Edit Distance with Single-Symbol Combinations and Splits

Manolis Christodoulakis[1] and Gerhard Brey[2]

[1] School of Computing & Technology, University of East London
Docklands Campus, 4–6 University Way, London E16 2RD, UK
m.christodoulakis@uel.ac.uk

[2] Centre for Computing in the Humanities, King's College London
26–29 Drury Lane, London WC2B 5RL, UK
gerhard.brey@kcl.ac.uk

Abstract. In this article we introduce new variants of the *edit distance* string similarity measure, where apart from the traditional insertion, deletion and substitution operations, two new operations are supported. The first one is called a *combination* and it allows two or more symbols from one string, to be "matched" against one symbol of the other. The dual of a combination, is the operation of a *split*, where one symbol from the first string is broken down into a sequence of two or more other symbols, that can then be matched against an equal number of symbols from the second string. The notions of combining and splitting symbols can be defined in a variety of ways, depending on how the application in hand defines similarity. Here we introduce three different possible definitions, and we provide an algorithm that deals with one of them. Our algorithm requires $O(L)$ time for preprocessing, and $O(mnk)$ time for computing the edit distance, where L is the total length of all the valid combinations/splits, and k is an upper bound on the number of valid splits of any single symbol.

Keywords: edit distance, combination, split, OCR

1 Introduction

One of the fundamental problems in string algorithmics has been, for more than 40 years now, the pattern matching problem, where exact copies of a given string, the *pattern*, need to be identified within a normally much larger string, the *text*. However, the need to relax the "exactness" of the pattern matching process, very soon became obvious. An endless list of applications benefit from *approximate*, rather than exact, pattern matching algorithms, including text processors, spell checking applications, information retrieval and bioinformatics.

Numerous approaches have been used to incorporate "inexactness" in pattern-matching (see for example [4], [2, Ch.12] or [7, Ch.10]), one of the most commonly used being the *edit distance* metric (also known as *Levenshtein* distance, as a credit to Vladimir Levenshtein who first mentioned it [3]). The edit distance between two strings, is simply defined as the minimum number of *edit operations* (substitutions, insertions, deletions) that are required to transform one string to the other.

In this paper, we introduce a new edit operation, called a *combination*, and its dual, called a *split*. These operations, in contrast to the traditional ones, apply to *sequences* of input symbols, rather than single symbols. The *combination* operation is that of combining two or more symbols from one string, and considering this combination to be equal to a single symbol from a second string. Equivalently, the single symbol of the second string can be *split* into the combination of symbols from the

Manolis Christodoulakis, Gerhard Brey: Edit Distance with Single-Symbol Combinations and Splits, pp. 208–217.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1

first. Defining which combinations match to what symbols, and vice versa, depends on the application. In all examples in this paper, we are referring to combinations that *look similar* to some symbol, but of course one can define any list of combinations that are meaningful for their application. For example, the sequence of symbols "rn" can be combined into the symbol "m", or "m" can be split to "rn", since the two look similar to each other.

The motivation for this new variant of the edit distance metric, comes from the approximate pattern matching problem on texts which originate from old documents that have been scanned and processed with Optical Character Recognition (OCR) Algorithms. In particular, we have been working on the Nineteenth-Century Serials Edition (NCSE) [5], which is a digital edition of six nineteenth-century newspaper and periodical titles. The corpus consists of about 100,000 pages that were micro-filmed, scanned in and processed using OCR software.

The quality of some of the text resulting from the OCR process varies from barely readable to illegible. This reflects the poor print quality of the original paper copies of the publications. An exact search for a pattern in the scanned and processed text would retrieve only a small number of matches, but ignore incorrectly spelled or distorted variations; on the other hand, an approximate search using general edit distance would yield too many false matches, since it cannot distinguish between "random" errors and errors that come from misinterpreted combinations of symbols which are common in OCR texts. For example, a general edit distance search for the name "Billington" in the corpus, would fail to distinguish between the approximate matches "Wellington" and "Billmgton", both of which have edit distance 2 from the pattern, but where in reality only the latter is a true match misinterpreted by the OCR software.

The paper is organised as follows. In Section 2 we describe the notation used throughout the paper, and formally define the notions of combinations and splits. In Section 3 we describe the preprocessing part of our algorithm and in Section 4 the main algorithm for computing the edit distance with combinations and splits. In Section 5 two variants of the problem we tackle here are presented. Finally, Section 6 contains our concluding remarks.

