Online Approximate Matching with Non-local Distances

Raphaël Clifford and Benjamin Sach

University of Bristol, Dept. of Computer Science, Bristol, BS8 1UB, UK
{clifford, sach}@cs.bris.ac.uk

Abstract. A black box method was recently given that solves the problem of online approximate matching for a class of problems whose distance functions can be classified as being local. A distance function is said to be local if for a pattern P of length m and any substring $T[i, i+m-1]$ of a text T, the distance between P and $T[i, i+m-1]$ is equal to $\Sigma_j \Delta(P[j], T[i+j-1])$, where Δ is any distance function between individual characters. We extend this line of work by showing how to tackle online approximate matching when the distance function is non-local. We give solutions which are applicable to a wide variety of matching problems including function and parameterised matching, swap matching, swap-mismatch, k-difference, k-difference with transpositions, overlap matching, edit distance/LCS, flipped bit, faulty bit and L_1 and L_2 rearrangement distances. The resulting unamortised online algorithms bound the worst case running time *per input character* to within a log factor of their comparable offline counterpart.

1 Introduction

A great deal of progress has been made in finding fast algorithms for a variety of important forms of approximate matching in the last few decades. The most common computational model in which these algorithms have been analysed assumes that the text and pattern are to be held in fast primary storage and that each query to the data has constant cost. However, increasingly it has become apparent that new applications such as those found in telecommunications or monitoring Internet traffic require a fresh approach. It may no longer be possible to store the entirety of the text and the worst case time per input character is often more important than the overall running time of any algorithm.

The model that we consider is a deterministic variant of data streaming where we assume we are given a pattern in advance and the text to which it is to be matched arrives one character at a time. The overall task is to report matches between the pattern and text as soon as they occur and to bound the worst case time *per input character*. Previous work

in this model showed how to convert offline algorithms for approximate pattern matching problems with simple distance functions into efficient online ones using a black box approach [8]. It is an important feature of both our approach and the previous work that the running time of the resulting algorithms is not amortised.

The main restriction for this black box solution was that the distance function defined by the approximate matching problem had to have the property of being *local*. A local distance function is defined to be one where the distance between a pattern P and a substring of the text T can be written as $\Sigma_j \Delta(P[j], T[i+j-1])$, where Δ is any distance function between individual characters in the input alphabet. In other words, the distance was simply measured as the sum of the distances between individual symbols. Although a number of interesting problems including exact matching with wildcards, matching under the Hamming norm and numerical measures such as the L_2 and L_1 norm have distance functions which are local, this left open the problem of how to handle the many matching problems with more sophisticated distance measures.

To appreciate the challenges that arise in online pattern matching when the distance function is non-local, consider for example the problem of function matching [3]. There is a function match between pattern P and $T[i, i+m-1]$ if there exists a function f (possibly distinct for each i) from the input alphabet Σ to itself such that $T[i+j] = f(P[j])$ for all $0 \leq j < m$. For example aba has a function match with $T = xyx$ but not $T' = xyy$ as a can not be mapped to two different letters. In the previous black box approach to creating online algorithms which we briefly describe in Section 2, distances would be found independently for different substrings of the pattern and the results combined. However, in this case whether $P[3] = a$ matches $T'[3] = y$ depends on the function chosen to map the characters in $P[1, 2]$ and vice versa. By definition only one choice of function is permitted for the whole of the pattern. As a result any matchings for the substrings of P would appear to have to depend on the results for all other substrings. In general for non-local distance functions, we must find a way efficiently to handle the dependencies between different parts of the pattern.

Our contribution is to present three general methods which can be applied successfully to convert a wide variety of non-local approximate matching problem into efficient unamortised online ones. We will refer to such algorithms as pseudo-realtime (PsR) throughout the paper by analogy to the realtime model for linear time algorithms. The techniques are necessarily no longer 'black box', depending in detail on the specific

offline algorithm being considered. As with the previous work for local distance functions, the running time per input character is guaranteed to be within a log factor of its offline counterpart.

