Pattern matching in pseudo real-time

Raphaël Clifford[a], Benjamin Sach[a,*]

[a]*Department of Computer Science, University of Bristol, UK*

Abstract

It has recently been shown how to construct online, non-amortised approximate pattern matching algorithms for a class of problems whose distance functions can be classified as being local. Informally, 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 $T[i, i+m-1]$ can be expressed as $\Sigma_j \Delta(P[j], T[i+j])$, where Δ is any distance function between individual characters. We show in this work how to tackle online approximate matching when the distance function is non-local. We give new 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 and L_1 and L_2 rearrangement distances. The resulting 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. It is an important feature of this model that the asymptotic time complexities must not be amortised. We refer to this as the *pseudo real-time* (PsR) model by analogy to the *real-time* model where

*Corresponding author

the time per input must be constant. The algorithms we present are in the RAM model with a word size of $\Omega(\log n)$ (where n is the total text length).

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 [7]. 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*. Informally, we think of a local distance function as one where the distance between a pattern P and a substring of the text T can be written as $\Sigma_j \Delta(P[j], T[i+j])$, where Δ is any distance function between individual characters. In other words, the distance was simply measured as the sum of the distances between individual symbols. Formally we define a local distance function as being expressible in the form

$$\Delta(P[0], T[i]) \circ \Delta(P[1], T[i+1]) \circ \ldots \circ \Delta(P[m-1], T[i+m-1])$$

where $\circ$ is any associative operator with arity two. Throughout we will assume that both Δ and $\circ$ are computable in constant time. When $\circ$ is defined to be addition, the definition simplifies to $\Sigma_j \Delta(P[j], T[i+j])$ as above and includes matching under the Hamming norm and numerical measures such as L_p^p norms for constant $p \geq 1$. When $\circ$ is defined to be the logical and operation, the definition simplifies to $\bigwedge_j \Delta(P[j], T[i+j])$ and includes exact matching and exact matching with wildcards. The previous work of [7] also considered the k-mismatch problem which can also be shown to be local under the given definition. In this case, we define $a \circ b$ to be $min(a+b, k+1)$.

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 [2]. There is a function match between pattern P and text substring $T[i, i+m-1]$ if there exists a function f (possibly different 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 $P = aba$ has a function match with $T = xyx$ but not $T' = xyy$ as a cannot be mapped to two different letters. In the previous black box approach that we briefly describe in Section 2, distances are found independently for different substrings of the pattern and the results combined. However, in this case whether $P[2] = a$ matches $T'[2] = y$ depends on the function chosen to map the characters in $P[0, 1]$ and vice versa. Therefore, any matchings for the substrings of P have to depend on the results for all other substrings. In general for non-local distance functions, we must find a way to handle efficiently 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 non-amortised online ones. The techniques are necessarily no longer black box, depending in detail on the specific offline algorithm being considered. The running time per input character of the new online pattern matching algorithms is guaranteed in each case to be within a log factor of its offline counterpart.

Our results are summarised in Table 1 which gives the time complexity of the best known offline algorithm divided by the text length and the multiplicative

penalty incurred by our onlinization process. All algorithms listed use only $O(m)$ space. In some cases, the methods we present allow us to create online pattern matching algorithms whose running time is the same as in the offline case, that is without any asymptotic overhead at all. For completeness, we also give examples of offline algorithms which translate immediately to the PsR setting with little or no modification.

Problem	Offline per char time	Online/PsR penalty
Method : Splitting [7] *(summarised in Section 2)*		
local matching	various [7]	$O(\log m)$ [7]
Method : Immediate		
edit distance/LCS	$O(m)$ [14]	$O(1)$
L_1 rearrangement	$O(m)$ [1]	$O(1)$
Method : PsR cross-correlations (Section 3.1)		
function (randomised)	$O(\log m)$ [2]	$O(\log m)$
self normalised	$O(\log m)$ [6]	$O(\log m)$
L_2 rearrangement	$O(\log m)$ [1]	$O(\log m)$
Method : Real-time KMP (Section 3.1.3)		
parameterised	$O(\log \lvert\Sigma_P\rvert)$ [5]	$O(1)$
Method : Split & correct (Section 3.2)		
function (deterministic)	$O(\lvert\Sigma_P\rvert \log m)$ [2]	$O(\log m)$
swap-mismatch	$O(\sqrt{m \log m})$ [4]	$O(1)$
swap	$O(\log m \log \lvert\Sigma_P\rvert)$ [3]	$O(\log m)$
overlap	$O(\log m)$ [3]	$O(\log m)$
Method : Split & feed (Section 3.3)		
k-differences	$O(k)$ [13]	$O(\log m)$
k-diff with transpositions	$O(k)$ [13]	$O(\log m)$

Table 1: Summary of new pseudo real-time (PsR) pattern matching results.

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 $|P| = m$. The alphabet from which the characters in the input are chosen 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 i-th character of T. This is a natural way to discuss alignments when the text is being streamed. The usual offline notion where $P[0]$ is aligned with the $T[i]$ will be termed a *left alignment* to avoid confusion.

The ideas we present build on the method of [7] for translating offline algorithms for problems with local distance functions (defined in the introduction). We briefly recap their methods here, which we refer to throughout as the *local method*. In the following we assume that the $\circ$ operator is addition but the result

immediately generalises to any associative operator (with arity two). Fischer and Stockmeyer [9] used a similar technique for online integer multiplication on multi-tape turing machines. 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 $S_1 = P[0, m/2 - 1]$ and subpattern S_j has length $m2^{-j}$ for $1 < j < s$ where $s \approx \log_2(m)$. 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 $C[i - j]$. At the end of this step C will contain $\sum_j \Delta(P[j], T[i + j])$ for every left alignment i in T as required.

To ensure that the work for each subpattern is completed before its result is needed, the text is partitioned conceptually 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 $3|S_j|/2$, with an overlap of length $|S_j|$ with the previous partition. 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. The total space requirement is $O(m)$, matching that of the corresponding offline algorithms. Let $T(n, m)$ be the time complexity of the offline algorithm used as a black box. The running time per text input character is shown to be $O(\Sigma_{j=1}^{\log_2 m} T(n, 2^{j-1})/n))$ which is bounded above by $O(T(n, m) \log(m)/n)$. Thus, for example, pattern matching with wildcards is shown to require $O(\log^2 m)$ time per text input character.

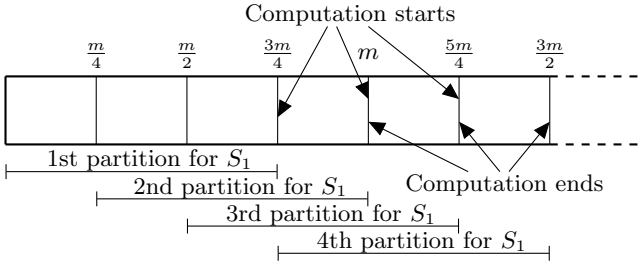

Figure 1: Partitioning of the text for subpattern S_1.

We are not aware of other previous work directly on the question of pattern matching in the streaming setting. However, if linear time preprocessing is allowed, it was shown in [12] that the k-difference problem can be solved in $O(k)$ per character. This work is notable for its ability to add characters both to the right and left of any growing portion of the text and still achieve this tight time bound. Unfortunately their techniques are not applicable as we do not have access to the text in advance so cannot perform the preprocessing required.

3. Algorithms for pseudo real-time translation

3.1. Pseudo real-time cross-correlation method

The cross-correlation and its implementation via the Fast Fourier Transform (FFT) 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 PsR simply and efficiently by using as its main tool the replacement of the offline cross-correlation with a PsR version. Lemma 1 gives the running time of this PsR replacement.