2 Preliminaries

Consider strings $x = x[1] \cdots x[n]$ and $y = y[1] \cdots y[m]$ over an alphabet Σ; the edit distance between x and y is defined as the minimum number of *edit operations* (insertions, deletions or substitutions) to transform x to y, or vice versa [6,3]. Implicitly, the simple edit distance assigns to each operation a unit cost, and computes the minimum cost of transforming x to y. A generalised version of the edit distance, is one that allows the different operations to have different costs; let d_{sub} be the cost for one substitution operation, and d_{indel} that of one insertion or deletion operation (one is a dual of the other, hence the identical cost). The generalised edit distance is defined then as the minimum cost of transforming x to y.

Notice how traditional variants of the edit distance always compare a single symbol from one string with either a single symbol from the other (e.g. $x[i]$ against $y[j]$) or a single symbol from one string with the empty string (e.g. $x[i]$ deletion or $y[j]$ insertion). In the variant of the edit distance that we introduce in this paper, we allow more than one symbol to be "matched" either against a single symbol (that is, many symbols are *combined* into one) or against a different combination of symbols (called

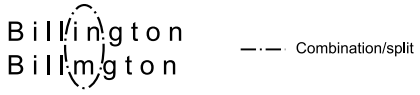

Figure 1. Example of a single-symbol combination

a *recombination*). For example, the symbol "m" is a combination of the symbols contained in the string "in" or "rri", and "b" is a combination of "lo". As seen in this example, there may very well exist, and they normally do, more than one combinations for the same symbol (symbol "m" in this example). Formally, we define combinations in the following way:

Definition 1. *Given a string $x = x[1] \cdots x[n]$ and a symbol α, α is called a* single-symbol combination*, or simply a* combination*, of x (equivalently, x is called a* split *of α), if and only if x is a valid match for α; we write $\alpha \mapsto x$.*

Obviously, any algorithm that makes use of combinations of symbols must be able to differentiate between meaningful (valid) and random combinations. In our case, meaningful ones are those combinations of symbols that *optically* resemble one or more other symbols. It is worth noting however that any kind of valid combinations may as well be used. For instance, one may consider combinations of symbols which *sound* similar to other (combinations of) symbols.

We assume that the list of valid combinations is given in the following way: for every symbol, α, for which valid combinations exist, a *combination list* $\mathcal{C}_\alpha$ is provided such that

$$\mathcal{C}_\alpha = \{x \in \Sigma^* | \alpha \mapsto x\}$$

We further introduce the following notation

$$k_\alpha = |\mathcal{C}_\alpha| \tag{1}$$

$$l_\alpha = \sum_{x \in \mathcal{C}_\alpha} |x| \tag{2}$$

$$L = \sum_{\alpha \in \Sigma} l_\alpha \tag{3}$$

Simply, k_α denotes the number of keywords (valid combinations) in $\mathcal{C}_\alpha$, l_α is sum of the lengths of all the strings in $\mathcal{C}_\alpha$, and L is the overall sum of lengths of all the strings over all combination lists.

With these definitions in place, the problem we are going to tackle in this paper is defined as follows:

Definition 2 (Edit Distance with Single-Symbol Combinations (EDSSC)). *Given strings $x = x[1] \cdots x[n]$ and $y = y[1] \cdots y[m]$, values d_{sub}, d_{indel} and d_{comb}, and lists $\mathcal{C}_\alpha$ for $\alpha \in \Sigma$, the* edit distance with single-symbol combinations *problem is that of finding the minimum cost of transforming x to y (equivalently, y to x) allowing substitutions, insertions or deletions, and single-symbol combinations or splits.*

An example of this problem is illustrated in Figure 1. The substring "in" of the first string is combined and matched against the symbol "m" of the second string. The EDSSC is therefore d_{comb}, since one combination operation is required to transform one string to the other. Normally, the cost for a combination/split will be smaller

Algorithm 6 Function NEXT for moving inside a tree T_α

```
1: function NEXT(v, a)
2:     while g(v, a) ="fail" do
3:         v ← f(v)
4:     return g(v, a)
```

(possibly zero) than that of a substitution or insertion/deletion; thus, the cost d_{comb} is going to be much smaller —reflecting the fact that the two strings look similar to each other— than the $d_{indel} + d_{sub}$ of the simple edit distance for the same pair of strings.