2 Preliminaries and Previous Work

Throughout the paper, T and P will be used to denote the text and pattern strings respectively. By convention, $|T| = n$ and $|P| = m$. For any other strings, a lowercase letter is used to denote length, for example $|A| = a$. $A||B$ denotes the concatenation of strings A and B. The character alphabet is denoted Σ (and Σ_P for the pattern alphabet). When discussing the alignment of the pattern and text we will often refer to *right alignments*. Right alignment i of P and T aligns the final character of P with the ith character of T. This is a natural way to discuss alignments when the text is being streamed. The usual offline notion where the first character of the pattern is aligned at a position in the text will be termed left alignment to avoid confusion.

The ideas we present build on the black box algorithm of [8] for local distance functions which we briefly describe here. The basic idea is to split the pattern into $O(\log m)$ consecutive subpatterns each having half the length of the previous one. The first subpattern $S_1 = P[1, m/2]$ and subpattern S_j has length $m2^{-j}$ for $1 \leq j < s$ where $s = \log_2(m) + 1$. S_s is set to be the last character of the pattern. The offline algorithm is then run for each subpattern against the whole of the text with the distances found added to an auxiliary array C. In this way, for any subpattern starting at position j of the pattern, its distance to a substring starting at position i of the text will be added to the count at $C[i - j + 1]$. At the end of this step C will contain $\Delta(P, T[i, i + m - 1])$ for every location i in t.

To ensure that the work for each subpattern is completed before its result is needed to report a match, the text was partitioned into overlapping substrings. Each of the $O(\log m)$ subpatterns has a different length and induces a different and independent partitioning of the text. Each partition of the text is set to be of size to $3|S_j|/2$, with an overlap of length $|S_j|$. For each subpattern, the work of a search does not have to be completed until $|S_j|/2$ characters after it starts and so we can set this work to be performed over the period between arrival of $T[i]$ and $T[i + |S_j|/2]$ as shown in Figure 1. This gives a space requirement of $O(m)$. Let $T(n, m)$ be the time complexity of the offline algorithm used as a black box. The running

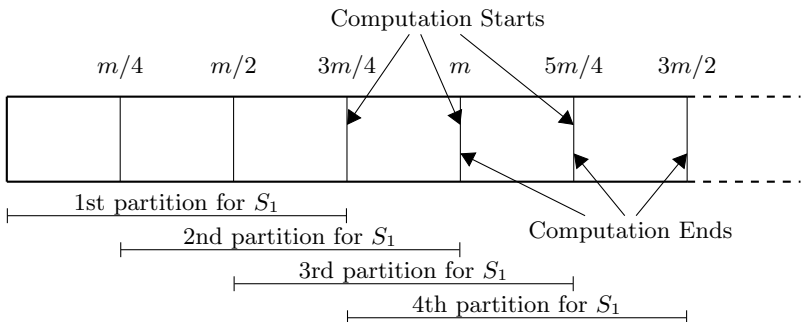

Fig. 1. Partitioning of the text for subpattern S_1.

time per text input character was shown to be $O(\Sigma_{j=1}^{\log_2 m} T(n, 2^{j-1})/n))$ which is bounded above by $O(T(n,m)\log(m)/n)$.

3 Our Results

We present an overview of problems that we solve in pseudo-realtime and the methods developed. Due to space restrictions we are not able to discuss each problem in detail, but to explain the main ideas we present examples for each of the main techniques.

- Some approximate matching problems with non-local distance functions translate immediately to the pseudo-realtime setting with no asymptotic time penalty and minimal modification. For example, edit distance and Longest Common Subsequence (LCS) where the offline algorithm completes the full dynamic programming table and faulty bits and L_1 rearrangement distance, where the offline solutions considers each alignment of the pattern and text separately are all naturally pseudo-realtime. As another example, the algorithm of Amir et al. [6] for the parameterised matching problem is a modification of KMP that allows a direct translation by applying the realtime modifications of Galil [9].
- Many pattern matching algorithms rely heavily on cross-correlations implemented via the fast Fourier transform (FFT). We show in Section 4.1 how these problems can be made pseudo-realtime efficiently by application of a PsR cross-correlation algorithm, using the L_2 rearrangement distance problem as an example.
- We next develop a method we call 'Split and Correct' in Section 4.2. The aim is to split the pattern into subpatterns as in Section 2 and

then correct for the non-local effects that occur across the boundaries between adjacent subpatterns. Pseudo-realtime swap-mismatch is given as an example.