Lemma 1. *Let T be an array (length n) received online and P be an array (length m) received in advance. For any i, when character $T[i]$ arrives, we can compute $(T \otimes P)[i-m+1] = \sum_j T[i+j-m+1]P[j]$ in $O(\log^2 m)$ time.*

Proof. As the cross-correlation problem is local, the local method of [7] can be applied. Using the standard $O(n \log m)$ time offline FFT based cross-correlation algorithm we get the the stated result. Note that the improvement from the original FFT complexity of $O(n \log n)$ results from the well-known pattern matching trick of dividing the text into $\lfloor n/m \rfloor$ overlapping sections, each of length $2m$. □

3.1.1. Self-normalised matching

The self-normalised distance between a pattern P (with integer valued symbols) and text T is defined in relation to the well-known L_2 distance where the objective is to calculate (the square root of) $\sum_j (P[j] - T[i+j])^2$ for all i. In the self-normalised variant, the pattern can be normalised by adding an equal quantity to each pattern value to minimise the L_2 distance. This quantity can be different for each alignment. Wildcards are also allowed in the pattern and the text. The distance can be formalised as $d(i) = \min_\alpha d_\alpha(i)$ where

$$d_\alpha(i) = \sum_{j=0}^{m-1} (\alpha + P[j] - T[i+j])^2 P'[j]T'[i+j]\,.$$

Here, P' and T' are binary arrays with $P'[j] = 0$ iff $P[j]$ is a wildcard (respectively for T'). This ensures that if either $P[j]$ or $T[i+j]$ is a wildcard at alignment i then the contribution of the pair to the distance is zero. Observe that as α is defined globally for each alignment, the problem is non-local. The problem was first considered by Clifford and Clifford [6] who gave an offline $O(n \log m)$ algorithm using the above formalisation. The online problem is to output $d(i)$ as $T[i+m-1]$ arrives under strict non-amortised time limits. We begin by briefly explaining a variant of their offline algorithm. By differentiating $d(i)$ with respect to α, Clifford and Clifford, show that the value of α which minimises $d(i)$ is given by

$$\hat{\alpha}_i = \frac{\sum_j P'[j]T''[i+j] - \sum_j P''[j]T'[i+j]}{\sum_j P'[j]T'[i+j]}\,.$$

Here P'' and T'' are P and T with all wildcards replaced with zeroes. Observe that $\hat{\alpha}_i$ can computed for all i using three cross-correlations (one for each sum). It remains to show how to compute $d_{\hat{\alpha}_i}(i) = d(i)$ for each alignment i. It is here we deviate from the original algorithm. Using a similar technique to that of Amir et al. for computing rearrangement distances [1] (see Section 3.1.4) we observe that for fixed i, the function $d_\alpha(i)$ is a polynomial in α of degree 2. Therefore given $d_0(i), d_1(i), d_2(i)$ and $\hat{\alpha}_i$ we can compute $d_{\hat{\alpha}_i}(i)$ by polynomial interpolation in constant time. To compute $d_\alpha(i)$ for fixed α we multiply out the distance function to obtain six terms which can be computed using cross-correlations in $O(n \log m)$ time.

Theorem 3.1. *The self-normalised distance problem (with wildcards) can be solved in $O(\log^2 m)$ time per character and $O(m)$ space in the PsR model.*

Proof. The offline algorithm for self-normalised distance presented above performs three stages. First, the minimising values of α are obtained using a constant number of cross-correlations. Second, $d_0(i), d_1(i)$ and $d_2(i)$ are computed for all i. This again uses a constant number of cross-correlations. By replacing the cross-correlations with the PsR version from Lemma 1, we can compute $d_0(i), d_1(i), d_2(i)$ and $\hat{\alpha}(i)$ as $T[i + m - 1]$ arrives in $O(\log^2 m)$ time per character and $O(m)$ space. The final stage uses polynomial interpolation to find $d_{\hat{\alpha}(i)}(i)$ from $d_0(i), d_1(i), d_2(i)$ and $\hat{\alpha}(i)$ in constant time. As this stage is independently computed for each i, it is also naturally online. □

3.1.2. Function matching (randomised)

For the function matching problem (defined in the introduction), Amir et al. [2] give an $O(n \log m)$ time randomised solution with probability $1/n$ of declaring a false positive. They also give a deterministic solution for small pattern alphabets that runs in $O(n|\Sigma_P| \log m)$ time which we consider in Section 3.2.1. We again begin by briefly explaining their randomised offline algorithm.

Amir et al. begin by proving that for any pattern P, there exists a pair of patterns such that for all i, P function matches $T[i, i + m - 1]$ iff both these patterns also function match $T[i, i + m - 1]$. Further, both patterns have the property that no symbol occurs more than twice. These patterns (both of length m) can be constructed from a single pass of P in $O(m)$ time without knowledge of T (and hence can be computed during preprocessing). We direct the reader to Lemma 3 of [2] for details of the constructions and continue with the assumption that no symbol occurs more than twice in P.

The central idea is to first assign each symbol in the text alphabet to a numerical value between 1 and n^2. For the pattern, each symbol which occurs only once is assigned the value 0. For symbols which occur twice, the first occurrence is assigned a random number x between 1 and n^2 and the second occurrence is assigned $-x$. They then compute the cross-correlation $T' \otimes P'$, where T' and P' are the chosen numerical representations of the text and pattern respectively.

Recall that $(T' \otimes P')[i] = \sum T'[i+j]P'[j]$ and first observe that symbols that occur only once in P have $P'[j] = 0$ and so contribute 0 to the sum. For a symbol which occurs twice in P in positions j and j' we have that $P'[j] = -P'[j']$. Further for a function match to exist we must have $T[i+j] = T[i+j']$ so $T'[i+j] = T'[i+j']$ and hence $T'[i+j]P'[j] - T'[i+j']P'[j'] = 0$. Therefore if a function match exists then $(T' \otimes P')[i] = 0$. The authors then go on to prove that the probability that $(T' \otimes P')[i] = 0$ when a function match does not exist is at most $1/(n^2)$ when the numerical values are chosen uniformly. Therefore the probability over all alignments of a false positive occurring is at most $1/n$.

Theorem 3.2. *The function matching problem can be solved in the randomised PsR model in $O(\log^2 m)$ time per character and $O(m)$ space with probability $1/n$ of declaring a false positive.*

Proof. The pattern transformation is independent of the text so can be performed before any text characters arrive. If the text transformation could be performed in PsR, the cross-correlation can be performed in $O(\log^2 m)$ time per character as described by Lemma 1.

The text transformation maps text characters to integers in the range 1 to n^2. Recall that we are working in the RAM model with words of size $\Omega(\log n)$ and therefore we can store each character of T' in a constant number of words.

The text transformation can certainly be performed directly in $O(\log |\Sigma_T|)$ time per character if we are prepared to precompute and store the text symbol mapping using $\Theta(|\Sigma_T|)$ space. However, $|\Sigma_T|$ may be much larger than m. To ensure that this is not the case we also perform an initial (deterministic) text transformation which reduces the size of Σ_T to be at most m. This transformation is also performed in PsR.

