A Formal Investigation of `Diff3`

Sanjeev Khanna[1], Keshav Kunal[2], and Benjamin C. Pierce[1]

[1] University of Pennsylvania
[2] Yahoo

Abstract. The `diff3` algorithm is widely considered the gold standard for merging uncoordinated changes to list-structured data such as text files. Surprisingly, its fundamental properties have never been studied in depth.

We offer a simple, abstract presentation of the `diff3` algorithm and investigate its behavior. Despite abundant anecdotal evidence that people find `diff3`'s behavior intuitive and predictable in practice, characterizing its good properties turns out to be rather delicate: a number of seemingly natural intuitions are incorrect in general. Our main result is a careful analysis of the intuition that edits to "well-separated" regions of the same document are guaranteed never to conflict.

1 Introduction

Users often want to edit a local copy of a replicated data structure, postponing the moment when their changes become visible to others until sometime later—when a set of changes has been finished and tested, when an offline laptop is reconnected to the network, etc. In general, when multiple users can edit at the same time, this reconciliation process requires a tool—a *synchronizer*—that can propagate non-conflicting changes between different copies of the data, while recognizing and flagging conflicts. Source code management systems, long-distance collaborative editing environments, and file synchronizers are examples.

Operation-based synchronizers work by keeping track of the complete sequences of operations that have been applied to each replica and, during reconciliation, attempting to synthesize a single unified view of the data structure's edit history. By contrast, a *state-based* synchronizer sees only the current versions of the replicas to be reconciled, together with an *archive* of the last state they had in common (perhaps saved away at the end of the last synchronization).

A crucial problem faced by a state-based synchronizer is how to *align* the information in the current replicas and the archive, so that it can tell where changes have been made. This can be accomplished in a variety of ways, depending on the nature of the data being synchronized. Where the data is rigidly structured or where keys are available (e.g., in personal information management applications such as address books), the proper alignment is generally clear. For more flexibly structured data, such as semistructured databases, file systems, and text documents, it is less clear how to reliably choose alignments that users consider natural. The issue is particularly vexing for pure textual (or, more generally, *list-structured*) data, which offers no predetermined points of reference for alignment—the structures are presented to the synchronizer as flat sequences of uninterpreted atoms (characters, words, or lines of text)—and for which common edits include arbitrary insertions, deletions, and rearrangements of existing material.

The best known tool for synchronization of textual data is `diff3`. Developed by Randy Smith in 1988 [1] and popularized in revision control systems such as CVS and Subversion, `diff3` and its relatives are relied on by millions of users for a huge range of collaborative tasks. The basic ideas of `diff3` also appear in numerous hybrid tools for synchronizing semi-structured data in formats like XML, such as Lindholm's 3DM [2], the work of Chawathe *et al.* [3], and FCDP [4].

Given its popularity, it is surprising that the fundamental properties of the `diff3` algorithm have never been explored. The published descriptions of its behavior (the GNU `difftools` manual [5] and comments in the source code) are helpful but rather low-level and operational, and we have been unable to find in the literature any rigorous analysis of the properties that users might want or expect from `diff3` and the circumstances under which they hold.

Our first contribution is to put the `diff3` algorithm itself on a more rigorous footing by offering a concise description of its behavior (§2-§3). Our model here is `diff` [6,7,8]—the two-way comparison algorithm used as a subroutine by `diff3`—which has not one but two elegant specifications: it can be viewed as computing either a longest common subsequence of its two inputs or a minimum-length edit script for turning one into the other by single-element insertions and deletions. Our specification of `diff3` is not quite this concise, but nearly. We give a compact reference implementation in half a page of pseudo-code.

Our second and main contribution is an analysis of `diff3`'s properties (§4). Most importantly, we examine the common intuition that, if the changes to the replicas are *local* to distinct and "well separated" regions, then `diff3` will always be able to merge them without conflicts. We show that the most obvious formulations of this intuition are, in fact, wrong, but identify a common and easily checked separation condition under which the property does hold. We also formalize intuitive notions of *idempotence* (the results of synchronization are "fully synchronized" except where edits conflict), *stability* (similar inputs lead to similar outputs), and the guarantee of *near-complete success* when the inputs have been changed in similar ways (even if these changes are large compared to the archive version), and show that none of these properties hold in general.

We cite only closely related work. Broader surveys of the literature on synchronization algorithms for other kinds of data and algorithms founded on different assumptions (such as operation-based techniques) can be found in [9,10].