3 Preprocessing

In this stage we preprocess the lists of valid combinations and splits, which are given as input; the purpose of preprocessing is to allow the main algorithm to run faster.

Recall that for every symbol α we are given a list, $\mathcal{C}_\alpha$, of combinations that match α. The preprocessing starts by building an Aho-Corasick automaton [1], T_α, from the strings contained in $\mathcal{C}_\alpha$, for every $\alpha \in \Sigma$. We will then slightly modify these T_α's to better suit our edit distance algorithm. First, let us briefly describe the Aho-Corasick automaton.

An Aho-Corasick automaton, T, is constructed from a set of keywords, $\mathcal{C}$, essentially by generating a trie of all the strings contained in $\mathcal{C}$, and computing functions g, f and *out* as described below. Let v be one node in T, a be a symbol, and $\mathcal{L}_v$ be the string spelled out on the path from the root to node v. Then:

- The "forward" (or goto) function $g(v, a)$ returns a node u in T if there exists an outgoing edge from v to u labeled with a, or returns "fail" if no such node u exists; exceptionally for the root node, $g(root, a) = root$ if in the trie there is no outgoing edge from *root* labeled with a.
- The "failure" function $f(v)$ returns a node u whose label, $\mathcal{L}_u$, corresponds to the longest proper suffix of $\mathcal{L}_v$ that occurs in T; if no such suffix exists, $f(v) = root$.
- The "output" function $out(v)$ returns all the keywords of C that are suffixes of $\mathcal{L}_v$.

The algorithm that searches a text, s, for any of the strings in $\mathcal{C}$, operates by repeatedly calling the function $g(v, a)$, where v is the current node at any stage and a is the symbol of s currently being processed; initially, v is the root of T and $a = s[1]$. If at any stage $g(v, a)$ returns "fail", then the failure function is used and the search continues from the failure link node, $g(f(v), a)$. To make the notation easier, we define function NEXT, shown in Algorithm 6, which for a node v and input symbol a, follows as many failure links as necessary (possibly zero) and then calls g once.

For the purposes of our algorithm we need to slightly modify function *out*. Instead of storing the actual keywords that match at a specific node, we only need to know the *lengths* of all these matching keywords. This information will be later used to compute the cost of performing combination operations. The modification is easily implemented during the construction of the automaton, without modifying the running time of the algorithm.

Figure 2 demonstrates an example of preprocessing the combination list of the symbol m, $\mathcal{C}_m = \{$iii,iin,in,ni,nn,rn,rri$\}$. Nodes are represented by circles, solid edges

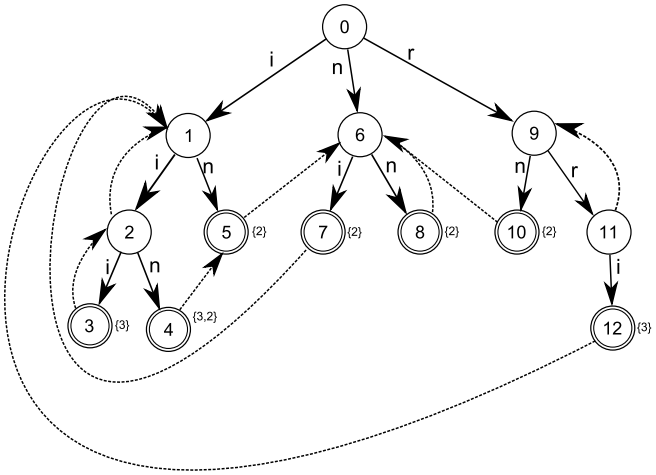

Figure 2. Preprocessing $\mathcal{C}_m = \{\text{iii,iin,in,ni,nn,rn,rri}\}$

represent the forward links (function g), and dashed edges represent the failure links (function f); for those nodes for which a failure link is not shown in the diagram, it is implied to be the root. Next to those nodes for which one or more keywords match, we show the lengths of the matching keywords. See for example, node 4 at which both keywords "iin" and "in" match, we store the lengths of these two keywords, 3 and 2 respectively.

