, \sigma)$ and before a subsequence $(i_2, \gamma_2)(i_3\gamma_3) \cdots$ such that $j \leq i_2$. The result is a new prefix containing $\cdots (i_1, \gamma_1)(j, a)(i_2 + 1, \gamma_2)(i_3 + 1, \gamma_3) \cdots$ which is quasi-normalized because $j < i_2 + 1$ and $(i_2, \alpha_2)(i_3, \alpha_3) \cdots$ is quasi-normalized. When x_{n+1} is a deletion $(j, \#)$, it is moved by applying equivalence (3.2) until $(j, \#)$ is placed after a subsequence $\cdots (i_1, \gamma_1)(i_2, \gamma_2)$ such that $(i_2, \gamma_2) \prec (j, \#)$ and before a subsequence $(i_3, \gamma_3)(i_4, \gamma_4) \cdots$ such that $j < i_3$ with two possible results:

- In the case of $i_2 = j$ and $\gamma_2 \in \Sigma$, the equivalence (3.3) is immediately applied and a prefix containing $\cdots (i_1, \gamma_1)(i_3 - 1, \gamma_3)(i_4 - 1, \gamma_4) \cdots$ is obtained, which is quasi-normalized because $\gamma_2 \in \Sigma$ implies $i_1 < i_2 < i_3$.
- Otherwise, a prefix containing the subsequence $\cdots (i_1, \gamma_1)(i_2, \gamma_2)(j, \#)(i_3 - 1, \gamma_3)(i_4 - 1, \gamma_4) \cdots$ is obtained, which is quasi-normalized because $j \leq i_3 - 1$ and $(i_3, \gamma_3)(i_4, \gamma_4) \cdots$ is quasi-normalized.

Interestingly, a quasi-normalized edit sequence can be transformed into an equivalent normalized edit sequence with a second application of the algorithm 1. Note that, in a quasi-normalized subsequence, all unnormalized subsequences match the pattern $x(k,\#)^m(k,\sigma_0)(k{+}1,\sigma_1)\cdots(k{+}n,\sigma_n)y$ with $x \neq (k,\#)$, $m > 0$, and $y \neq (k{+}n{+}1,\sigma_{n+1})$. After equivalence (3.1) has been applied, these patterns lead to a normalized subsequence of the form $x(k,\sigma_0)(k+1,\sigma_1)\cdots(k+n,\sigma_n)(k+n+1,\#)^m y$ which contains neither empty nor unnormalized sequences: on the one hand, $x \prec (k,\#)$ and $x \neq (k,\#)$ implies $x \prec (k,\sigma_0)$; on the other hand, y modifies a position which is strictly larger than $k+n$ but it is not of the type $(k+n+1,\sigma_{n+1})$ and, thus, $(k+n+1,\#) \prec y$.

Therefore, the edit sequence obtained after the application of Algorithm 1 to a quasi-normalized sequence is normalized. For practical applications, this quadratic-time algorithm can be replaced, as is described in Appendix, by a much more efficient combination of a sorting method based on merge sort – which works in $\mathcal{O}(N \log N)$ time – followed by a linear time procedure that sorts quasi-normalized blocks.

4. Updating edit sequences

The procedure described in the previous section permits the identification of equivalent edit sequences, since they share identical normal forms. It also permits the systematic comparison of log files L_i storing edit sequences performed at the local replica stored at site s_i derived from a common source file S. If two edit sequences L_1 and L_2 have normalized equivalents $\bar{L}_1 = (i_1,\alpha_1)\cdots(i_M,\alpha_M)$ and $\bar{L}_2 = (j_1,\beta_1)\cdots(j_N,\beta_N)$ respectively which are not identical, it is possible to create a new edit sequence $L_M = L_1 \oplus L_2$ which integrates the information contained in L_1 and L_2.