– Finally we present a method we call 'Split and Feed' in Section 4.3. This method is applied to problems where matching a subpattern to the text can affect the alignment of other subpatterns with the text. We explain this method using the k-differences problem as an example. We also comment that the k-difference with transpositions problem can be solved by combining this technique with that of Split and Correct.

A brief summary of these results along with the multiplicative time penalty incurred for the corresponding online/PsR algorithms is given in Table 1. This list is intended to be exemplary rather than comprehensive and in particular we anticipate that our methods can be applied equally successfully to a larger range of problems we have yet to consider. In each case we require at most $O(m)$ space. Note that in the case of edit distance there also exists an $O(n(1+\frac{m}{\log n}))$ time offline solution for finite alphabets [12].

Problem	Offline per char time	Online/PsR penalty	**Method**
local matching	various	$O(\log m)$	Splitting [8]
function	various [3]	$O(\log m)$	PsR Cross-correlations
parameterised	$O(\log \lvert\Sigma\rvert)$ [6]	$O(1)$	Realtime KMP
edit distance/LCS	$O(m)$ [11]	$O(1)$	Immediate
k-differences	$O(k)$ [10]	$O(\log m)$	Split & Feed
swap-mismatch	$O(\sqrt{m \log m})$ [5]	$O(1)$	Split & Correct
swap	$O(\log m \log \lvert\Sigma\rvert)$ [4]	$O(\log m)$	Split & Correct
overlap	$O(\log n)$ [4]	$O(\log m)$	Split & Correct
k-diff with transpositions	$O(k)$ [10]	$O(\log m)$	Split & Correct + Feed
self normalised	$O(\log m)$ [7]	$O(\log m)$	PsR Cross-correlations
faulty bits	$O(m \log m)$ [2]	$O(1)$	Immediate
flipped bits	$O(\log m)$ [2]	$O(\log m)$	PsR Cross-correlations
L_1 rearrangement	$O(m)$ [1]	$O(1)$	Immediate
L_2 rearrangement	$O(\log m)$ [1]	$O(\log m)$	PsR Cross-correlations

Table 1. Summary of main pseudo-realtime pattern matching results.

4 Algorithms for Pseudo-realtime Translation

We now give an overview of each method along with examples of their application.

4.1 Pseudo-realtime Cross-correlation Method

The cross-correlation is an important technique in pattern matching and lies at the heart of many of the fastest algorithms known. We discuss a class of non-local problems that can be made pseudo-realtime simply and efficiently by using as its main tool the replacement of the offline cross-correlation with a pseudo-realtime version. Lemma 1 gives the running time per input character.

Lemma 1 (Pseudo-realtime Cross-correlation) *Let X be an array received online and Y an array received in advance. For any i, when character $X[i]$ arrives, we can compute $(X \otimes Y)[i-m+1] = \sum_{j=0}^{y-1} X[i+j-m+1]Y[j]$ in $O(\log^2 m)$ time. As the cross-correlation is local, this is immediate from application of the black box method of [8].*

For the function matching problem, Amir et al.[3] give a solution for small pattern alphabets in $O(n|\Sigma_P| \log m)$ and a randomised solution to the general problem that runs in $O(n \log m)$ time with failure probability $1/n$ of declaring a false positive. Both algorithms can be made pseudo-realtime in $O(|\Sigma_P| \log^2 m)$ and $O(\log^2 m)$ time per character respectively using PsR cross-correlations and by reordering the computation of the offline algorithms.