4 EDSSC Algorithm

Let $x = x[1] \cdots x[n]$ and $y = y[1] \cdots y[m]$ be the two strings whose distance we want to compute; assume that the combination lists have already been provided and pre-processed, yielding Aho-Corasick automata T_α, for all $\alpha \in \Sigma$ for which valid combinations exist.

The algorithm works by processing gradually increasing prefixes of x and y. While processing prefixes $x[1..i]$ and $y[1..j]$, where $1 \leq i \leq n$ and $1 \leq j \leq m$, the permitted edit operations are as follows:

- substitution of a symbol $x[i]$ with the symbol $y[j]$, with cost d_{sub}
- insertion of the symbol $y[j]$ into x, with cost d_{indel}
- deletion of the symbol $x[i]$ from x, with cost d_{indel}
- combination of the symbols $x[\ell..i]$ (for some $1 \leq \ell < i$) to match $y[j]$, with cost d_{comb}
- split of the symbol $x[i]$ to match $y[h..j]$ (for some $1 \leq h < j$), with cost d_{comb}

The algorithm maintains a dynamic programming table $D(0..n, 0..m)$, where $D(i, j)$ represents the minimum cost of transforming $x[1..i]$ to $y[1..j]$, allowing the operations mentioned above. In contrast to the traditional edit distance dynamic programming algorithm, the whole D table must be maintained throughout the operation of the algorithm, rather than the last row (column) which were the only needed in the row-wise (column-wise) operation of traditional dynamic programming.

The base conditions for D are $D(i, 0) = i \cdot d_{indel}$ and $D(0, j) = j \cdot d_{indel}$, similarly to the simple edit distance algorithm. The correctness of these base conditions comes from the fact that the empty string ϵ is not a valid split for any $\alpha \in \Sigma$, and thus the

only way to transform $x[1..i]$ to ϵ is by deleting $x[1..i]$ from x and similarly the only way of transforming ϵ to $y[1..j]$ is by inserting $y[1..j]$ into x.

The recurrence relation for $D(i,j)$, for $i,j > 0$, is

$$D(i,j) = \mathbf{min} \begin{cases} D(i-1,j-1) + \textsc{sub}(x[i], y[j]) & \{\text{substitution}\} \\ D(i,j-1) + d_{indel} & \{\text{insertion}\} \\ D(i-1,j) + d_{indel} & \{\text{deletion}\} \\ \textsc{comb}(x[1..i], y[j]) & \{\text{combination}\} \\ \textsc{split}(x[i], y[1..j]) & \{\text{split}\} \end{cases}$$

where the functions SUB, COMB and SPLIT are defined as follows:

- SUB: this function simply compares $x[i]$ with $y[j]$ and returns 0 if they are equal and d_{sub} otherwise;
- COMB: finds the suffix of $x[1..i]$ that can be combined into symbol $y[j]$ with the minimum cost and returns this cost, or returns $+\infty$ if there is no such suffix;
- SPLIT: finds the suffix of $y[1..j]$ that can be combined into symbol $x[i]$ with the minimum cost and returns this cost, or returns $+\infty$ if there is no such suffix.

Next, we will demonstrate how functions COMB and SPLIT operate. We will assume a row-wise scan of the dynamic programming table D (a column-wise scanning is equivalent).

4.1 Combinations

Recall that, while computing $D(i,j)$, a combination operation translates to finding one suffix of $x[1..i]$ that can be combined to $y[j]$; here we are interested in the suffix with the minimum cost. This suffix can be found by searching the combination list of $y[j]$, $\mathcal{C}_{y[j]}$, and more specifically by making use of the Aho-Corasick automaton, $T_{y[j]}$, which was created during preprocessing. $T_{y[j]}$ is traversed, starting from the root, and descending down the tree, using failure links where necessary (functions NEXT, see Section 3), until the whole prefix $x[1..i]$ has been spelled out. Then the number of valid combinations is the number of elements in the set returned by function *out*, in the last node we visited in $T_{y[j]}$. If this set is empty, then there is no suffix of $x[1..i]$ that is a valid split of $y[j]$ and thus the algorithm returns $+\infty$ cost. On the other hand, if it has one or more elements, the algorithm must check the cost of each of those combinations and return the minimum.