A naive solution will propagate non-conflicting operations in L_1 and L_2 to L_M: for example, those identical in the normalized sequences $\bar{L}_1$ and $\bar{L}_2$ or accessing disjoint contents in the text. However, it is well known that the design of a procedure that maintains consistency between replicas is a non-trivial challenge in operation transformations [22], where all sites must reach a convergent state at quiescence. Moreover, proving the correctness of the transformation is a subtle task and various methods with which to verify this automatically [7] have been proposed. The proof essentially checks whether some sufficient conditions are met but the question if weaker requirements may suffice does not appear to be established.

Figure 1 illustrates how the result of merging three normalized edit sequences e_1, e_2 and e_3 differs if identical insertions – the leading $(0,a)$ in $e_1 \oplus e_2$ and e_3 – are merged during the construction of the synchronized sequence L_M. As customarily done, a precedence of sites (here, lower identifiers have higher priority) is applied to determine the order in which insertions with identical positions that originated at different sites must be applied. A careful analysis shows that this kind of insertions, even if they are sorted according to the priority of the originating site,

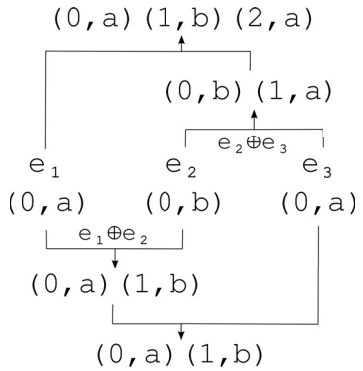

FIGURE 1. Merging two edit sequences from three different sites: $e_1 \oplus (e_2 \oplus e_3)$ and $(e_1 \oplus e_2) \oplus e_3$.

is the source of the inconsistency in the results. Indeed, concurrent insertions may be temporally placed at different positions in the sequence L_M, depending on the order in which the information regarding external editions arrives at the site, but they can only be merged when they are contiguous. Note that this cannot be the case with deletions, because a delete either removes a character inserted previously on the same site (and its effect is then the cancellation of the former operation) or it removes a character that was already in the original document; therefore, there is no ambiguity in the effect a delete generates in the source text.

Owing to this ambiguity, the early *dOPT* algorithm [5] was found to be incorrect. The improved algorithm *adOPTed* [17] solved this issue – as proved by [12] –, essentially by preserving all insertions. This choice generates some unexpected results, such as a duplicated character a in the former example in which two users perform identical corrections. However, this will make it easier to supervise the result when, for example, one user adds the word *as* and a second user concurrently adds the word *all*, since the merged file will show the whole information through the string *asall*.

Therefore, in order to build $L_M = L_1 \oplus L_2$ from the normalized sequences $\bar{L}_1$ and $\bar{L}_2$, the reconciliation process should preserve the semantics of the users' actions. For example, $abc \cdot (1,\#)(1,\#) = a$, but the addition of an insertion such as $(0,a)$ as the very first operation in the sequence leads to the unexpected result $abc \cdot (0,a)(1,\#)(1,\#) = ac$. The addition of an insertion (i,σ) to a normalized edit sequence before an operation (k,γ) such that $(i,\sigma) \prec (k,\gamma)$ clearly begs an increase in the position of all operations later in the edit sequence – so that the example becomes $(0,a)(2,\#)(2,\#)$ – while removing an insertion (i,σ) begs for a decrease in their value.[2] We thus define $\delta(\#) = -1$ and $\delta(\gamma) = 1$ for all $\gamma \neq \#$. It is possible

[2]The manipulation of deletions requires further considerations because removing, for example, the first $(0,\#)$ in $(0,\#)(0,\#)(1,a)(2,b)$ should lead to the sequence $(0,\#)(2,a)(3,b)$, that is, only the removal of the last deletion in a run follows the analogous to the simple rule for insertion removal.

then to apply the following set of propagation rules, provided that two indices m and n are used to traverse the normalized sequences $\bar{L}_1 = (i_1, \alpha_1) \dots (i_n, \alpha_N)$ and $\bar{L}_2 = (j_1, \beta_1) \dots (j_M, \beta_M)$ respectively and two parameters ω_1 and ω_2 represent the offsets that must be applied to later operations in $\bar{L}_1$ and $\bar{L}_2$:

- A0: add $(i_m + \omega_1, \alpha_m)$ to L_M and set $m = m + 1$ and $n = n + 1$.
- A1: add $(i_m + \omega_1, \alpha_m)$ to L_M and set $m = m + 1$ and $\omega_2 = \omega_2 + \delta(\alpha_m)$.
- A2: add $(j_n + \omega_2, \beta_n)$ to L_M and set $n = n + 1$ and $\omega_1 = \omega_1 + \delta(\beta_n)$.

The first action (A0) will be applied only if $i_m + \omega_1 = j_n + \omega_2$ and $\alpha_m = \beta_n = \#$. The second action (A1) will be applied in the following cases:

1. $n = N$ ($\bar{L}_2$ exhausted).
2. $i_m + \omega_1 < j_n + \omega_2$ (position precedence).
3. $i_m + \omega_1 = j_n + \omega_2$ and $\alpha_m \neq \#$ and $\beta_n = \#$ (precedence of insertions over deletes).
4. $i_m + \omega_1 = j_n + \omega_2$, $\alpha_m \neq \#$, $\beta_n \neq \#$ and s_1 has precedence over s_2 (site precedence).

In all the remaining cases, the third action (A2) will be applied. The preference for insertions over deletions makes the result L_M quasi-normalized but its normalization requires the application of the block sorting procedure described in the previous section. The symmetry of the procedure guarantees that $L_1 \oplus L_2 = L_2 \oplus L_1$ and also $(L_1 \oplus L_2) \oplus L_3 = L_1 \oplus (L_2 \oplus L_3)$.

The sequence L_M obtained with this procedure at site s_i can be applied to the original state in order to obtain a convergent updated state. However, if the source state has not been preserved and only the most recent state is available, it is necessary to compute L'_i such that $L_M \equiv L_i L'_i$. The edit sequence L'_i can be obtained by comparing L_M with L_i.

The removal of operations in L_i which are not in L_M affects only some of the operations later in the edit sequence, depending, in general, on the intermediate operations. Fortunately, any operation can be easily moved until it becomes the last operation in the edit sequence by applying the following set of commutation relations, which are valid for all $a, b \in \Sigma$:

$$
\begin{aligned}
(i,a)(j,b) &\equiv \begin{cases} (j-1,b)(i,a) & \text{if } i < j \\ (j,b)(i+1,a) & \text{if } i \geq j \end{cases} \\
(i,a)(j,\#) &\equiv \begin{cases} (j-1,\#)(i,a) & \text{if } i < j \\ \epsilon & \text{if } i = j \\ (j,\#)(i-1,a) & \text{if } i > j \end{cases} \\
(i,\#)(j,b) &\equiv \begin{cases} (j+1,b)(i,\#) & \text{if } i \leq j \\ (j,b)(i+1,\#) & \text{if } i \geq j \end{cases} \\
(i,\#)(j,\#) &\equiv \begin{cases} (j+1,\#)(i,\#) & \text{if } i \leq j \\ (j,\#)(i,\#) & \text{if } i = j \\ (j,\#)(i-1,\#) & \text{if } i > j. \end{cases}
\end{aligned}
\tag{4.1}
$$

Once the operation becomes the last operation in the sequence, it can be safely removed. For example, $abcde \cdot (2,\#)(0,\#)(0,\#)(1,\#) = d$, while simply removing the initial operation $(2,\#)$ does not preserve the user intention because $abcde \cdot (0,\#)(0,\#)(1,\#) = ce$. However, by applying the equivalences shown above in order to move the initial operation $(2,\#)$ backwards, one obtains $(2,\#)(0,\#)(0,\#)(1,\#) \equiv (0,\#)(0,\#)(2,\#)(1,\#)$. After the last operation in the sequence is removed, the expected result $abcde \cdot (0,\#)(0,\#)(2,\#) = cd$ is then obtained. It is worth noting that, in some cases, the moved operation is cancelled with a later operation owing to the application of the forth equivalence in (4.1) – $(i,\#)(i,a) = \epsilon$ –, and the process thus stops before the end of the sequence is reached. This situation reflects the fact that an operation equivalent to that which is to be removed has been later performed in the sequence, and it is a thus reasonable choice to keep just one of them.