Our initial transformation constructs a text T'' with an alphabet which is a subset of $\{1 \ldots m\}$ from the original text T in PsR as follows. As text characters arrive we maintain a balanced binary search tree of the at most m symbols of the text symbols which occur in a sliding window of m positions in T with their corresponding symbols in T'' (which will change as the window slides). When a new text character $T[i]$ arrives, if there is a pair $(T[i], a)$ in the tree, we let $T''[i] = a$. If the character is not in the tree then we let $T''[i]$ be first unused symbol from $1 \ldots m$ and insert the pair $(T[i], T''[i])$ into the tree. We also remove any symbol from the tree which no longer occurs in the last m positions. As the tree is of size $O(m)$, the updates take $O(\log m)$ time per character. Observe that as the sliding window is of size m, if $T[i]$ and $T[i']$ are at most m positions apart then $T[i] = T[i']$ iff $T''[i] = T''[i']$. Therefore by the problem definition, a function match occurs in T iff it occurs in T''. □

3.1.3. Parameterised matching

We now turn to parameterised matching, which is a restriction of function matching (defined in the introduction and discussed above) that requires that the function, $f : \Sigma \to \Sigma$ be injective (i.e. one-to-one). While our PsR solution to this problem does not use the PsR cross-correlation method we include it

here because of its relation to function matching. Amir, Farach and Muthukrishnan [5] gave an $O(n \log |\Sigma_P|)$ time solution for this problem based on the Knuth-Morris-Pratt (KMP) algorithm for exact string matching [11]. Their algorithm is suitable for online computation but the time complexity is amortised. We briefly explain their algorithm and demonstrate that it can be deamortised by the technique used by Galil [10] to deamortise the KMP algorithm.

Amir et. al maintain the length $\ell(i)$, of the longest prefix of P which has a parameterised match with (p-matches) a suffix of $T[0 \dots i]$. A p-match occurs between P and $T[i-m+1, i]$ iff $\ell(i)$ is m. When a new text character $T[i+1]$ arrives the algorithm determines whether the $(\ell(i)+1)$-length prefix of P p-matches the $(\ell(i)+1)$-length suffix of $T[0 \dots i+1]$. This can be decided by comparing $P[\ell(i)]$ and $T[i+1]$ with their previous occurrences in the $\ell(i)$-length P prefix and T suffix respectively. If exactly one has a previous occurrence then no match occurs (as it must be aligned with an occurrence of a different character). Similarly, if neither occurs previously then a match does occur. If it is found that an $(\ell(i)+1)$-length p-match does not occur then we follow a 'failure' link as in KMP. The value $\mathit{fail}(\ell(i))$ is the length of the longest proper prefix of $P[0 \dots \ell(i)-1]$ which p-matches a suffix of $P[0 \dots \ell(i)-1]$. Amir et al. showed that p-matching is transitive and therefore $P[0 \dots \mathit{fail}(\ell(i))-1]$ p-matches $T[i-\mathit{fail}(\ell(i))+1, i]$. The failure links depend only on P and Amir et al. demonstrate that the links can be computed in $O(m \log |\Sigma|)$ time. Once a failure link has been followed the algorithm continues determining $\ell(i+1)$ using the new (smaller) prefix of length $\mathit{fail}(\ell(i))$. This process may be repeated up to m times for a given $T[i]$. However, over all the number of failure links followed in linear as ℓ only increases at most n times. The overall time complexity is $O(n \log |\Sigma|)$ due to the need to look up the previous occurrences of arriving text characters (in a binary search tree).

Theorem 3.3. *The parameterised matching problem can be solved in the PsR model in $O(\log |\Sigma_P|)$ time per character and $O(m)$ space.*

Proof. Galil [10] showed that any linear time online algorithm fulfilling a simple predictability condition can be deamortised to create a real-time algorithm. Here we show that this technique can be applied to the $O(n \log |\Sigma|)$ online algorithm described above to create an $O(\log |\Sigma|)$ time per character PsR algorithm. The new PsR algorithm has two 'processes' which operate in parallel. Process **A** is the original online algorithm which only provides its output to process **B**. Process **B** is a new algorithm which always takes $O(\log |\Sigma|)$ time per character. Process **B** always processes the most recently arrived character but as **A** is amortised it may fall behind **B**. We consider a *unit of work* performed by **A** to be one of the following: *following a failure link* or *outputting a result*. Process **B** operates as follows when $T[i]$ arrives. First it waits until **A** has done two units of work (or **A** has stopped computing) then if **A** has outputted a result for right alignment i, **B** outputs the same result otherwise, **B** outputs "no match". Certainly, this new algorithm is non-amortised and produces an output as each text character arrives in $O(\log |\Sigma_P|)$ time. It remains to show that the algorithm gives no false negatives.

Let i be a text index of a right alignment where a match occurs and let k be the smallest integer such that **A** performed computation on $T[i-k]$ as it arrived. The central observation is that as $\ell(i) = m$ and for all i', $\ell(i'+1) \leq \ell(i')+1$, at most k failure links can be followed by **A** while processing $T[i-k+1, i]$ as each link decreases ℓ by at least one. Therefore while processing $T[i-k+1, i]$, **A** performs at most $2k$ units of work. Further as between indices $i'-k+1$ and $i'-1$, **A** does not catch up with **B**, it performs two units of work during each text arrival. Thus **A** catches up with **B** as $T[i]$ arrives so **B** outputs "match". □

3.1.4. L_2 rearrangement distance

The L_2 rearrangement distance problem, first introduced by Amir et al. [1], allows us to describe a more sophisticated application of PsR cross-correlations. At right alignment i, consider any permutation π such that $T[i-m+1+j] = P[\pi(j)]$ for all j and define $cost_{re}(\pi) = \sum_j (j-\pi(j))^2$. The L_2 rearrangement distance is defined to be the minimum $cost_{re}$ over all such permutations. If no such permutation exists, then the distance is defined to be ∞. Observe that this is the case iff there exists some character which occurs more frequently in P than in $T[i-m+1, i]$. Such alignments can be simply detected using a sliding window approach in $O(\log m)$ time per character. We briefly recap the offline solution and demonstrate how it can be converted to operate in PsR.

For all $a \in \Sigma$, let $\psi_a(X)$ be an array of the indices of all occurrences of character a in some string X; further we define $occ_a(X) = |\psi_a(X)|$. Amir et al. showed that the minimum cost rearrangement is to move the j-th occurrence of each symbol in the pattern to align with the j-th occurrence of the same symbol in the text substring (for all symbols and all j). The index of the j-th occurrence of $a \in \Sigma$ is given by $\psi_a(P)[j]$ and similarly the index of j-th occurrence of a in $T[i-m+1, i]$ is $\psi_a(T[i-m+1, i])[j]$. Therefore the contribution to the cost is $((\psi_a(P)[j] - \psi_a(T[i-m+1, i])[j])^2$. If the cost is finite, we have that $occ_a(T[i-m+1, i]) = occ_a(P)$ and therefore $\psi_a(T[i-m+1, i])[j] = \psi_a(T)[occ_a(T[0, i]) - occ_a(P) + j] - (i-m+1)$. Summing over all pattern characters, we have that the minimum cost rearrangement is given by:

$$G_\alpha(P,T)[i] = \sum_{a \in \Sigma} d_\alpha(\psi_a(P), \psi_a(T))[occ_a(T[0,i])] \text{ where } \alpha = (i-m+1)$$

$$\text{and } d_\alpha(P',T')[i'] = \sum_{j=0}^{|P'|-1} (\alpha + P'[j] - T'[i'-|P'|+1+j])^2$$

Note that here d_α is a simplification of the distance function used in Section 3.1.2 as there are no wildcards in the input. Amir et al. observe that G can be viewed as a polynomial of degree 2 in α (for fixed P and T) and therefore if we can compute $G_\alpha(P,T)[i]$ for three fixed values of α, by polynomial interpolation we can find $G_\alpha(P,T)[i]$ for any α in constant time. Therefore they concentrate on computing $G_\alpha(P,T)[i]$ for fixed $\alpha = 0, 1, 2$. They next observe that although the distance G is the sum over $|\Sigma|$ terms we have that $occ_a(T[0, i-1]) = occ_a(T[0, i])$ for all $a \in \Sigma$ except $a = T[i]$. Therefore

$G_\alpha(P,T)[i-1]$ and $G_\alpha(P,T)[i]$ differ only on one term. Thus if we can compute $d_\alpha(\psi_a(P),\psi_a(T))$ for all $a \in \Sigma$, we can compute $G_\alpha(P,T)[i]$ (for fixed α) by a simple sliding window approach. Amir et al. then consider computation of $d_\alpha(\psi_a(P),\psi_a(T))$ as a pattern matching problem with pattern $\psi_a(P)$ and text $\psi_a(T)$. They demonstrate that by multiplying out the quadratic equation, and separating, d_α can be expressed as six sums, each of which can be computed using a cross-correlation and linear time to square the values in the pattern and/or text for some terms. Therefore $d_\alpha(\psi_a(P),\psi_a(T))$ for some fixed α and all $a \in \Sigma$ can be computed in $O(\sum_{a\in\Sigma} occ_a(T)\log occ_a(P)) = O(n\log m)$ time.