2 Warmup

Let us begin with a small example illustrating the basic operation of `diff3`. Figure 1(a) shows the initial configuration: O is the archive—the last common version—and A and B are the current versions that have diverged from O. (Whoever edited A has swapped $4, 5$ and $2, 3$, while 3 has gotten moved after 5 in B.) The first thing `diff3` does is to call the two-way comparison tool `diff` to find *maximum matchings* (or longest common subsequences) between O and A

$A = [1, 4, 5, 2, 3, 6]$
$O = [1, 2, 3, 4, 5, 6]$
$B = [1, 2, 4, 5, 3, 6]$

(a) inputs

A	1	4	5	2	3			6
O	1			2	3	4	5	6

O	1	2	3	4	5		6
B	1	2		4	5	3	6

(b) maximum matches

A	1	4,5	2	3	6
O	1		2	3,4,5	6
B	1		2	4,5,3	6

(c) `diff3` parse

A'	1	4,5	2	3	6
O'	1	4,5	2	3,4,5	6
B'	1	4,5	2	4,5,3	6

(d) calculated output

```
1
4
5
2
<<<<<<< A
3
||||||| O
3
4
5
=======
4
5
3
>>>>>>> B
6
```

(e) printed output

Fig. 1. Warmup Example

and between O and B, as shown in Figure 1(b). It then takes the regions where O differs from either A or B and coalesces the ones that overlap, leading to the alternating sequence of *stable* (all replicas equal) and *unstable* (one or both replicas changed) *chunks* shown in Figure 1(c).[1] Finally, it examines what has changed in each chunk and decides what changes can be propagated, as shown in Figure 1(d)—here, the second chunk is changed only in A (by inserting $4, 5$), so this change can be propagated to B, but the fourth chunk has changes in both A and B, so nothing can be propagated.

At this point, the actual `diff3` tool is finished: it simply walks over the chunks and, depending on what flags are provided on the command line, outputs something appropriate for each chunk. For example, Figure 1(e) shows the output from invoking `diff3 -m A O B`, where the `-m` flag requests a merged version of the files. For non-conflicting chunks, a single version is printed; for conflicts, the whole chunk.

Our analysis is a tiny bit more refined: We consider `diff3` as having *three* outputs—the new versions of A, O, and B with all non-conflicting changes in A reflected in B' and O' and all non-conflicting changes in B reflected in A' and O'. At the same time, we calculate a new archive O' that reflects all the changes that were successfully propagated, keeping the state from O in conflicting regions. (This extra refinement is just for purposes of analysis. In principle, it could also be useful in practice: after a partially successful synchronization, the current replicas are left in a partially updated but usable state, in contrast with tools like CVS based on the actual `diff3`, where conflicts cause the current replicas to

[1] The `diff3` manual [11] uses the term *hunks* for what we are calling unstable chunks; stable chunks are not named explicitly.

be polluted with information about conflicting chunks. However, we will see in §4.2 that re-running `diff3` after a partially conflicting run can have unexpected consequences.)

3 The `Diff3` Algorithm

We assume given some set of *atoms* $\mathcal{A}$. (In practice, these might be lines of text, as in GNU `diff3`, or they could be words, characters, etc.) We write $\mathcal{A}^*$ for the set of lists with elements drawn from $\mathcal{A}$ and use variables J, K, L, O, A, B, and C to stand for elements of $\mathcal{A}^*$. If L is a list and $k \in \{1, \ldots, |L|\}$, then $L[k]$ denotes the kth element of L. A *span* in a list L is a pair of indices $[i..j]$ with $1 \leq i, j \leq |L|$. We write $L[i..j]$ for the list of elements of L in locations i through j; if $j < i$, this is the empty list. The *length* of a span $[i..j]$ is $j - i + 1$ if $i \leq j$ and 0 if $i > j$.

A *configuration* is a triple $(A, O, B) \in \mathcal{A}^* \times \mathcal{A}^* \times \mathcal{A}^*$. We usually write configurations in the more suggestive notation $(A \leftarrow O \rightarrow B)$ to emphasize that O is the archive from which A and B have been derived.