The edit operations in L_M which are not in L_i can neither be safely inserted in the middle of the sequence (since later operations may require updating) nor appended unchanged at the end. For example, if the operation x_1 is added before the operation x_2, then x_2 should be rewritten as x_2' such that $x_1 x_2' \equiv x_2 x_1'$ for some operation x_1'. Thus, for $x_1, x_2, y_1, y_2 \in E$, we will write $x_1^R y_1 \equiv x_2 y_2^R$ if and only if $x_1 x_2 \equiv y_1 y_2$. These equivalences can easily be derived from the commutation relations (4.1):

$$
\begin{aligned}
(i,a)^R\,(j,b) &\equiv \begin{cases} (j+1,b)(i,a)^R & \text{if } i \le j \wedge a \neq b \\ (j,b)(i+1,a)^R & \text{if } i \ge j \wedge a \neq b \\ \epsilon & \text{if } i = j \wedge a = b \end{cases} \\
(i,a)^R\,(j,\#) &\equiv \begin{cases} (j+1,\#)(i,a)^R & \text{if } i \le j \\ (j,\#)(i-1,a)^R & \text{if } i > j \end{cases} \\
(i,\#)^R\,(j,b) &\equiv \begin{cases} (j-1,b)(i,\#)^R & \text{if } i < j \\ (j,b)(i+1,\#)^R & \text{if } i \ge j \end{cases} \\
(i,\#)^R\,(j,\#) &\equiv \begin{cases} (j-1,\#)(i,\#)^R & \text{if } i < j \\ \epsilon & \text{if } i = j \\ (j,\#)(i-1,\#)^R & \text{if } i > j. \end{cases}
\end{aligned}
\tag{4.2}
$$

If it is necessary to insert an operation z before an edit sequence $x_1 \cdots x_N$, a fast procedure to compute the resulting sequence is, therefore, to move backwards z^R in the sequence $z z^R x_1 \cdots x_N$ using the equivalences (4.2) until either the moved operation is cancelled or the end of the sequence is reached (and the moved operation is then removed). For example, in order to add a deletion $(2,\#)$ before the sequence $(0,\#)(1,\#)$, $(2,\#)^R(0,\#)(1,\#) \equiv (0,\#)(1,\#)^R(1,\#) \equiv (0,\#)$ is computed, and the resulting edit sequence after the addition is therefore $(2,\#)(0,\#)$.

In summary, we can implement a procedure to obtain a new edit sequence L_i' from L_M that can be applied to the current state. The algorithm starts with an empty list L_i' and proceeds iteratively as follows:

– Find the first operation z which differs in L_M and L_i.

- If z is in L_i but not in L_M (z has been removed with regard to L_i), then move z backwards in L_iL_i' using the equivalences (4.1) until either the equivalence $(i,a)(i,\#) \equiv \epsilon$ (third equivalence in the list) is applied or the end of the sequence is reached. In the last case, remove the last operation in the edit sequence.
- If z is in L_M but not in L_i (z has been added with regard to L_i), then add zz^R to L_iL_i' and move z^R backwards using the equivalences (4.2) until either the equivalence $z^Rz \equiv \epsilon$ (third or nine equivalences in the list) can be applied or the end of the sequence is reached. In the last case, remove the last (reverse) operation in the edit sequence.

5. Conclusion

We have defined a normalized form for the sequences of edit operations performed on a text file. This normalized form is unique, and thus permits the direct comparison of local editions which have been performed on a common source text file. We also provide simple and efficient algorithms with which to obtain a normalized edit sequence from a general sequence.