Theorem 3.4. *The L_2 rearrangement distance problem can be solved in the PsR model in $O(\log^2 m)$ time per character and $O(m)$ space.*

Proof. We first observe that as the polynomial interpolation is performed independently for each alignment (in constant time), we only need to consider PsR computation of G_α for constant α. Further the sliding window approach used to compute $G_\alpha(P,T)[i]$ from $G_\alpha(P,T)[i-1]$ is also naturally suited to PsR computation assuming we can compute the value of $d_\alpha(\psi_a(P),\psi_a(T))[occ_a(T[0,i])]$ for $a = T[i]$ as $T[i]$ arrives. In the offline setting, Amir et al. showed that we can precompute all such values using cross-correlations in $O(n\log m)$ time. In the online setting, consider the related problem of computing $d_\alpha(P',T')$ for all i' with pattern P' and text T' which arrives online. Certainly we could use PsR cross-correlations to compute $d_\alpha(P',T')[i']$ as $T'[i']$ arrives in $O(\log^2|P'|)$ time. Analogously to the decomposition of the problem into $|\Sigma|$ subproblems in the original algorithm, we decompose the stream into $|\Sigma|$ substreams. Each substream corresponds to a single symbol $a \in \Sigma$ with $P' = \psi_a(P)$ and text $T' = \psi_a(T)$. When a character $a = T[i]$ arrives we consider it to be the arrival of $\psi_a(T)[occ_a(T[0,i])] = i$ in the substream for symbol a which incurs $O(\log^2|P'|)$ work. Note that during this arrival no other substream performs any computation as no character has arrived in their stream. Therefore the total time complexity is upper bounded by $O(\log^2|P'|) = O(\log^2 m)$ time per character. As the cross-correlations use linear space, the total space is upper bounded by $O(\sum_a occ_a(P)) = O(m)$. □

3.2. Split and correct

The simplest variant of the 'split and correct' method operates as follows: First, we *split* the text into $O(\log m)$ consecutive subpatterns each half the length of the last and apply the local method to find the distance from all subpatterns to the text at each alignment. Second, we *correct* the distance at each alignment to account for non-local effects between subpatterns that are ignored by the local method. The function matching problem considered below gives a simple concrete example of this method. For more sophisticated examples such as swap-mismatch (Section 3.2.2) and overlap matching (Section 3.2.4), it is also necessary to perform a constant number of different transformations to each subpattern before matching. These transformations are used to simulate different possible non-local effects. We also need to determine which transformations would have been applied to each subpattern 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.

3.2.1. Function matching (deterministic)

The function matching problem (defined in the introduction) gives us a simple example of where the 'split and correct' method is effective. Amir et al. [2] give a solution for small pattern alphabets that runs in $O(n|\Sigma_P|\log m)$ time. Here we use this algorithm as a black box. Motivated by the local method we consider splitting the pattern into $O(\log m)$ consecutive subpatterns $S_1, S_2 \ldots S_s$, each half the length of the last so that $S_1 = P[0, m/2 - 1]$ and $S_s = P[m-1]$. While function matching is not a local problem, we can still use local method to schedule the use of an offline algorithm to compute function matches of each subpattern $S_1, S_2 \ldots S_s$ with T. Here we consider the 'distance' at some alignment between the k-th subpattern, S_k, and T to be 1 if a function match exists and 0 otherwise. Recall that the local method adds the distances to an auxiliary array C as they are computed. Observe that $C[i-m+1]$ will equal s iff all subpatterns had a function match with the text at right alignment i. If there is a function match at some alignment then there is certainly a function match for each subpattern (using the same function). Therefore $C[i-m+1] = s$ is a necessary condition for a match.

By the problem definition, a function match exists at some alignment iff for each symbol $a \in \Sigma_P$, all occurrences of a in P align with occurrences of the same character in T. Further, if $C[i-m+1] = s$ we know that for each subpattern S_k, all occurrences of a align with occurrences of the same character in T. We call this character $f_k(a)$ (which may be different at each alignment). Observe that P function matches $T[i-m+1, i]$ iff $f_1(a) = f_2(a) = \ldots = f_s(a)$ for all a. To enable us to determine any $f_k(a)$ we keep a list of the indices of the first occurrences of each symbol in each subpattern (in $O(m)$ space). As there are only $O(|\Sigma_P|\log m)$ $f_k(a)$ characters, we can check whether $f_1(a) = \ldots = f_s(a)$ for all a at each alignment by inspecting the text in $O(|\Sigma_P|\log m)$ time per character.

Theorem 3.5. *The function matching problem can be solved deterministically in $O(|\Sigma_P|\log^2 m)$ time per character and $O(m)$ space in the PsR model.*

Proof. We use the $O(n|\Sigma_P|\log m)$ offline algorithm of Amir et al. [2] as our black box for the local method. This stage has a time complexity of $O(|\Sigma_P|\log^2 m)$ per character as there are $O(\log m)$ subpatterns. From the local method, we also have that the space complexity is $O(m)$. We also check that the functions found for each subpattern are consistent at each alignment. This is done separately for each alignment and as was observed above takes $O(|\Sigma_P|\log m)$ time per character. □

3.2.2. Swap-mismatch

The swap-mismatch distance between equal length strings is the minimum number of moves required to transform P into T referred to as $cost_{sm}(P, T)$.

The valid moves are *swap* (swap two adjacent characters) and *mismatch* (replace a character). Further, each character in P can only be involved in at most one move. On non-equal length strings, at right alignment i, the distance is defined to be $cost_{sm}(P, T[i-m+1, i])$. We present an $O(\sqrt{m \log m})$ time per character solution using the best known offline method of Amir et al. [4] as a black box.

Let $S_1, S_2 \dots S_s$ be consecutive subpatterns of P, each half the length of the last, as defined in the local method. Additionally, let l_j and r_j be the indices of the leftmost and rightmost characters of section S_j respectively. Consider naively using the local method to compute $\sum_{j=1}^{s} cost_{sm}(S_j, T[i-m+1+l_j, i-m+1+r_j])$ for all right alignments, i, in PsR. In this attempted solution we have inadvertently added the additional constraint that r_j and l_{j+1} cannot be swapped (for any j). Our solution corrects for this by defining a set of 'boundary indicators' for all $0 < j < s$: $b_{i,j} = 1$ if $P[r_j]$ and $P[l_{j+1}]$ are to be swapped in our transformation of P into $T[i-m+1, i]$ and 0 otherwise. Trivially, we let $b_{i,0} = b_{i,s} = 0$ for all i. Definition 1 gives the conditions under which we swap l_{j+1} and r_j. The observation and notation in Lemma 2 are partly inspired by the work on swap and overlap matching by Amir et al. [3].