A *synchronizer* is a function that takes as input a configuration $(A \leftarrow O \rightarrow B)$ and yields another configuration $(A' \leftarrow O' \rightarrow B')$. We say that $(A \leftarrow O \rightarrow B) \Rightarrow (A' \leftarrow O' \rightarrow B')$ is a *run* of the synchronizer. A run $(A \leftarrow O \rightarrow B) \Rightarrow (C \leftarrow C \rightarrow C)$, where the three components of the output configuration are identical, is said to be *conflict free*. We write $(A \leftarrow O \rightarrow B) \Rightarrow C$ in this case.

The first step of `diff3` is to call a two-way comparison subroutine on (O, A) and (O, B) to compute a non-crossing matching M_A between the indices of O and A—that is, a boolean function on pairs of indices from O and A such that if $M_A[i, j] = \mathit{true}$ then (a) $O[i] = A[j]$, (b) $M_A[i', j] = \mathit{false}$ and $M_A[i, j'] = \mathit{false}$ whenever $i' \neq i$ and $j' \neq j$, and (c) $M_A[i', j'] = \mathit{false}$ whenever either $i' < i$ and $j' > j$ or $i' > i$ and $j' < j$—and a non-crossing matching M_B between the indices of O and B. We treat this algorithm as a black box, simply assuming (a) that it is deterministic, and (b) that it always yields *maximum* matchings. For the counterexamples in the next section, we have verified that the matchings we use correspond to the ones actually chosen by GNU `diff3`.

A *chunk* (from A, O, and B) is a triple $H = ([a_i..a_j], [o_i..o_j], [b_i..b_j])$ of a span in A, a span in O, and a span in B such that at least one of the three is non-empty. The *size* of a chunk is the sum of the lengths of all three spans. Write $A[H]$ for $A[a_i..a_j] \in \mathcal{A}^*$, and similarly $O[H] = O[o_i..o_j]$ and $B[H] = B[b_i..b_j]$.

A *stable chunk* is a chunk in which all three spans have the same length and corresponding indices are matched in all three—i.e., a chunk $([a..a+k-1], [o..o+k-1], [b..b+k-1])$ for some $k > 0$, with $M_A[o+i, a+i] = M_B[o+i, b+i] = \mathit{true}$ for each $0 \leq i < k$. That is, a stable chunk corresponds to a span in O that is matched in both M_A and M_B. An *unstable chunk* is one that is not stable. An unstable chunk H is classified as follows:

H is *changed in A*	if	$O[H] = B[H] \neq A[H]$
H is *changed in B*	if	$O[H] = A[H] \neq B[H]$
H is *falsely conflicting*	if	$O[H] \neq A[H] = B[H]$
H is *(truly) conflicting*	if	$O[H] \neq A[H] \neq B[H] \neq O[H]$

```
1. Initialize $\ell_O = \ell_A = \ell_B = 0$.
2. Find the least positive integer $i$ such that either $M_A[\ell_O + i, \ell_A + i] = false$
   or $M_B[\ell_O + i, \ell_B + i] = false$. If $i$ does not exist, then skip to step 3 to
   output a final stable chunk.
   (a) If $i = 1$, then find the least integer $o > \ell_O$ such that there exist
       indices $a, b$ with $M_A[o, a] = M_B[o, b] = true$. If $o$ does not exist, then
       skip to step 3 to output a final unstable chunk. Otherwise, output the
       (unstable) chunk

           $C = ([\ell_A + 1 .. a - 1], [\ell_O + 1 .. o - 1], [\ell_B + 1 .. b - 1])$.

       Set $\ell_O = o - 1$, $\ell_A = a - 1$, and $\ell_B = b - 1$, and repeat step 2.
   (b) If $i > 1$, output the (stable) chunk

           $C = ([\ell_A + 1 .. \ell_A + i - 1], [\ell_O + 1 .. \ell_O + i - 1], [\ell_B + 1 .. \ell_B + i - 1])$.

       Set $\ell_O = \ell_O + i - 1$, $\ell_A = \ell_A + i - 1$, and $\ell_B = \ell_B + i - 1$, and repeat
       step 2.
3. If ($\ell_O < |O|$ or $\ell_A < |A|$ or $\ell_B < |B|$), output a final chunk

           $C = ([\ell_A + 1 .. |A|], [\ell_O + 1 .. |O|], [\ell_B + 1 .. |B|])$.
```