Let v be the last node visited in T when parsing $x[1..i]$, and $out(v) = \{v_1, \ldots, v_r\}$ be the output of node v, where $v_1, \ldots, v_r$ are integers representing the lengths of the keywords that match suffixes of $x[1..i]$. Thus, $x[i - v_1 + 1..i], \ldots, x[i - v_r + 1..i]$ are all the valid splits of $y[j]$. What remains to be done is compute the cost of each of those.

If $x[i - v_1 + 1..i]$ is combined and matched to $y[j]$, then the prefix $x[1..i - v_1]$ of x and the prefix $y[1..j-1]$ of y must be aligned with each other. That is, the cost of this combination operation is the minimum cost of transforming $x[1..i - v_1]$ to $y[1..j-1]$, plus the cost of combining $x[i-v_1+1..i]$ into $y[j]$, i.e. $D(i-v_1, j-1)+d_{comb}$. We repeat the same process for all the values in $out(v)$ and return the minimum, and since the cost for every combination is constant, d_{comb}, the minimum cost for a combination at cell $D(i,j)$ is

$$\mathbf{min}\{D(i - v_1, j-1), \ldots, D(i - v_r, j-1)\} + d_{comb}$$

The algorithm for COMB, as it has been described so far, is not efficient: for every prefix $x[1..i]$ we spend $O(i)$ time to traverse $T_{y[j]}$ from the root until all i symbols of $x[1..i]$ have been processed. Let v be the last node visited in this traverse. We notice that, during the computation of $D(i+1, j)$, when the prefix $x[1..i+1]$ must be spelled out on $T_{y[j]}$, one can avoid parsing this whole prefix simply by remembering that $x[1..i]$ ended in node v. Then, the traversal for $x[1..i+1]$ requires only one call to function NEXT, $\text{NEXT}(v, x[i+1])$.

To take advantage of this observation, and given the assumption that D is processed in a row-wise manner, the algorithm must store the node where the suffix $x[1..i]$ ends in each $T_{y[j]}$ (for all $1 \leq j \leq m$). To do that we create a vector $t[1..m]$, which while working on the i-th row of D will contain values $t[j] = \text{NEXT}(root(T_{y[j]}), x[1..i])$, $1 \leq j \leq m$. Then, during processing the $(i+1)$-th row of D, $t[j]$ is updated with the value $t[j] = \text{NEXT}(t[j], x[i+1])$. The initial values of vector t —that correspond to the empty prefix of x— are of course $t[j] = root(T_{y[j]})$.

4.2 Splits

A split operation is the dual of a combination; a split on x is a combination on y and vice versa. During the computation of $D(i, j)$, a split means finding the suffix of $y[1..j]$ that is a valid split of $x[i]$ and is of minimal cost. Similarly to the combination operation, this can be found by spelling out $y[1..j]$ on the Aho-Corasick automaton of $x[i]$, $T_{x[i]}$.

Storing and recalling the last node visited by the prefix $y[1..j]$ on $T_{x[i]}$, works here too, only somewhat in a simpler way. The row-wise processing of D ensures that all prefixes of y are processed one after the other on $T_{x[i]}$, before moving to $T_{x[i+1]}$. Thus in this case, only a single variable u is required such that, during the computation of $D(i, j)$, $u = \text{NEXT}(root(T_{x[i]}), y[1..j])$, and then while computing $D(i, j+1)$, u is updated as $u = \text{NEXT}(u, y[j+1])$. The initial value of u, which corresponds to the empty prefix of y, is $u = root(T_{x[i]})$.

4.3 Analysis of the Algorithm

The complete algorithm for the "edit distance with single-symbol combinations" problem is shown in Algorithm 7. Notice that the first argument of COMB (SPLIT) is only a node in an Aho-Corasick automaton, the function does not need to know the whole prefix of x (y) that has already been processed.

Theorem 3. *Algorithm 7 requires $O(L)$ time for preprocessing, and $O(mnk)$ time for computing the edit distance, where L is the total length of all the valid combinations (see Eq. 3), and k is an upper bound on the number of valid splits of any single symbol.*

Proof. The preprocessing consists of building an Aho-Corasick automaton, T_α, for each list of valid combinations, $\mathcal{C}_\alpha$. This process requires time linear in the length of the input [1], that is $O(l_\alpha)$ (see Eq. 2). The updating of function *out* to return the lengths of the matching keywords, rather than the keywords themselves, does not increase the running time since during construction we can, in constant time per entry in $out(v)$, replace every keyword $x \in out(v)$ with its length, $|x|$. Collectively, the preprocessing of all the combination lists requires $O(\sum_{\alpha \in \Sigma} l_\alpha) = O(L)$ time.