Definition 1. *At right alignment i, the j-th boundary indicator, $b_{i,j} = 1$ iff*

1. $T[i-m+1+l_{j+1}] \neq P[l_{j+1}] = T[i-m+1+r_j]$ *and*
2. $T[i-m+1+r_j] \neq P[r_j] = T[i-m+1+l_{j+1}]$ *and*
3. *There exists an odd ℓ such that $P[r_i-\ell] \neq y$ or $T[i-m+1+r_i-\ell] \neq x$ and $T[i-m+1+r_i-\ell+1..i-m+1+r_i] = y(xy)^*$ and $P[r_i-\ell+1..r_i] = x(yx)^*$ where $y = T[i-m+1+r_i]$ and $x = P[r_i]$ and $y(xy)^*$ is a "y" followed by zero or more copies of "xy".*

Lemma 2. *There is an optimal swap-mismatch transformation of P into $T[i-m+1, i]$ where for all j, r_j and l_{j+1} are swapped iff $b_{i,j} = 1$.*

Proof. For notational simplicity, we prove the case when $n = m$ and $i = 0$. However, the result immediately generalises. Let "$xyx\dots$" be an alternating string of some symbols x and y. We define (q, r) to be a run if $T[q \dots r] = xyx\dots$ and $P[q \dots r] = yxy\dots$ for some $x \neq y$ in Σ and $r > q$. Further we term a run *maximal* if all other runs are either completely contained within it or are disjoint from it. In other words we require that maximal runs are disjoint. If there is no run, (q, r) such that $q \leq j \leq r$ then $P[j]$ cannot be swapped in any correct transformation as this would leave a character unmatched by the definition of a run (and each character can only be involved in one move).

Consider the following transformation of P into T. Any position x which is not in a run is not swapped and is mismatched if $P[x] \neq T[x]$. Any maximal run (q, r) is transformed by swapping $P[q+2x]$ with $P[q+2x+1]$ for all $x < (r-q)/2$. If $r-q$ is even then $P[r]$ is mismatched. By the definition of a run, this transformation correctly transforms P into T. All maximal runs are transformed optimally in isolation. Further as they are disjoint, $P[q]$ or $P[r]$ cannot be swapped with $P[q-1]$ or $P[r+1]$ respectively. Therefore we can consider them in isolation and the transformation is optimal. It is easily verified that in this transformation, for all j, r_j and l_{j+1} are swapped iff $b_{i,j} = 1$. □

For any subpattern S_j, the boundary indicators $b_{i,j-1}$ and $b_{i,j}$ determine whether either r_j or l_j are involved in a swap at alignment i. We need to ensure that if either r_j or l_j is swapped, no further moves are applied to that position. We do this by removing those positions from the subpatterns. For each subpattern we define four transformed subpatterns one for each combination of $b_{i,j-1}$ and $b_{i,j}$. For $x, y \in \{0,1\}$, let $S_j^{(x)(y)} = P[l_j + x, r_j - y]$ represent these transformed subpatterns. Lemma 3 gives the key relationship between the subpatterns $S_j^{(x)(y)}$, the boundary indicators, $b_{i,j}$ and $cost_{sm}(P, T[i-m+1, i])$.

Lemma 3. *The cost of transforming P into $T[i-m+1, i]$ is equal to*

$$\sum_{j=1}^{(s-1)} b_{i,j} + \sum_{j=0}^{s} cost_{sm}(S_j^{(b_{i,j-1})(b_{i,j})}, T[i-m+1+l_j+b_{i,j-1}, i-m+1+r_j-b_{i,j}]).$$

Proof. The function for given i corresponds to a transformation (TR) formed by: First, apply the optimal transformation of each $S_j^{(b_{i,j-1})(b_{i,j})}$ into $T[i-m+1+l_j+b_{i,j-1}, i-m+1+r_j-b_{i,j}]$. Second, swap $P[r_j]$ and $P[l_{j+1}]$ iff $b_{i,j} = 1$. Observe that $S_j^{(b_{i,j-1})(b_{i,j})}$ includes $P[r_i]$ iff $b_{i,j} = 1$ (similarly with $P[l_{j+1}]$). Therefore each position in P is considered exactly once so TR is a correct transformation of P into $T[i-m+1, i]$. It remains to show that this transformation has minimal cost. By Lemma 2 there is an optimal transformation, OPT, which for all j, swaps $P[r_j]$ and $P[l_{j+1}]$ iff $b_{i,j} = 1$. If this optimal transformation requires fewer total moves than TR, there must be a pattern subsection, $S_j^{(b_{i,j-1})(b_{i,j})}$ which OPT transforms into $T[i-m+1+l_j+b_{i,j-1}, i-m+1+r_j-b_{i,j}]$, in fewer than $cost_{sm}(S_j^{(b_{i,j-1})(b_{i,j})}, T[i-m+1+l_j+b_{i,j-1}, i-m+1+r_j-b_{i,j}])$ moves which is a contradiction. □

We can now describe our algorithm which performs three stages:

1. Calculate matches of T against $S_j^{(0)(0)}, S_j^{(0)(1)}, S_j^{(1)(0)}$ and $S_j^{(1)(1)}$ for all j at all alignments. This is done using the local 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 separately for each indicator by checking the conditions in Definition 1 directly in real-time.
3. Combine the results of stages 1 and 2 using the relation stated in Lemma 3. This is computed directly, requiring $O(\log m)$ time per character.

Theorem 3.6. *The swap-mismatch problem can be solved in $O(\sqrt{m \log m})$ time per character and $O(m)$ space in the PsR model.*

Proof. The correctness of our algorithm is immediate from the application of Lemma 3. The offline algorithm of Amir et al. has a $T(n, m) = O(n\sqrt{m \log m})$ time complexity. Therefore stage 1 which uses the local method requires a total of $O(\Sigma_{j=1}^{\log_2 m} T(n, 2^{j-1})/n))$ time per character which is upper bounded by

$O(\sqrt{m \log m})$. For the j-th boundary indicator, all three properties in Definition 1 can be checked in real-time. The first two are immediate, the third by maintaining the length of the longest alternating sequence ending at $T[i-m+r_j]$. As there are $O(\log m)$ such indicators, we require $O(\log m)$ time per character. The final stage also requires $O(\log m)$ time to directly combine the results giving an overall complexity of $O(\sqrt{m \log m})$ time per character. Further, each stage uses $O(m)$ space. □

3.2.3. Overlap matching

The overlap matching problem is defined on a pattern P and text T which are both binary. We call a contiguous sequence of 1s in P or T a 1-segment. P overlap matches $T[i-m+1, i]$ if all 1-segments in P have an even length overlap with 1-segments in $T[i-m+1, i]$. This problem was first considered by Amir et al. [3] and was partly motivated by its use in solving the swap matching problem (see Section 3.2.4).

Consider splitting the pattern into $O(\log m)$ subpatterns each having half the length of the previous one as with the local method. The first subpattern $S_1 = P[0, m/2 - 1]$ 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 problem with this approach is that any 1-segment which crosses a boundary between two subpatterns S_i and S_{i+1} has been split which may make the parity of any overlaps with 1-segments in $T[i-m+1, i]$ incorrect. Instead let S'_j be S_j with any contiguous 1s at the beginning or end of S_j replaced with 0s. We refer to these removed 1-segments as crossing 1-segments and they will be handled separately (analogously to the boundary indicators in the swap-mismatch algorithm above). We use P' to denote $S'_1 S'_2 \dots S'_s$. We can now outline our algorithm which performs all three stages in PsR:

1. Find matches of S'_j for all j in T at all alignments using the local method.
2. Find matches of each crossing 1-segment in T at all alignments.
3. Combine the results of stages one and two by finding all alignments where all S_j and all crossing 1-segments match.