Fig. 2. The `Diff3` Algorithm

A chunk is called *conflicting* if it is either falsely or truly conflicting; a *non-conflicting* chunk is thus either stable or else changed only in A or B. Given a chunk H, we define the *output* of H to be the following triple of lists:

$$out(H) \quad = \quad \begin{cases} (A[H], O[H], B[H]) \text{ if } H \text{ is stable or conflicting} \\ (A[H], A[H], A[H]) \text{ if } H \text{ is changed in } A \\ (B[H], B[H], B[H]) \text{ if } H \text{ is changed in } B \end{cases}$$

A `diff3` *parse* of A, O, and B with respect to the matchings M_A and M_B is a sequence of stable and unstable chunks such that, (I) whenever $M_A[o, a] = M_B[o, b] = true$, the indices a, o, and b appear together in some stable chunk, and (II) each stable chunk is as large as possible. Observe that, under these conditions, the given matchings M_A and M_B uniquely determine the division of the inputs into an alternating sequence of stable and unstable chunks. Figure 2 gives a concrete algorithm for computing these chunks from the matchings.

Lemma 3.1. For any matchings M_A between A and O and and M_B between B and O, the algorithm in Figure 2 outputs a `diff3` parse.

Proof. For (I), observe that the beginning of each unstable chunk, identified in step $2(a)$, is an index ℓ_O+1 in O such that $M_A[\ell_O+1, \ell_A+1] = false$ or $M_B[\ell_O+1, \ell_B + 1] = false$. The chunk then spans the elements $O[\ell_O + 1], ..., O[o - 1]$ in O, where $o > \ell_O$ is the least index such that (i) there exist a, b with $M_A[o, a] = M_B[o, b] = true$, or (ii) $O[o-1]$ is the last element in O. Thus an unstable chunk can not contain an element in O that is matched in both M_A and M_B.

Now suppose property (II) is violated in some parse output by the algorithm. Consider the first stable chunk C that violates the maximality condition. The chunk (if any) that precedes C must be an unstable chunk or else C is not the first stable chunk to violate the maximality property. By (I), we know that no elements in the unstable chunk preceding C (if any) could have been included in C. Also, if C is output in step $2(b)$, it terminates at $A[\ell_A+i-1], O[\ell_O+i-1]$, and $B[\ell_B+i-1]$ where i satisfies the condition that either $M_A[\ell_O+i, \ell_A+i] = \mathit{false}$ or $M_B[\ell_O+i, \ell_B+i] = \mathit{false}$. Clearly, no more elements could be included in C. Similarly, if C is output in step 3, then none of A, O, or B can contain any elements that follow C. Thus C must be maximal—a contradiction. □

Finally, if $P = [H_1, \ldots, H_n]$ is a parse—a sequence of chunks—then the *output* of P is obtained by concatenating the outputs for each chunk,

$$\mathit{out}(P) \quad = \quad (\mathit{concat}([A_1..A_n]),\ \mathit{concat}([O_1..O_n]),\ \mathit{concat}([B_1..B_n])),$$

where $\mathit{out}(H_i) = (A_i, O_i, B_i)$ for each $1 \leq i \leq n$.

4 Properties of `Diff3`

We now explore a number of intuitive properties that one might expect a synchronization algorithm such as `diff3` to possess... and encounter some surprises.

4.1 Locality

Users of version control systems such as CVS can often be heard saying things like "I'll change this section of the file and you change that one and we'll sync up when we're done," in perfect confidence that this synchronization will be unproblematic. Indeed, perhaps the most important property that users of `diff3` expect in practice is that, if A and B have been changed only in "non-overlapping ways," then synchronization will produce a unique, conflict-free result.

To investigate this intuition, let us focus on the case where A makes changes only at one end of the file while B makes changes only at the other end of the file. Define a *tiling* τ for a list O to be a partition of O into three lists O_1, O_2, and O_3 such that $O = O_1O_2O_3$. A configuration $(A \leftarrow O \rightarrow B)$ is *τ-respecting* if O_1 and O_3 are each modified in at most one of A and B and O_2 is modified in neither. If only one of O_1 or O_3 gets modified at all or if both O_1 and O_3 are modified in the same list, the result will obviously be conflict free. The interesting case is when both A and B make changes.

Next, we need to formalize the intuitive condition of the edited regions being "well separated." Two possible ways of doing this come immediately to mind:

– require that the edited regions be separated by a *large* untouched region—i.e., that O_2 be longer than any of A_1, O_1, O_3, or B_3; or
– require that the separating region be *different* from anything appearing anywhere else—i.e., that the string O_2 not occur in O_1, A_1, O_3, or B_3.