In the edit distance algorithm, initialization (lines 10–14 of Algorithm 7) takes $O(m+n)$ time for assigning values to the first column of D, of size $O(n)$, the first row

Algorithm 7 EDSSC algorithm

```
1: function EDSSC(x, y)
2:     n ← |x|, m ← |y|
3:     for all α ∈ Σ do                {Preprocess combination lists}
4:         T_α ← AHO-CORASICK(C_α)
5:         for all v ∈ T_α do
6:             out' ← {}
7:             for all x ∈ out(v) do
8:                 out' ← out' ∪ {|x|}
9:             out ← out'
10:    for i ← 1 to n do               {Initializations}
11:        D(i, 0) ← i · d_indel
12:    for j ← 1 to m do
13:        D(0, j) ← j · d_indel
14:        t[j] ← root(T_y[j])
15:    for i ← 1 to n do               {Main algorithm}
16:        u ← root(T_x[i])
17:        for j ← 1 to m do
18:            D(i, j) ← min(D(i − 1, j − 1)+SUB(x[i], y[j]),
                           D(i, j − 1) + d_indel, D(i − 1, j) + d_indel,
                           COMB(t[j], x[i]), SPLIT(u, y[j]))
19:    return D(n, m)

20: function SUB(α, β)
21:    if α = β then
22:        return 0
23:    else
24:        return d_sub

25: function COMB(v, α)
26:    v ← NEXT(v, α)                  {Update v}
27:    min ← +∞
28:    for all r ∈ out(v) do
29:        if D(i − r, j − 1) + d_comb < min then
30:            min ← D(i − r, j − 1) + d_comb
31:    return min

32: function SUB(v, α)
33:    v ← NEXT(v, α)                  {Update v}
34:    min ← +∞
35:    for all r ∈ out(v) do
36:        if D(i − 1, j − r) + d_comb < min then
37:            min ← D(i − 1, j − r) + d_comb
38:    return min
```

of D, of size $O(m)$, and the vector, t, of size $O(m)$, which is initialized with pointers to the roots of $T_{y[j]}$. The main body of the algorithm (lines 15–18) computes the values of $D(i, j)$ for $1 \leq i \leq n$ and $1 \leq j \leq m$, by performing one of the operations substitution, insertion, deletion, combination, or split. The first three clearly require constant time, to examine cells $D(i-1, j-1)$, $D(i, j-1)$ and $D(i-1, j)$ respectively, while combinations and splits need deeper analysis.

A combination on cell $D(i, j)$, first makes a call to NEXT (line 26), which in turn makes zero or more calls to function f and a single call to function g (Algorithm 6). Although the number of calls to f is not constant, from [1] we know that collectively for a string of size n, there are $O(n)$ calls to both f and g. At the end of our algorithm, we have processed m columns, each of which can be seen collectively as a single parsing of string x on one Aho-Corasick automaton, i.e. $O(mn)$ time; similarly each of the n rows can be seen collectively as a single parsing of string y on one Aho-Corasick automaton, i.e. $O(mn)$ time. Therefore, overall, the total time spent on NEXT is $O(mn)$.

After calling NEXT, the function COMB computes the cost of each combination suggested by $out(v)$, in time constant per entry or $|out(v)|$ in total, as can be seen in lines 28–30 of Algorithm 7. Unfortunately, in the worst case the number of elements in $out(v)$ can be as large as the number of keywords, k_α, in the Aho-Corasick automaton. Let k be an upper bound of k_α, that is $k_\alpha \leq k$ for all $\alpha \in \Sigma$. Then, for every call to COMB we spend $O(k)$ time to examine all the valid combinations. Thus, the total time for computing the whole table D is $O(mnk)$.

It is worth noting that, despite the worst-case running time being $O(mnk)$, in practice the algorithm is expected to run in time closer to $O(mn)$, since the number of matching strings, $out(v)$, for every node v, is rarely larger than two or three, and most often it is either zero or one (see for example Figure 2).

5 Variants of the EDSSC Problem

In this section we present two possible variants of the EDSSC problem, which further extend the notions of combining and splitting symbols. Both can be seen as useful generalisations of EDSSC, but unfortunately the algorithm we presented in this paper cannot (at least not trivially) extend to solve them, and we leave these as open problems for further investigation.