The L_2 rearrangement distance problem first introduced by Amir et al. [1] allows us to describe a slightly more sophisticated application of this general method. At right alignment i, consider all permutations π so that $T[i-m+j] = P[\pi(j)]$ for all j and define $cost(\pi) = \sum_j |j - \pi(j)|^2$. The L_2 rearrangement distance is defined to be the minimum *cost* over all such permutations (or ∞ if no such permutation exists). It is clear from the definition that this is a highly non-local problem. Analysis of the offline $O(n \log m)$ solution shows that the main challenge lies in its cross-correlation stage but that the remaining work still requires careful scheduling for the overall technique to be successful. We present a pseudo-realtime version of their algorithm which runs in $O(\log^2 m)$ time per character using $O(m)$ total space, equalling the space requirements for the offline solution.

For all $a \in \Sigma$, let $\psi_a(X)$ be an array of the indices of all occurrences of character a in X; further we define $occ_a(X) = |\psi_a(X)|$. Consider the following functions:

$$F_x(P', T')[i'] = \sum_{j=0}^{|P'|-1} (P'[j] - T'[i' + j - |P'|] + x)^2 \ . \tag{1}$$

$$G_x(P, T)[i] = \sum_{a \in \Sigma} F_x(\psi_a(P), \psi_a(T), a)[occ_a(T[1, i])] \ . \tag{2}$$

Amir et al. show that if we set $x = (i - m)$ then $G_{(i-m)}(P, T)[i]$ is exactly the distance between $T[i - m + 1, i]$ and P. This assumes that the distance is less than ∞ which can be checked in $O(\log m)$ time per character. Further, it is shown that if we can calculate $G_x(P, T)[i]$ for fixed $x = 0, 1, 2$ then $G_{(i-m)}(P, T)[i]$ can be computed by polynomial interpolation in constant time per character. Therefore, in the remainder we need only consider a fixed x.

Observe that the sums, $G_x(P, T)[i - 1]$ and $G_x(P, T)[i]$ differ only at the term where $a = T[i]$. Thus, if we can update the corresponding F_x when we receive $T[i]$ in pseudo-realtime, a sliding window approach will allow us to compute $G_x(P, T)[i]$. To compute the F_x terms in pseudo-realtime we split the data and computation by symbol. When a symbol $T[i] = a$ arrives we consider this as the arrival of a new index for the array $\psi_a(T)$. In this way we create one array of indices per character in the input alphabet and can consider each separately. It is important for the pseudo-realtime algorithm that when a symbol a arrives, the only work that is carried out relates to the array $\psi_a(T)$ and no others. The computation of F_x can therefore be computed independently for each symbol. The classification of the arriving character can be handled using a binary search tree in $O(\log m)$ time.

It remains to show how to compute $F_x(P', T')[i']$ efficiently in pseudo-realtime. By multiplying out F_x observe that it can be computed using PsR cross-correlations and a sliding window. Applying Lemma 1 the resulting pseudo-realtime algorithm runs in $O(\log^2(|P'|))$ time per character and $O(|P'|)$ space. However, $|P'| \leq m$ so $O(\log^2(|P'|)) \in O(\log^2 m)$ time per character and this dominates the overall time complexity. The total space is dominated by the working space of computing each F_x. For each a, $|P'| = occ_a(P)$ giving a total space of $\sum_a O(occ_a(P)) \in O(m)$. Theorem 1 summarises the result.

Theorem 1. *The L_2 rearrangement distance problem can be solved in pseudo-realtime in $O(\log^2 m)$ time per character and $O(m)$ space.*

4.2 Split and Correct

The 'Split and Correct' method we develop in this Section can most easily be applied to non-local pattern matching problems where the distance function between the pattern and substrings of the text can be expressed as the cost of a sequence of moves. In the pseudo-realtime setting, a non-local move is defined to be one which changes characters in more than one of the subpatterns in the split pattern. We consider in particular, problems where the number of possible non-local moves with respect to a given subpattern is bounded by a constant. For this class of problems, we split the pattern into subpatterns as before and create a set of transformed subpatterns by applying all valid combinations of non-local moves to each subpattern. Matches of all of these patterns with the text can be found with no effect on time complexity as the number of such moves is constant per subpattern. For each boundary between two adjacent subpatterns, we will now need to compute the number and type of non-local moves that would occur in a globally optimal alignment between pattern and text. This allows us to select the appropriate transformed subpatterns at each alignment and recombine the results.