A	$1, 2, (1, 2)^{n-1}$	$1, 2, 1, 2$
O	$(1, 2)^n$	$1, 2$
B	$(1, 2)^n$	3
	stable	conflict

Fig. 3. Counter-example for locality

Most users of `diff3` would probably guess (as we did) that either of these conditions is enough to guarantee a conflict-free synchronization. As the following example shows, this guess is wrong on both counts.

Let $O_1 = \emptyset, O_2 = (1, 2)^n$, and $O_3 = 1, 2$, for some positive integer n. In replica A, the O_1 component is modified to $A_1 = 1, 2$ while in the replica B, the O_3 component is modified to $B_3 = 3$. Consider the maximum matching M_A for pair (O, A) where the $1, 2$ term in A_1 is matched to the first $1, 2$ term in O_2 component of O. Then the $(1, 2)^{n-1}$ prefix in the O_2 component in A is matched to the $(1, 2)^{n-1}$ suffix in the O_2 component of O. Finally, the last $(1, 2)$ term in the O_2 component of A is matched to the O_3 component of O. For the pair (O, B), the only maximum matching is one where their O_2 components are matched. As shown in Fig. 3, we have a ("true") conflict in this run. Note that the conflict is independent of the value of the parameter n and that it occurs even when the stable region O_2 is arbitrarily large.

At this point, one might begin to wonder whether, despite all the anecdotal evidence to the contrary, `diff3` might not be safe to use under *any* set of conditions that can be concisely characterized. Fortunately, this is too pessimistic. We can get the property we want by strengthening the second intuition.

Call a τ-respecting configuration $(A \leftarrow O \rightarrow B)$ *safe* if the O_2 component contains an element x that occurs exactly once in each of O, A, and B. Notice that there are no constraints on the length of O_2: it may contain just x.

Theorem 4.1.1. Every safe τ-respecting configuration $(A \leftarrow O \rightarrow B)$ leads to a unique conflict-free synchronization.

Such configurations are common in practice: for example, if the structures being synchronized are replicas of a source code file, it is reasonable to expect that O_2 will contain some completely unique line, such as a procedure header or a distinctive comment. The theorem can thus be viewed as justifying the common belief in `diff3`'s locality. Its proof rests on a technical property.

Lemma 4.1.2. Suppose we are given a configuration $(A \leftarrow O \rightarrow B)$, a matching M_A between O and A, and a matching M_B between O and B. If there exists an element z that occurs uniquely in each of A, O, B and if both M_A and M_B match the element z, then z must be contained in a stable chunk in the `diff3` parse that results from M_A and M_B.

Proof. Let α_O, α_A, and α_B respectively denote the locations of the element z in O, A, and B. We prove the property by iteratively considering the chunks that are output by the `diff3` algorithm until the point that element z appears in some output chunk for the first time. Let ℓ_O, ℓ_A, and ℓ_B (see Figure 2) be the indices denoting the locations of the last elements in O, A, and B that were processed by the algorithm. By assumption, $\ell_O < \alpha_O, \ell_A < \alpha_A$, and $\ell_B < \alpha_B$.

If the next chunk being output is an unstable chunk as in step 2(a), then the chunk ends just before the least offset in O at which there exists an element matched in both M_A and M_B. Clearly, the updated indices ℓ_O, ℓ_A, and ℓ_B must again satisfy the property $\ell_O < \alpha_O, \ell_A < \alpha_A$, and $\ell_B < \alpha_B$ since $M_A[\alpha_O, \alpha_A] = M_B[\alpha_o, \alpha_B] = \mathit{true}$. On the other hand, if the next chunk being output is a stable chunk as in step 2(b), then the chunk ends just before the least offset at which there exists an element in O that is not matched in at least one of M_A or M_B. If the updated indices still satisfy $\ell_O < \alpha_O, \ell_A < \alpha_A$, and $\ell_B < \alpha_B$, then we continue with the iterative process, maintaining the invariant. Otherwise, the element z must appear in this stable chunk, establishing the desired property.