In comparison to traditional replication methods that guarantee consistency [12], the temporal precedence is not enforced by our approach, since our normalization essentially maintains a precedence based on the position of the source file affected by the edit operation so that later edits cannot influence the preceding ones.

This normalization avoids one of the inefficiencies in the transmission of operation logs: if a block of operations cancels a previous operation, for example a deletion removes a large set of changes performed previously, then the normalized edit sequence does not include these edit operations. This question has been previously addressed with procedures specifically designed for the log files transferred in replication [19]. The other obvious source of inefficiency, changes that have been identically performed on both sites, can be avoided by using delta compression applied to the normalized logs.

The formalism can be easily extended so that delete operations also contain which character is erased in the text. This extension makes some equivalences between edit sequences unsafe because they depend on the text they operate on: for instance, if $(0,\bar{a})$ denotes the deletion of a character a at position 1, the output when $(0,\bar{a})(0,a)$ operates on $S = s_1 \cdots s_L$ will be undefined if $s_1 \neq a$ but equivalent to ϵ otherwise. However, the extension may present some advantages which are worth exploring:

- One can define the reverse of edit sequences because the original file can be recovered from the output and the applied edit sequence, provided that the output is not undefined.
- If $S \cdot X = S \cdot Y$ is not undefined, then X and Y generate identical output when applied to any source file (provided that both results are not undefined).

– The length of the shortest equivalent sequence is related to the *indel distance* (the edit distance when substitutions are not permitted) between the source and the output text.

We also plan to explore the extension of the ideas presented here to the case of the comparison of structured texts – such as those contained in XML documents, configuration files or syntactically parsed content – exploiting both the particular features that its synchronization presents [6,10] and previous ideas developed for the comparison of tree structures. [2,25]

Acknowledgements. This research was partially supported by the Spanish CICyT through grant TIN2006-15071-C03-01 and the Spanish Ministry of Foreign Affairs and Cooperation through a MAEC-AECID grant (program II-A).

Appendix. Faster normalization based on merge sort

The construction of a quasi-normalized edit sequence can be implemented as an adaptation of the merge sort algorithm. The key in this procedure is a modified merge function [3] which builds a normalized edit sequence Z from two normalized edit sequences $X = x_1x_2\cdots x_M$ and $Y = y_1y_2\cdots y_N$. This function proceeds iteratively and at every iteration a suffix $x_mx_{m+1}\cdots x_M$, a suffix $y_ny_{n+1}\cdots y_N$ and a prefix $z_1z_2\cdots z_r$ of the final result Z are considered. After every iteration the algorithm guarantees that $z_r \prec x_m \wedge z_r \prec y_n$ and also that the sequence $z_1z_2\cdots z_r$ is quasi-normalized. After the last iteration is complete, a final sorting procedure (Algorithm 2) is applied which transforms a quasi-normalized edit sequence into a normalized sequence in linear time.

The merge function uses to indexes, m and n, to iterate over the normalized edit sequences X and Y respectively and an auxiliary integer ω that stores the offset that must be applied to the positions in $x_{m+1}\cdots X_M$ owing to the edit operations in $y_1\cdots y_{n-1}$, which have been previously added to the output Z. If $m > M$ or $n > N$, the function adds the remaining operations ($y_n\cdots y_N$ or $x_m\cdots x_M$ respectively) before returning Z. Otherwise, it examines the initial operations in $x_m\cdots x_M$ and $y_n\cdots y_N$: let $x_m = (i,\alpha)$ and, if $m < M$, $x_{m+1} = (j,\beta)$ and let $y_n = (k,\gamma)$ and, if $n < N$, $y_{n+1} = (l,\delta)$. If y_n is an insertion (k,γ) with $\gamma \neq \#$, only two cases need to be considered:

(1) If $i+\omega < k$, then add $(i+\omega,\alpha)$ to Z and set $m = m+1$.
(2) Otherwise, add (k,γ) to the output and set $n = n+1$ and $\omega = \omega+1$.