To make the explanation concrete, we show how this general method can be applied to the Swap-Mismatch problem. The related *Swap* matching and *Overlap* matching problems, first addressed by Amir et al. [4] can also be solved by the method detailed above although in the latter case a slight generalisation of the notion of a move is required.

Swap-Mismatch. Swap-Mismatch distance between equal length strings is the minimum number of moves required to transform P into T (referred to as $cost(P, T)$). The valid moves are *swap* (swap two adjacent characters) and *mismatch* (replace a character). As overlapping swap and mismatch operations can always be replaced by two mismatches at no extra cost, the minimal cost transformation need never apply two moves to the same character. On non-equal length strings, at right alignment i, the distance is defined to be $cost(P, T[i - m + 1, i])$. The solution we present gives an $O(\sqrt{m \log m})$ time per character solution if applied to the best known offline method of Amir et al. [5].

Let l_j and r_j be the leftmost and rightmost indices of subpattern S_j respectively (split as before). Following the Split and Correct method, we define a set of 'boundary indicators' for all $0 < j < s$: $b_{ij} = 1$ if $P[r_j]$

and $P[l_{j+1}]$ are swapped in some minimal cost transformation of P into $T[i-m+1,i]$ and 0 otherwise. Trivially, we let $b_{i0} = b_{is} = 0$ for all i. The remainder of the section explains first how to use these indicators and secondly, how to compute them efficiently.

A black box solution using boundary indicators. For any subpattern S_j, the valid non-local moves are swaps at each end, giving a total of four transformed subpatterns. For $x, y \in \{0,1\}$, let $S_j^{(x)(y)} = P[l_j + x, r_j - y]$ represent these transformed subpatterns. We ignore the swapped characters at the boundaries at this stage as the costs incurred by them will be accounted for by the boundary indicators. Recall that in the black box method of [8] there is a final stage of accumulation of the distances found between subpatterns and the different substrings of the text into an auxiliary array C. To compute the swap-mismatch distance from $S_j^{(x)(y)}$ to T for all j and all x, y we apply this method to an offline swap-mismatch algorithm but modify this final stage. Having computed the distances for each $S_j^{(x)(y)}$, we use the boundary indicators to pick which $S_j^{(x)(y)}$ to include in the sum at each alignment and therefore to add to C. Lemma 2 shows that we are therefore able to calculate $cost(P, T[i-m+1,i])$ at each right alignment i with additive $O(\log m)$ time per alignment.

Lemma 2 *At right alignment i, the distance from P to T is equal to $\sum_{j=1}^{(s-1)} b_{ij}$ plus for each j, the distance from $S_j^{(b_{i(j-1)})(b_{ij})}$ to T at right alignment $i - m + r_j - b_{ij}$.*

Computing the boundary indicators. Lemma 3 allows us to find the boundaries across which swaps will occur in an optimal transformation. Both conditions can readily be checked in constant time per character. In the overall algorithm, we wish to compute boundary indicators for $O(\log m)$ different boundaries, requiring $O(\log m)$ time per text character. We define $y(xy)^*$ to be a "y" followed by zero or more copies of "xy".

Lemma 3 *If $n = m$, there is an optimal swap-mismatch transformation of P into T where a swap occurs across the boundary between $P[i] = x$ and $P[i+1] = y$ iff*

1. *$P[i] = T[i+1]$ and $P[i+1] = T[i]$*
2. *There exists an odd ℓ such that $T[i-\ell+1..i] = y(xy)^*$, $P[i-\ell+1..i] = x(yx)^*$ and $P[i-\ell] \neq y$ or $T[i-\ell] \neq y$*

Overall, the algorithm performs three steps, all in pseudo-realtime:

1. Calculate matches of T against $S_j^{00}, S_j^{01}, S_j^{10}$ and S_j^{11} for all j at all alignments. This is done using the black box method applied to the offline method of Amir et al. in $O(\sqrt{m \log m})$ time per character.
2. Calculate the boundary indicators at all alignments. This is computed using the method above in $O(\log m)$ time per character.
3. Combine the results of steps one and two using the relation stated in Lemma 2. This is computed directly and requires $O(\log m)$ time per character.