Proof of 4.1.1. Assume wlog that O_1 is modified to A_1 in A (i.e., $A = A_1 O_2 O_3$) and that O_3 is modified to B_3 in B (i.e., $B = O_1 O_2 B_3$). Consider any maximum matching M_A between O and A. We claim that the element x must be matched in M_A. Suppose not. Let ℓ denote the number of elements that are matched by M_A between the A_1 component of A and O_1 component of O. Since the element x is not matched in M_A, the total number of elements matched by M_A is bounded by $\ell + (|O_2| + |O_3| - 1)$. Now consider the matching M'_A that agrees with M_A in the matching of elements between A_1 and O_1 and also completely matches the O_2 and O_3 components of A and O. Then the total number of elements matched by M'_A is $\ell + (|O_2| + |O_3|)$, contradicting the assumption that M_A is a maximum matching. Thus x must be matched in M_A. Moreover, since A and O are identical after x, M_A must match all elements in A after x to all elements in O after x, in order to be a maximum matching. Similarly, M_B must match all the elements up to x in B to all the elements up to x in O.

By Lemma 4.1.2, x must be contained in a stable chunk in `diff3`'s output. To complete the proof, consider any *un*stable chunk H output by the algorithm. Since the unique element x is contained in a stable chunk, either all elements in the A, O, and B components of chunk H precede x or they all follow x. In the former case, H must only be "changed in A," since M_B matches all elements up to x in B to all elements up to x in O. Similarly, in the latter case, H must be "changed in B." Thus, every unstable chunk is conflict free.

Finally, to see that the resulting output is unique, note that, in every parse, all the chunks above x are either stable or changed in A and those below x are stable or changed in B. Thus, in the output, the elements up to x will be taken from A while the elements following x will come from B. □

This well-separation condition is quite delicate, and we have found it difficult to generalize. For example, one might guess that it can be extended to situations where each user has made edits in multiple regions of the list, provided that these regions are separated by unique elements and no region is edited in both A and B. More precisely, let us say that a *generalized tiling* τ is a partition of O in

A	1	*2*	4		6		8
O	1	*2,3*	4	*5,5,5*	6	*7*	8
B	1		4	*5,5,5*	6	*2,3,4*	8
	stable	conflict	stable	changed in A	stable	conflict	stable

A	1		2		4	*6*	8
O	1		2	3	4	*6, 7*	8
B	1	*4,6*	2	*3*	4		8
	stable	changed in B	stable	changed in A	stable	conflict	stable

Fig. 4. Counter-example to idempotence

to $2k+1$ non-empty pieces for some positive integer $k \geq 1$, say, $O_1, O_2, ..., O_{2k+1}$. We now say a configuration $(A \leftarrow O \rightarrow B)$ is *τ-respecting* if each piece O_{2i+1} for $0 \leq i \leq k$ is modified in at most one of A and B, while each piece O_{2i} for $0 \leq i \leq k$ is modified in neither. A τ-respecting configuration $(A \leftarrow O \rightarrow B)$ is said to be *safe* if each O_{2i} component contains an element x_{2i} that occurs exactly once in each of O, A, and B.

But this generalization no longer ensures a conflict-free synchronization. For example, consider the extension even to $k = 2$; so $O = O_1O_2O_3O_4O_5$. Furthermore, assume that for any $1 \leq i < j \leq 5$, O_i and O_j are disjoint, that is, they do not share any elements. Let $A = A_1O_2O_3O_4A_5$, and let $B = O_1O_2B_3O_4O_5$. Also, let $A_1 = O_5$, and $A_5 = B_3 = \emptyset$. Now if $|O_5| > |O|/2$, then the unique maximum matching M_A between A and O matches the A_1 component in A to O_5 in O. On the other hand, consider the maximum matching M_B between B and O that matches them in all components except B_3 to O_3. It is easy to see that the first `diff3` chunk will be a conflict.

4.2 Idempotence

In the rest of this section, we consider some other intuitive properties that users might expect of `diff3` and show that, in fact, it possesses none of them.

To begin, let us take the intuition that every run of a synchronizer should "do as much as possible" and reach a stable state: synchronizing again immediately should propagate no further changes. This can be stated formally as follows:

Property 4.2.1. A synchronization algorithm is *idempotent* if $(A \leftarrow O \rightarrow B) \Rightarrow (A' \leftarrow O' \rightarrow B')$ implies $(A' \leftarrow O' \rightarrow B') \Rightarrow (A' \leftarrow O' \rightarrow B')$.

Fact 4.2.2. `Diff3` is *not* idempotent.