However, if $\gamma = \#$, all the following cases must be considered:

(1) If $i+\omega < k$, then add $(i+\omega,\alpha)$ to Z and set $m = m+1$.
(2) If $i+\omega = k$ a number of sub-cases may appear,
   (a) If $j > i+1$ (or $m \geq M$) and $\alpha \neq \#$ then set $m = m+1$, $n = n+1$ and $\omega = \omega-1$.
   (b) If $j > i+1$ (or $m \geq M$) and $\alpha = \#$ then add $(k,\#)(i+\omega,\#)$ to Z and set $m = m+1$, $n = n+1$ and $\omega = \omega-1$.

(c) If $j = i$ then add $(i + \omega, \#)$ to Z and set $m = m + 2$.
(d) If $j = i + 1$:
  (i) If $\alpha \neq \#$ then add $(k, \#)(i + \omega, \#)$ to Z.
  (ii) Set $m = m + 2$, $n = n + 1$ and $\omega = \omega - 1$.
  (iii) If $\beta \neq \#$ then
    (A) If $n = N$ or $\delta \neq \#$ or $l > k$ then add $(j + \omega - 1, \beta)$ to Z.
    (B) Otherwise, set $n = n + 1$ and $\omega = \omega - 1$.
(e) If $j = i + 1$ and $\beta = \#$ then add (k, β) to Z and set $m = m + 1$, $n = n + 1$ and $\omega = \omega - 1$.

(3) If $i + \omega = k + 1$ then add $(k, \#)(k, \alpha)$ to Z and set $m = m + 1$, $n = n + 1$ and $\omega = \omega - 1$. Moreover,
(a) If $\alpha = \#$ or $n = N$ or $\delta \neq \#$ or $l \neq k$ then add $(i + \omega - 1, \alpha)$ to Z.
(b) Otherwise, set $n = n + 1$ and $\omega = \omega - 1$.

(4) If $i + \omega > k + 1$, then add $(k, \#)$ to Z, and set $n = n + 1$ and $\omega = \omega - 1$.

After every merge, the sorting Algorithm 2 is called. This function essentially counts the number of deletions in a run (line 5) and if they are followed by the suitable insertions (matching the pattern described at the end of Sect. 2), every insertion is then swapped with a deletion in the preceding run (lines 7-8). Once the pattern end is reached, all deletions in the run become identical (see line 12).

Algorithm 2 `blockSort`(X)

Input: A quasi-normalized edit sequence $X = x_1 \cdots x_N$
Output: A normalized edit sequence Y such that $X \equiv Y$.

```
1:  length = 1
2:  for n = 2, ... N do
3:    (k, γ) ← x_n
4:    if γ = # ∧ x_n = x_{n−1} then
5:      length ← length + 1
6:    else if x_{n−1} ⋠ x_n then
7:      x_{n−length} ← x_n
8:      x_n ← (k + 1, #)
9:    else
10:     while length > 1 do
11:       x_{n−length} ← x_{n−1}
12:       length ← length − 1
13:     end while
14:   end if
15: end for
16: while length > 1 do
17:   x_{N+1−length} ← x_N
18:   length ← length − 1
19: end while
20: return X
```

References

[1] S. Balasubramaniam and B.C. Pierce, What is a file synchronizer, in *Forth Annual ACM/IEEE International Conference on Mobile Computing and Networking* (*MobiCom'98*) (1998) 98–1085.

[2] P. Bille, A survey on tree edit distance and related problems. *Theoret. Comput. Sci.* **337** (2005) 217–239.

[3] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to algorithms*. 6th edition, MIT Press and McGraw-Hill Book Company (1992).

[4] R. Cox and W. Josephson, *File synchronization with vector time pairs*. Technical Report MIT-CSAIL-TR-2005-014 and MIT-LCS-TM-650, MIT Computer Science and Artificial Intelligence Laboratory (2005).