Similar to the way we defined the combination lists, $\mathcal{C}_\alpha$, for symbols $\alpha \in \Sigma$, we could also define *recombination* lists, $\mathcal{C}_x$, $x \in \Sigma^*$; that is, lists of combinations of symbols that validly represent a different combination of symbols. For example, $\mathcal{C}_{bl} = \{lol, ld\}$. In this way, when computing the edit distance between strings x and y, we allow whole substrings of x (rather than single symbols) to be matched against different substrings of y, and vice versa. Therefore, a more general version of the EDSSC problem is that of allowing the operation of symbol recombinations, in parallel to all operations allowed in the EDSSC problem.

Definition 4 (Edit Distance with Re-Combinations (EDRC)). *Given strings $x = x[1] \cdots x[n]$ and $y = y[1] \cdots y[m]$, values d_{sub}, d_{indel} and d_{comb}, and lists $\mathcal{C}_x$ for $x \in \Sigma^*$, the* edit distance with recombinations *problem is that of finding the minimum cost of transforming x to y (equivalently, y to x) allowing substitutions, insertions or deletions, single-symbol combinations or splits, and string recombination operations.*

An even more general variant of the edit distance with (re-)combinations problem, is that of allowing transitive (re-)combinations of symbols. We illustrate this problem with an example. Consider strings x ="m" and y ="rri", and combination lists $\mathcal{C}_m = \{rn, in\}$ and $\mathcal{C}_n = \{ri\}$; although the two strings look similar there is no explicit valid combination "rri" in $\mathcal{C}_m$. However, notice that the suffix "ri" of y is a valid combination in $\mathcal{C}_n$, and thus y could be matched to y'="rn"; now $x \mapsto y'$ since "rn" $\in \mathcal{C}_m$, and thus we can infer that $x \mapsto y$.

Definition 5 (Edit Distance with Transitive Combinations (EDTC)). *Given strings $x = x[1] \cdots x[n]$ and $y = y[1] \cdots y[m]$, values d_{sub}, d_{indel} and d_{comb}, and lists $\mathcal{C}_x$ for $x \in \Sigma^*$, the* edit distance with transitive combinations *problem is that of finding the minimum cost of transforming x to y (equivalently, y to x) allowing substitutions, insertions or deletions, single-symbol combinations or splits, and string recombination operations, where any of the (re-)combination operations can be transitive:*

$$x \mapsto y \text{ and } y \mapsto z \quad : \quad x \mapsto z \qquad \text{where } x, y, z \in \Sigma^*$$

6 Conclusions

In this paper we have introduced the problem of edit distance with single-symbol combinations and splits, where in addition to the traditional edit distance operations, consecutive symbols in one string may be combined and matched against one symbol from the other. Our algorithm runs in $O(mnk)$ time with a prior $O(L)$ time for preprocessing, where L is the sum of lengths of all the valid combinations and k is the maximum number of valid splits for any symbol.

We also defined two variants of the above problem which allow a) two or more consecutive symbols of one string to match a different sequence of two or more consecutive symbols in the other string (recombinations), and b) combinations of symbols to be constructed by combining known smaller valid combinations (transitive combinations). These two variants are equally, if not more, interesting problems, both from a practical/application point of view, as well as from an algorithmic point of view, and have been left as open problems. As it stands, the algorithm we presented here does not appear to extend towards solving either of these problems, and further research is required.

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: An aid to bibliographic search.* Communications of the ACM, 18(6) 1975, pp. 333–340.
2. M. Crochemore and W. Rytter: *Jewels of Stringology: Text Algorithms*, World Scientific, 2002.
3. V. I. Levenshtein: *Binary codes capable of correcting deletions, insertions and reversals.* Soviet Physics Doklady, 10 1966, pp. 707–710.
4. G. Navarro: *A guided tour to approximate string matching.* ACM Computing Surveys, 33(1) 2001, pp. 31–88.
5. *Nineteenth-Century Serials Edition (NCSE)*: http://www.ncse.ac.uk.
6. D. Sankoff and J. Kruskal, eds., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, 1983.
7. B. Smyth: *Computing Patterns in Strings*, Pearson Education, 2003.