Counterexample. Consider the run in the top part of Figure 4, where

$$\begin{aligned} &([1,2,4,6,8] \leftarrow [1,2,3,4,5,5,5,6,7,8] \rightarrow [1,4,5,5,5,6,2,3,4,8]) \\ \Rightarrow\ &([1,2,4,6,8] \leftarrow [1,2,3,4,6,7,8] \rightarrow [1,4,6,2,3,4,8]). \end{aligned}$$

The output configuration can take another step, shown in the bottom part of Figure 4, leading to

$$\begin{aligned}&([1,2,4,6,8] \leftarrow [1,2,3,4,6,7,8] \rightarrow [1,4,6,2,3,4,8])\\ \Rightarrow &([1,4,6,2,4,6,8] \leftarrow [1,4,6,2,4,6,7,8] \rightarrow [1,4,6,2,4,8]).\end{aligned}$$

Note that diff3 has no choice in either case: each of the input configurations has just one pair of maximum matchings. (Ensuring this is the role of the blocks of repeated 5s in the first configuration.) □

4.3 Near Success on Similar Replicas

The diff3 algorithm begins by comparing O, separately, with A and with B; it never compares A and B directly. Nevertheless, it seems reasonable to expect that, even if A and B are very different from O, we should still be able to synchronize successfully, as long as A and B themselves are similar. Unfortunately, this intuition is misleading.

For any pair of replicas A, B, let $m(A,B)$ denote the length of a largest common subsequence for A and B. Let ϵ be some function mapping natural numbers to reals between 0 and 1. A pair of replicas A, B is said to be *ϵ-close* if $m(A,B) \geq (1-\epsilon(n))n$, where $n = \max\{|A|, |B|\}$. We can now formally define stability properties involving the notion of "similarity."

Property 4.3.1. A synchronization algorithm guarantees *near success on similar replicas* if there exists a universal constant $c > 0$ such that, for any ϵ-close pair (A, B), if $(A \leftarrow O \rightarrow B) \Rightarrow (A' \leftarrow O' \rightarrow B')$, then A' and B' are $(c\epsilon)$-close.

Fact 4.3.2. Diff3 does *not* guarantee near success on similar replicas.

Counterexample. Consider the input configuration

$$(A \leftarrow O \rightarrow B) = \begin{pmatrix} [1, \frac{n}{2}+1, \ldots, n-1, 2, \ldots, \frac{n}{2}, n] \\ \uparrow \\ [1, \ldots, n] \\ \downarrow \\ [1, 2, \frac{n}{2}+1, \ldots, n-1, 3, \ldots, \frac{n}{2}, n] \end{pmatrix}$$

(generalizing the one we saw in Section 2). Note that the pair (A, B) is $\frac{1}{n}$-close, as their largest common subsequence is of length $n-1$. The unique maximum common subsequence of O and A is $[1, 2, \ldots, n/2, n]$; between O and B it is $[1, 2, n/2+1, \ldots, n-1, n]$. This leads to three stable diff3 chunks and two unstable chunks, as shown in Figure 5. Though the second of these is conflicting, the first is updated only in A; the output of this chunk thus propagates $[n/2 + 1, \ldots, n-1]$ to O and B , yielding the complete output

$$(A' \leftarrow O' \rightarrow B') = \begin{pmatrix} [1, \frac{n}{2}+1, \ldots, n-1, 2, \ldots, \frac{n}{2}, n] \\ \uparrow \\ [1, \frac{n}{2}+1, \ldots, n-1, 2, \ldots, n] \\ \downarrow \\ [1, \frac{n}{2}+1, \ldots, n-1, 2, \frac{n}{2}+1, \ldots, n-1, 3, \ldots, \frac{n}{2}, n] \end{pmatrix}.$$

A	1	$\frac{n}{2}+1, \ldots, n-1$	2	$3, \ldots, \frac{n}{2}$	n
O	1		2	$3, \ldots, n-1$	n
B	1		2	$\frac{n}{2}+1, \ldots, n-1, 3, \ldots, \frac{n}{2}$	n
	stable	changed in A	stable	conflict	stable

Fig. 5. Counter-example to several properties

In the final reconciled state, A' and B' are only about $\frac{1}{3}$-close ($m(A', B') = n$, while $\max\{|A'|, |B'|\}$ is about $\frac{3n}{2}$), and so no constant c exists such that they are $\frac{c}{n}$-close for every positive n. □

4.4 Stability

Another intuitively reasonable property is that any two runs whose inputs are similar should have similar outputs.