Theorem 3.7. *The overlap matching problem can be solved in $O(\log^2 m)$ time per character and $O(m)$ space in the PsR model.*

Proof. By applying the local method in stage 1 on an offline algorithm for overlap matching to the subpatterns S'_j we compute all overlap matches of P' in T in PsR. We calculate all matches correctly because all 1-segments in P' are completely contained in a single subpattern. Observe that if P matches $T[i-m+1, i]$ then P' matches $T[i-m+1, i]$. Further if P' matches $T[i-m+1, i]$ but P does not match $T[i-m+1, i]$ then there exists a crossing 1-segment in P which has odd overlap with some 1-segment in T. Any mismatching crossing 1-segments are found by stage 2. Therefore we correctly find all alignments where P matches T by finding the alignments where P' and all crossing 1-segments match. This completes the correctness.

Stage 1 can be computed in $O(\log^2 m)$ time per character in PsR using the Amir et al. [3] offline algorithm for overlap matching as a black box. Further, for a given 1-segment, $P[a \dots b]$, there is an odd length overlap with some 1-segment in $T[i-m+1, i]$ iff there exists an odd length 1-segment in $T[i-m+1+a, i-m+1+b]$. We keep a count of the number of odd length 1-segments in $T[i-m+1+a, i-m+1+b]$ (in constant space) which we can update in constant time when each $T[i]$ arrives. However there are only $O(\log m)$ such crossing 1-segments so stage 2 can be computed in $O(\log m)$ time per character. Finally as there are a total of $O(\log m)$ results per alignment, stage 3 can be computed directly in $O(\log m)$ time per character. □

3.2.4. Swap matching

In the swap matching problem, we say that P matches $T[i-m+1, i]$ if we can transform P into $T[i-m+1, i]$ by swapping adjacent characters. As with the swap-mismatch problem, each character is swapped at most once. Amir et al. [3] showed that this problem can be solved offline in $O(n \log m \log |\Sigma_P|)$ time.

Amir et al. [3] first show that the swap matching problem on a binary alphabet can be reduced to overlap matching with a constant overhead. From T they create two text transformations T_e and T_o. T_e is given by first splitting T into maximal length alternating substrings, for example $T = 010001101$ would become "010", "0" "01", "101". All substrings in which the 1s begin on even text indices are replaced with contiguous 1s and all others with 0s so in our example $T_e = 000100111$. T_o is defined by $T_o[i] = (1 - T_e[i])$ for all i. The pattern transformations, P_e and P_o are computed in the same manner. Observe that these transformations can be computed online. Amir et al. show that a swap match occurs at an even alignment i iff P_e overlap matches T_e and P_o overlap matches T_o. Analogously for an odd alignment they consider the pattern/text pairs P_e, T_o and P_o, T_e. Hence, by using the PsR overlap matching algorithm in Section 3.2.3 that we can achieve a time complexity of $O(\log^2 m)$ per character and $O(m)$ space for binary alphabets.

For general alphabets of size $|\Sigma_P|$, Amir et al. show that the swap matching problem can be reduced to swap matching on a binary alphabet with a multiplicative $O(\log |\Sigma_P|)$ factor. The reduction constructs $O(\log |\Sigma_P|)$ binary transformations of P and T where each transformation replaces all characters equal to a chosen symbol with 1 and all other characters with 0. Observe that it is simple to perform each transformation on the text online in real-time. Amir et al. show that a swap match occurs iff an overlap match occurs in all binary transformations. This gives us the desired complexity of $O(\log^2 m \log |\Sigma_P|)$ time per character. However, naively this results in a space complexity of $O(m \log |\Sigma_P|)$. To overcome this, in the proof of Theorem 3.8 we show that we can modify the PsR overlap matching algorithm in Section 3.2.3 to take advantage of the fact that the $O(\log |\Sigma_P|)$ instances of overlap matching in the reduction are binary.

Theorem 3.8. *The swap matching problem can be solved in $O(\log^2 m \log |\Sigma_P|)$ time per character and $O(m)$ space in the PsR model.*

Proof. Recall the PsR overlap matching algorithm in Section 3.2.3. First consider that stage two of the overlap matching algorithm, which finds matches for the $O(\log m)$ crossing 1-segments requires only a total of $O(\log m)$ space (apart from access to the transformed pattern and text). Further observe that there are $O(\log |\Sigma_P|)$ binary patterns and texts and therefore we can pack all the pattern transformations and the last m characters of each text transformation into $O(m)$ words. Hence we can perform stage two in a total of $O(\log m \log |\Sigma_P| + m) = O(m)$ space.

Now consider stage one which finds matches using the local method. The local method makes a number of calls to a suitable offline algorithm and distributes the work over a range of text character arrives. Our reduction runs $p = O(\log |\Sigma_P|)$ instances of the overlap algorithm in parallel all on patterns of the same size. Observe from Section 2 that the partitioning used by the local method depends only on the pattern size. Therefore, all instances of the PsR overlap algorithm make calls to the offline algorithm at the same time and distribute the work over the same range, $T[a \dots b]$, using the same amount of time per character, w and the same amount of space s. The total time per character is pw and the total space is ps. However, we observe that the results of the offline computation can all be stored in only s words by the bit-packing arguments above. Therefore, we can change the algorithm to work at the same total rate but make the offline calls sequential (rather than parallel). Now the total time per character is still pw but the space is only s words and all results are available when $T[b]$ arrives. This gives a total of $O(m)$ space as required. □

3.3. Split and feed

The final conversion technique that we discuss is termed 'split and feed'. This will allow us to handle pattern matching problems defined by dynamic programming recurrences. We will require these recurrences to satisfy three main properties which we show are satisfied by, for example, k-differences (see Section 3.3.1) and k-differences with transpositions (see Section 3.3.2). The first and simplest property is just that the recurrence must be expressible in the following form, $D[j,i] = f(D[j,i-1], D[j-1,i], D[j-1,i-1], T[i], P[j])$. We assume that f is some function that can be computed in constant time. When each text character $T[i]$ arrives, the goal is to output $D[m-1,i]$ as quickly as possible. Observe that any local distance pattern matching problem can be expressed in this form by simply setting $D[j,i] = D[j-1,i-1] \circ \Delta(P[j], T[i])$.

Notation. We use $D[x \dots y, a]$ to refer to the sub-column, $D[j,a]$ where $x \leq j \leq y$. Similarly we use $D[x, a \dots b]$ for the sub-row, $D[x,i]$ where $a \leq i \leq b$. The notation $D[x \dots y, a \dots b]$ refers to the rectangle $D[j,i]$ where $x \leq j \leq y$ and $a \leq i \leq b$, which we commonly refer to as a block.

Analogously to the local method which splits the pattern (and text) into sections, the overall idea of our split and feed method is to split the underlying dynamic programming table into overlapping blocks. For each block we require that we can compute the entries on the bottom and right edges from the entries on the top and left edges of a block using an appropriate offline algorithm

(see Figure 2). Definition 2 gives a more formal description of this, the second property required for the split and feed technique. This offline algorithm may be the same as the one we wish to convert to online or in some case may require some modification. It is, of course, assumed that any such offline algorithm computes the bottom and right edge entries asymptotically faster than computing all block entries directly. The results for bordering blocks are then fed from one to another, hence the name 'split and feed'. A similar technique of dividing the dynamic table into blocks appeared in [8]

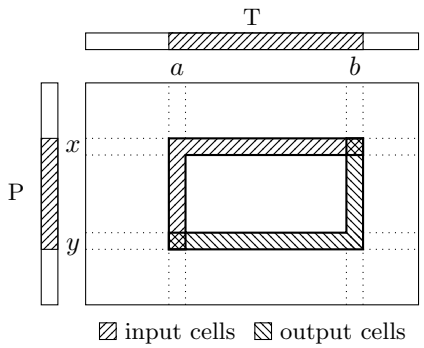

Figure 2: Blockwise decomposition on the block $[x \ldots y] \times [a \ldots b]$.