Theorem 2 gives the running time for pseudo-realtime swap-mismatch.

Theorem 2. *The swap-mismatch problem can be solved pseudo-realtime in $O(\sqrt{m \log m})$ time per character and $O(m)$ space.*

4.3 Split and Feed

The final technique that we discuss is termed 'Split and Feed'. Here we consider pattern matching problems where the non-local nature of the distance function affects the alignment of subpatterns. Where the distance function is local, the positions of alignments of all subpatterns is fixed for a given alignment of the whole pattern and text. However for problems where insertion and deletion are permitted, for example, this no longer holds and we can no longer apply the previously described Split and Correct method. Consider matching a pattern $P = A||B$ against some text T where A and B are substrings. Under such distance functions, optimal matches of P against T may be composed of sub-optimal matches of A and B. Edit distance and the k-differences problem have this property. Therefore, we cannot compute matches of P by separately computing matches of its sub-patterns.

As before the method splits the pattern, P, into sub-patterns, $P = S_1||S_2||S_3 \ldots ||S_s$. The overall idea of the method is to iteratively use the distances from $R_{j-1} = S_1||S_2|| \ldots ||S_{j-1}$ to T to compute the distances from $R_j = S_1||S_2|| \ldots ||S_{j-1}||S_j$ to T. [1] We refer to this process as 'feeding' the results from distances to R_{j-1} into the input of the computation of distances to R_j. The computation associated with R_j is termed level j. Note that level s computes distances against $R_s = P$ as required. This feeding of results ensures that optimal matches composed of sub-optimal sub-pattern matches are computed correctly. We motivate the Split and Feed method by considering a pseudo-realtime solution for the k-difference problem.

[1] where $R_1 = S_1$

k-differences. The edit distance between two strings is the minimum number of moves required to transform P into T. We refer to this distance as $cost(P, T)$. The valid moves are *insert* (insert a character), *delete* (delete a character) and *mismatch* (replace a character). In the pattern matching case, we define an array Cost: at right alignment i, the distance $\text{Cost}(P, T)[i] = \min_{\ell<i} \text{cost}(P, T[\ell, i])$. The k-difference problem is to output the distance at all positions i where $\text{Cost}(P, T)[i] \leq k$, we call this a match. We also refer to a match of P as shorthand for a substring of T that P can be transformed into in $\leq k$ moves. Observe that both *insert* and *delete* operations are non-local and affect alignment of other characters.

A straightforward approach to solving this problem would split the pattern into halving lengths and consider each separately. However, even if a solution for the previously mentioned problems were found, there is an added difficulty we have to consider. Subpatterns much shorter than k still require text partitions that are $\Theta(k)$, increasing the overall time complexity of the algorithm. As a result we insist that the smallest subpattern has size larger than k. However we are now required to carry out extra work in order to find the distances that include this final subpattern. We assume throughout that $k \leq m/8$. If this is not the case then the direct dynamic programming solution runs in $O(m) \in O(k)$ time per character without modification.

Our solution splits the pattern into subpatterns of halving length $S_1, S_2, S_3 \ldots S_s$ so that $P = S_1||S_2||S_3 \ldots ||S_s$. In this case s is selected to be largest integer so that $|S_s| \geq 4k$. The final subpattern, S_s, is therefore of size $4k \leq |S_s| < 8k$. As discussed in the method overview, computation occurs in levels where level j computes the distances from $R_j = R_{j-1}||S_j$ to T using the distances from $R_{j-1} = S_1||S_2|| \ldots ||S_{j-1}$ to T.

Level j can also be viewed as computing distances from S_j to T but incorporating a *starting cost array*. The starting cost array for level j gives the cost of beginning a match of S_j at each left alignment (remembering that the output is in terms of right alignments). We let the starting cost at left alignment i equal the distance from R_{j-1} to T at right alignment $(i-1)$. The distances computed from S_j to T using this starting cost array can be shown to equal the distances from R_j to T. Intuitively, this states that a match of S_j must be proceeded directly by a match of R_j.