[5] C.A. Ellis and S.J. Gibbs, Concurrency control in groupware systems, in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data.* J. Clifford, B.G. Lindsay and D. Maier, Eds. ACM Press (1989) 399–407.

[6] J.N. Foster, M.B. Greenwald, C. Kirkegaard, B.C. Pierce and A. Schmitt, Exploiting schemas in data synchronization. *J. Comput. System Sci.* **73** (2007) 669–689.

[7] A. Imine, M. Rusinowitch, G. Oster and P. Molli, Formal design and verification of operational transformation algorithms for copies convergence. *Theoret. Comput. Sci.* **351** (2006) 167–183.

[8] A.-M. Kermarrec, A.I.T. Rowstron, M. Shapiro and P. Druschel, The IceCube approach to the reconciliation of divergent replicas, in *PODC 2001, Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*. ACM (2001) 210–218.

[9] S. Khanna, K. Kunal and B.C. Pierce, A formal investigation of diff3, in *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Arvind and Prasad, Eds. (2007).

[10] T. Lindholm, A three-way merge for XML documents, in *DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering*. ACM, New York, NY, USA (2004) 1–10.

[11] E. Lippe and N. van Oosterom, Operation-based merging. *SIGSOFT Softw. Eng. Notes* **17** (1992) 78–87.

[12] B. Lushman, and G.V. Cormack, Proof of correctness of Ressel's adOPTed algorithm. *Inform. Process. Lett.* **86** (2003) 303–310.

[13] W.J. Masek and M.S. Paterson, A faster algorithm computing string edit distances. *J. Comput. System Sci.* **20** (1980) 18–31.

[14] F. Mattern, Virtual time and global states of distributed systems, in *Proc. Workshop on Parallel and Distributed Algorithms*, M. Cosnard, Ed., Chateau de Bonas, France. Elsevier (1988) 215–226.

[15] B.C. Pierce and J. Vouillon, *What's in Unison? A formal specification and reference implementation of a file synchronizer*. Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania (2004).

[16] N. Ramsey and E. Csirmaz, An algebraic approach to file synchronization. *SIGSOFT Softw. Eng. Notes* **26** (2001) 175–185.

[17] M. Ressel, D. Nitsche-Ruhland and R. Gunzenhäuser, An integrating, transformation-oriented approach to concurrency control and undo in group editors, in *CSCW '96, Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work*. Boston, MA, USA, ACM (1996) 288–297.

[18] Y. Saito and M. Shapiro, Optimistic replication. *ACM Comput. Surv.* **37** (2005) 42–81.

[19] H. Shen and C. Sun, A log compression algorithm for operation-based version control systems, in *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, *COMPSAC '02*. IEEE Computer Society Washington, DC, USA (2002) 867–872.

[20] T. Suel and N. Memon, Algorithms for delta compression and remote file synchronization, in *Lossless Compression Handbook*, K. Sayood, Ed. Academic Press (2003) 269–290.

[21] C. Sun and C.A. Ellis, Operational transformation in real-time group editors: Issues, algorithms, and achievements, in *CSCW98, Proceedings of the ACM 1998 Conference on Computer Supported Cooperative Work*. ACM (1998) 59–68.
[22] C. Sun, X. Jia, Y. Zhang, Y. Yang and D. Chen, Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Computer-Human Interaction* **5** (1998) 63–108.
[23] W.F. Tichy, The string-to-string correction problem with block moves. *ACM Trans. Comput. Syst.* **2** (1984) 309–321.
[24] A. Tridgell and P. Mackerras, *The rsync algorithm*. Technical Report TR-CS-96-05, Department of Computer Science, Faculty of Engineering and Information Technology, The Australian National University (1996).
[25] K. Zhang and D. Shasha, Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.* **18** (1989) 1245–1262.

Communicated by R. Baeza-Yates.
Received May 7, 2009. Accepted February 7, 2011.