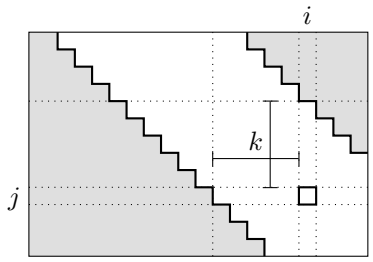

Figure 3: Cells independent of $D[j, i]$ under a k-bounded recurrence (shaded).

Definition 2. *Given a pattern P, a text T and a dynamic programming recurrence, D, as defined above, Algorithm A is blockwise decomposable if given as input, $P[x \ldots y]$, $T[a \ldots b]$, $D[x, a \ldots b]$ and $D[x \ldots y, a]$, A outputs $D[y, a \ldots b]$ and $D[x \ldots y, b]$. We refer to this as running A on block $[x \ldots y] \times [a \ldots b]$.*

The final required property is that the dynamic programming recurrence is k-bounded for some constant k (see Definition 3 and Figure 3). Intuitively, it states that the non-locality in the problem is in some sense bounded. In the case of k-differences, for example, that the number of inserts and deletes is limited.

Definition 3. *A cell $D[j, i]$ is independent of another cell if changing the value of that cell has no effect on the value of $D[j, i]$. A dynamic programming recurrence is k-bounded if every arbitrary cell $D[j, i]$ is independent of $D[j - k - y, i - x]$ and $D[j - x, i - k - y]$ for all $y > x \geq 0$.*

We can now set out the main split and feed result.

Theorem 3.9. *Let D be a k-bounded dynamic programming recurrence of the form given above. Further, let A be offline blockwise decomposable algorithm for computing D which runs in time $T(n, m)$ and uses $S(n, m)$ space. There exists another algorithm in the PsR model which outputs $D[m-1, i]$ as $T[i]$ arrives in $O((T(n, m)/n) \log m + k)$ time per character and requires $O(S(m, m))$ space.*

We delay explicit description of the construction of this new online algorithm and a full proof until Section 3.3.3 in order to present some motivating examples.

3.3.1. k-differences

The first application we give for the split and feed technique is the well known problem called pattern matching with 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 $cost_{kd}(P, T)$. The valid moves are *insert* (insert a character), *delete* (delete a character) and *mismatch* (replace a character). In the pattern matching case at each right alignment i the goal is to output the minimum distance from P to any suffix of $T[0 \dots i]$. This can be formalised as $\min_{\ell \leq i} \text{cost}_{kd}(P, T[\ell, i])$. In the k-difference problem the number of moves is restricted to be at most k. As both insert and delete operations are non-local and affect alignment of other characters we cannot simply split the pattern and search each independently. The k-differences problem can be formulated as a dynamic programming recurrence as follows:

$$D[j,i] = \min \begin{cases} D[j,i-1]+1 & \text{(delete)} \\ D[j-1,i]+1 & \text{(insert)} \\ D[j-1,i-1]+1-\text{eq}(i,j) & \text{(mismatch)} \\ k+1 & (k\text{-bounded}) \end{cases}$$

Here $\text{eq}(i,j) = 1$ if $T[i] = P[j]$ and 0 otherwise. Further we initialise $D[j,-1] = j+1$ and $D[-1,i] = 0$ for all i, j. In this formulation an output of $k+1$ is interpreted as "more than k moves required". As required, $D[m-1,i] = \min_{\ell \leq i} \text{cost}_{kd}(P, T[\ell, i])$. Consider the calculation of cell $D[j,i]$. If its value comes from $D[j-1,i]$, this corresponds to an insert, $D[j,i-1]$ corresponds to a delete and $D[j-1,i-1]$ to a (mis)match.

Lemma 4. *The k-difference problem is k-bounded.*

Proof. The cell $D[j-k-y, i-x]$ with $y > x \geq 0$ is at least $k+1$ cells further above $D[j,i]$ than it is to the left of it. Thus any sequence of moves from $D[j-k-y, i-x]$ to $D[j,i]$ must include at least $k+1$ inserts so cannot affect $D[j,i]$. Similarly for $D[j-x, i-y-k]$ with $y > x \geq 0$. □

The last property we must show for k-difference is that there is an offline algorithm which is blockwise decomposable and which runs in $O(k(H+W))$ per block of width $W = b - a + 1$ and height $H = y - x + 1$. We derive such an algorithm by a modification of the Landau-Vishkin algorithm [13] for the k-difference problem. Algorithm 1 sets out the main steps. The main change to the original algorithm that is made is to permit the use of block edges in the dynamic programming table as input rather than strings. In the algorithm, LCE refers to Longest Common Extension. The value $\ell = \text{LCE}(i,j)$ is defined to be the largest ℓ such that $T[i, i+\ell-1] = P[j, j+\ell-1]$.

Lemma 5. *Algorithm 1 is a blockwise decomposable offline solution for the k-difference problem and runs in $O(k(H+W))$ time for a block of width W and height H using $O(H+W)$ space.*

Algorithm 1: Blockwise-decomposable k-differences

Input: $P[x \ldots y]$, $T[a \ldots b]$, $D[x, a \ldots b]$ and $D[x \ldots y.a]$
Output: $D[y, a \ldots b]$ and $D[x \ldots y, b]$

```
1  Preprocess P[x...y] and T[a...b] for constant time LCE queries [15];
2  for d = y − a to x − b do
3      i ← max(a, x − d);  v ← D[i + d, i];
4      ch(d, v) ← i + LCE(i + 1, i + 1 + d);
5      if ch(d, v) > min(b, y − d) then ch(d, v) ← ∞;
6      ch(d, v′) ← −∞ for all v′ < v;
7  end
8  for v = 0 to k do
       // Compute ch(d, v + 1) from ch(d − 1, v), ch(d, v) and ch(d + 1, v)
9      for d = y − a to x − b do
10         if ch(d, v) ≠ −∞ then
11             i ← max(ch(d − 1, v) + 1, ch(d, v) + 1, ch(d + 1, v)) ;
12             ch(d, v + 1) ← i + LCE(i + 1, i + 1 + d) ;
13             if ch(d, v + 1) > min(b, y − d) then ch(d, v) ← ∞;
14         end
15     end
16 end
17 for d = y − a to x − b do
18     i = max(a, x − d);
19     if ch(d, k + 1) < ∞ then D[i + d, i] ← k + 1;
20     else D[i + d, i] ← min{k | ch(d, k) = ∞};
21 end
```

Proof. We define x, y, a, b as in Algorithm 1 so that the block has width $W = b - a + 1$ and height $H = y - x + 1$. Consider the dth diagonal in the dynamic programming table, defined by $D[i+d, i]$ where $\min(a, x-d) \leq i \leq \max(b, y-d)$. Observe that the values of the cells on a diagonal are monotonic and non-decreasing. For each diagonal let $ch'(d, v) = i$ for all d and $0 \leq v \leq k + 1$ such that $D[i + d, i] = v$ and $D[i + 1 + d, i + 1] > v$. In other words $ch'(d, v)$ is the column index of the last cell on diagonal d to have the value v. We can determine the value of any cell $D[i + d, i]$ on diagonal d by finding the v such that $ch'(d, v - 1) < i$ and $ch'(d, v) \geq i$. Thus, inspection of the values of ch' is sufficient to determine $D[y, a \ldots b]$ and $D[x \ldots y, b]$ as required.