A modification to the hybrid dynamic programming solution of [10] allows us to compute these distances offline in $O(|S_j|k)$ time if the portion of the text we are considering is of length $O(|S_j|)$. We will use this modified offline algorithm as a tool in our pseudo-realtime algorithm, dis-

tributing its work over the time taken for portions of the text to arrive. Details of the modification are left for the full version of the paper.

We can now describe the structure of the pseudo-realtime k-difference algorithm.

Feeding one level into another. For all levels $j < s$, we split T into partitions of length $7|S_j|/4$ with overlap $3|S_j|/2$. We will ensure that $\text{Cost}(R_j, T)[i]$ has been computed when $T[i + 5|S_j|/8]$ arrives. As $k < 4|S_j|$, if a match of $R_j = R_{j-1}||S_j$ ends in $T[y - |S_j|/2 + 1, y]$ for some partition ending at y then the corresponding match of R_{j-1} must end in $T[y-7|S_j|/4+1, y-3|S_j|/4]$. This splits the partition into two disjoint sections marked in Figure 2 as *matches start* and *matches end*. To compute distances from R_j in the *matches end* section using the modified hybrid algorithm we only need the distances from R_{j-1} in the *matches start* section. Computing R_j distances in the *matches end* section is sufficient as the text partitions overlap.

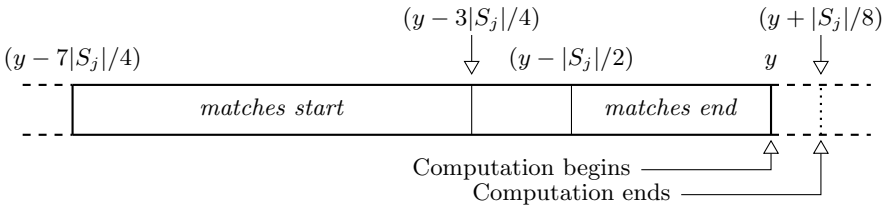

Fig. 2. The structure of a text partition for subpattern S_j ending at position y.

Using $\text{Cost}(R_{(j-1)}, T)[y - 7|S_j|/4 + 1, y - 3|S_j|/4]$ as the starting cost array, text partition, $T[y - 7|S_j|/4 + 1, y]$ and pattern S_j we can compute $\text{Cost}(R_j, T)[(y-|S_j|/2+1, y]$ in $O(|S_j|\, k)$ time (offline) using the modified hybrid method above. [2]

We begin computation as soon as the last text character of the partition is received. However, we distribute the work over the time allocated to the next $|S_j|/8$ character arrivals. As we only process one partition at a time, we use $O(k)$ time per text character (per level). This computation requires that level $j - 1$ has outputted $\text{Cost}(R_{(j-1)}, T)[y - 7|S_j|/4 + 1, y - 3|S_j|/4]$ before character y is received. In the worst case, a level j partition ending at y needs the output of the level $j - 1$ partition ending at $y - |S_j|/2$ as shown in Figure 3. This partition will finish computing

[2] For $j = 1$, we let $R_0 = \emptyset$ (the empty string) so that $\text{Cost}(R_0, T)[x] = 0$ for all x.

after $|S_{j-1}|/8 = |S_j|/4$ characters arrive which is before character y is received. As a result, computation of the relevant section of $\text{Cost}(T, R_{(j-1)})$ completes before it is needed by level j.

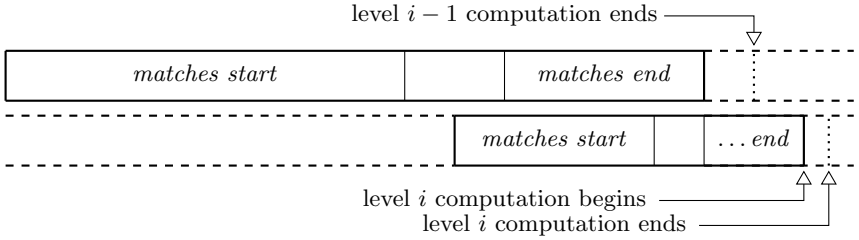

Fig. 3. Alignment of a level $i-1$ partition (above) and a level i partition (below).