Property 4.4.1. A synchronization algorithm is *stable* if there exists a universal constant $c > 0$ such that, for any three pairs (O_1, O_2), (A_1, A_2), and (B_1, B_2), such that each pair is ϵ-close, if $(A_1 \leftarrow O_1 \rightarrow B_1) \Rightarrow (A'_1 \leftarrow O'_1 \rightarrow B'_1)$ and $(A_2 \leftarrow O_2 \rightarrow B_2) \Rightarrow (A'_2 \leftarrow O'_2 \rightarrow B'_2)$, then each pair of replicas (O'_1, O'_2), (A'_1, A'_2), and (B'_1, B'_2) is $c\epsilon$-close.

Fact 4.4.2. `Diff3` is *not* stable, even for non-conflicting runs.

Counterexample. Consider the runs

$$\begin{array}{l}([X,Y,X] \leftarrow [X,Y,0,Y,X] \rightarrow [Y,X,0,Y]) \Rightarrow [Y,X,0] \\ ([X,Y,X] \leftarrow [X,Y,0,Y,X] \rightarrow [0,Y,X,Y]) \Rightarrow [0,X,Y],\end{array}$$

where $X = [1, \ldots, \frac{n}{2}]$ and $Y = [\frac{n}{2}+1, \ldots, n]$. It is easy to see that the corresponding pairs in the two input configurations are all $\frac{2}{3n}$-close while the output is only about $\frac{1}{2}$-close. □

5 Future Work

Our formalization suggests a number of interesting variations on `diff3`. For example, instead of asking for separate matchings of (O, A) and (O, B) could we try to compute a maximum *joint* matching of (A, O, B)? (Note that having maximum matchings for (O, A) and (O, B) does not imply having a maximum matching of (A, O, B). For instance, if $O = [1, 2, 3, 4, 5, 6]$, $B = [4, 5, 1, 2, 3]$, and $A = [4, 5, 6, 1, 2]$, the unique maximum matchings for the pairs leads to an empty match for the triple though clearly one can choose either $[1, 2]$ or $[4, 5]$ as the matching elements.) Alternatively, the choice of two-way matchings could be biased by their effect on the output, especially when deciding between two similar choices, since there are instances when a choosing a different maximum match or even a slightly sub-optimal matching can lead to better results.

Acknowledgments

We gratefully acknowledge stimulating discussions about list synchronization with James Leifer and Catuscia Palamidessi. Nate Foster helped us understand some of the intricacies of `diff3`'s behavior. This research has been supported by the National Science Foundation under grants 0113226, *Principles and Practice of Synchronization*, and 0429836, *Harmony: The Art of Reconciliation*.

References

1. Smith, R.: GNU diff3, Version 2.8.1, April 2002; distributed with GNU diffutils package (1988)
2. Lindholm, T.: A three-way merge for xml documents. In: DocEng 2004: Proceedings of the 2004 ACM symposium on Document engineering, pp. 1–10. ACM Press, New York (2004)
3. Chawathe, S.S., Rajamaran, A., Garcia-Molina, H., Widom, J.: Change detection in hierarchically structured information. ACM SIGMOD Record 25(2), 493–504 (1996)
4. Lanham, M., Kang, A., Hammer, J., Helal, A., Wilson, J.: Format-independent change detection and propagation in support of mobile computing. In: Brazilian Symposium on Databases (SBBD), Gramado, Brazil, pp. 27–41 (October 2002)
5. MacKenzie, D., Eggert, P., Stallman, R.: Comparing and Merging Files with GNU diff and patch. Network Theory Ltd. Printed version of GNU manual (2003)
6. Miller, W., Myers, E.W.: A file comparison program. Softw., Pract. Exper. 15(11), 1025–1040 (1985)
7. Myers, E.W.: An o(nd) difference algorithm and its variations. Algorithmica 1(2), 251–266 (1986)
8. Ukkonen, E.: Algorithms for approximate string matching. Information and Control 64(1-3), 100–118 (1985)
9. Foster, J.N., Greenwald, M.B., Kirkegaard, C., Pierce, B.C., Schmitt, A.: Exploiting schemas in data synchronization. Journal of Computer and System Sciences (2007) To appear. Extended abstract in Database Programming Languages (DBPL) (2005)
10. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Software Eng. 28(5), 449–462 (2002)
11. Stallman, R., et al.: Comparing and merging files, Manual for GNU diffutils (2002), available at `www.gnu.org`