The final level of computation. The aim of this stage is to produce the bottom $|S_s| + 1$ rows of the dynamic programming table for the edit distance between P and T so that we can output $\text{Cost}(P, T)[i]$ when $T[i]$ arrives. Level $(s-1)$ provides us with the top row of this table in the form of $\text{Cost}(R_{(s-1)}, T)$. If a match of P ends at right alignment i, in the worst case, the corresponding match of prefix $R_{(s-1)}$ ends at position $i - |S_s| - k \leq i - 5|S_s|/4$, for example where there are k inserts into S_s. Therefore, we only need $\text{Cost}(R_{(s-1)}, T)[x]$ for $x < i - 5|S_s|/4$ to compute $\text{Cost}(P, T)[i]$ which we have before $T[i]$ arrives from level $s-1$. Use of this fact, coupled with careful work scheduling allows us to fill the dynamic table a constant number of columns per text character so that column i is filled as $T[i]$ is seen. $\text{Cost}(P, T)[i]$ is then the value of the bottom cell of column i. Each column is of height $O(S_s) \in O(k)$ as $|S_s| \leq 8k$, so we perform $O(k)$ work per character as required.

Theorem 3. *The time complexity for the pseudo-realtime k-differences algorithm is $O(k \log m)$ per character. Each level requires $O(|S_j|)$ space, giving a total of $O(m)$ space.*

Combining Split and Feed and Split and Correct. The k-difference problem with transpositions allows an additional move, *transposition*; a restricted *swap* which can only occur before all other moves types. This problem can be solved offline in $O(kn)$ time by a simple modification of the method of Landau and Vishkin [10]. Although no single method we

have discussed will convert this algorithm to be pseudo-realtime, by applying the Split and Feed and Split and Correct methods simultaneously an $O(k \log m)$ time per character algorithm results.

Acknowledgements. The authors would like to thank Benny and Ely Porat for many helpful discussions at an early stage of the work on this paper and Dave Arthur for his comments on the final draft.

References

[1] Amihood Amir, Yonatan Aumann, Gary Benson, Avivit Levy, Ohad Lipsky, Ely Porat, Steven Skiena, and Uzi Vishne. Pattern matching with address errors: rearrangement distances. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 1221–1229, 2006.

[2] Amihood Amir, Yonatan Aumann, Oren Kapah, Avivit Levy, and Ely Porat. Approximate string matching with address bit errors. In *Symposium on Combinatorial Pattern Matching*, pages 118–129, 2008.

[3] Amihood Amir, Yonatan Aumann, Moshe Lewenstein, and Ely Porat. Function matching. *SIAM Journal on Computing*, 35(5):1007–1022, 2006.

[4] Amihood Amir, Richard Cole, Ramesh Hariharan, Moshe Lewenstein, and Ely Porat. Overlap matching. *Inf. Comput.*, 181(1):57–74, 2003.

[5] Amihood Amir, Estrella Eisenberg, and Ely Porat. Swap and mismatch edit distance. *Algorithmica*, 45(1):109–120, 2006.

[6] Amihood Amir, Martin Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Inf. Process. Lett.*, 49(3):111–115, 1994.

[7] Peter Clifford and Raphaël Clifford. Self-normalised distance with don't cares. In *Symposium on Combinatorial Pattern Matching*, pages 63–70, 2007.

[8] Raphaël Clifford, Klim Efremenko, Benny Porat, and Ely Porat. A black box for online approximate pattern matching. In *Symposium on Combinatorial Pattern Matching*, pages 143–151, 2008.

[9] Zvi Galil. String matching in real time. *Journal of the ACM*, 28(1):134–149, 1981.

[10] Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37(1):63–78, 1988.

[11] I. V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 1966.

[12] William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.