We begin by proving that for all d, v, when Algorithm 1 concludes either $ch(d, v) = ch'(d, v)$ or $ch(d, v) = \pm\infty$. For some d, let v be smallest such that $ch'(d, v + 1) \neq ch(d, v + 1) \neq \pm\infty$. The value of $ch(d, v + 1)$ is completely determined by line 11 of algorithm 1:

$$i \leftarrow \max(ch(d - 1, v) + 1, ch(d, v) + 1, ch(d + 1, v)) .$$

As v is smallest and $ch(d, v+1) \neq +\infty$ then $ch(d-1, v)$ $ch(d, v)$ and $ch(d+1, v)$ must be $-\infty$ or correct. Further, it was proven by Landau and Vishkin [13] that

if $ch(d-1,v)$, $ch(d,v)$ and $ch(d+1,v)$ are all correct then $ch(d,v) = ch'(d,v)$. Consider the case, $ch(d-1,v) = -\infty$ but $ch(d,v)$ and $ch(d+1,v)$ are correct. By the algorithm description as $ch(d-1,v) = -\infty$ and $ch(d,v) \neq -\infty$ we have that $ch'(d-1,v) \leq ch'(d,v) = ch(d,v)$. Therefore, in this case, $ch(d-1,v)$ being incorrectly set to $-\infty$ does not affect calculation of i. The other cases follow similarly leading to the contradiction, $ch'(d,v+1) = ch(d,v+1)$.

Further, $ch(d,v) = \infty$ iff $ch'(d,v)$ is below/right of the block $[x \dots y] \times [a \dots b]$. Thus $ch(d,v) = \infty$ iff $D[max(a,x-d), max(a,x-d)+d] \leq v$ which completes the correctness.

For the running time, we can preprocess the input for constant time LCE queries in $O(W+H)$ time. The initialisation of ch, the central algorithm and determining the output can each be upper bounded by $O(k(W+H))$ time. The total time complexity is therefore $O(k(W+H))$ as required. Further note that only $O(W+H)$ non $-\infty$ values of ch need to be stored at any time. Thus Algorithm 1 can be implemented in $O(W+H)$ space. □

We can now give the main k-differences result.

Theorem 3.10. *The k-differences problem can be solved in $O(k \log m)$ time per character and $O(m)$ space in the PsR model.*

Proof. We have shown that k-differences can be described by a k-bounded dynamic programming recurrence and that an $O(kn)$ blockwise decomposable offline algorithm exists using $O(n)$ space when run on a block of width n and height $m \leq n$. Therefore by Theorem 3.9, the result follows. □

3.3.2. k-difference with transpositions

The k-difference problem with transpositions allows an additional move, *transposition* which swaps two adjacent characters. However, each character can be transposed at most once and all transpositions must be performed before any other moves. Hence a transposition move can be seen as a restriction of the swap move (see Section 3.2.2). Transpositions can be incorporated into the k-difference dynamic programming recurrence as follows (Ukkonen [16]):

$$D[j,i] = \min \begin{cases} D[j,i-1]+1 & \text{(delete)} \\ D[j-1,i]+1 & \text{(insert)} \\ D[j-1,i-1]+1-\text{eq}(i,j) & \text{(mismatch)} \\ D[j-2,i-2]+3-\text{eq}(i,j-1)-\text{eq}(i-1,j) & \text{(transpose)} \\ k+1 & (k\text{-bounded}) \end{cases}$$

Here $\text{eq}(i,j) = 1$ if $T[i] = P[j]$ and 0 otherwise as before. The boundary conditions are set as before. However, this recurrence is not of the form required for Theorem 3.9 as it also depends on $D[j-2,i-2]$. Therefore, we define a new recurrence, D', where $D'[j,i]$ is the tuple $(D[j,i], T[i], P[j], D[j-1,i-1])$. Observe that $D'[j,i]$ is now of the desired form as $P[j-1]$, $T[i-1]$ and $D[j-2,i-2]$ are all encoded in $D'[j-1,i-1]$. Further we can store each tuple in a constant number of words and extract $D[j,i]$ from $D'[j,i]$ in constant time and

thus Theorem 3.9 applies. For notational clarity, we consider the k-difference with transpositions problem to be to output $D'[m-1,i]$ as $T[i]$ arrives in PsR which suffices to solve the original formulation.

Note that this tuple re-encoding technique can be extended to any recurrence where $D[j,i]$ depends on cells in the rectangle $D[(j-h)\dots j,(i-w)\dots i]$ excluding $D[j,i]$ itself as well as $T[i-w,i]$ and $P[j-h,j]$ where w,h are constant.

Lemma 6. *The k-difference problem with transpositions is $(k+1)$-bounded.*

Proof. Consider the first formulation of k-differences with transpositions, D. The cell $D[j-k-y,i-x]$ with $y > x \geq 0$ is at least $k+1$ cells further above $D[j,i]$ than it is to the left of it. As for k-differences, any sequence of moves from $D[j-k-y,i-x]$ to $D[j,i]$ must include at least $k+1$ inserts as both swaps and mismatches act diagonally (and deletes horizontally). Therefore $D[j-k-y,i-x]$ cannot affect $D[j,i]$. Similarly for $D[j-x,i-y-k]$ with $y > x \geq 0$ and hence D is k-bounded. Now consider the second formulation, D'. Let $D'[j,i] = (a,b,c,d)$. By definition, the values of b,c are independent of all cells of D'. Further, a depends on $D'[r,q]$ iff $D[j,i]$ depends on $D[r,q]$. Similarly, d depends on $D'[r,q]$ iff $D[j-1,i-1]$ depends on $D[r,q]$. Hence as D is k-bounded, D' is $(k+1)$-bounded. □

Recall Algorithm 1 which was shown to be a blockwise decomposable offline solution for the k-difference. The main idea of the algorithm was to compute for all d,v the value of $ch(d,v)$, the index of last column on each diagonal d for which $D[j,i] \leq v$. Ukkonen et al. [16] (Section 4, page 117) gave a simple modification to the formula for $ch(d,v)$ (lines 11 and 12 of Algorithm 1) which incorporates the transposition operation. We define Algorithm 2 as the result of applying Ukkonen's modification to lines 11 and 12 of Algorithm 1 as well as the simple modifications to store $D'[j,i]$ as a tuple, $(D[j,i],T[i],P[j],D[j-1,i-1])$.

Lemma 7. *Algorithm 2 is a blockwise decomposable offline solution for the k-difference problem with transpositions and runs in $O(k(H+W))$ per block of width $W = b-a+1$ and height $H = y-x+1$.*

Proof. It is easily verified that as the original algorithm of Ukkonen is correct under this modification ours is also. Ukkonen's modification adds constant additional work to computing each $ch(d,v)$ and hence the algorithm still runs in $O(k(H+W))$ time on a block of width H and height W. Further storing of $D'[j,i]$ as a tuple increases the space by a multiplicative constant and therefore Algorithm 2 requires $O(H+W)$ space. □

We can now give the k-differences with transpositions result.

Theorem 3.11. *The k-differences problem with transpositions can be solved in $O(k \log m)$ time per character and $O(m)$ space in the PsR model.*

Proof. We have shown that k-differences with transpositions can be described by a $(k+1)$-bounded dynamic programming recurrence of the correct form and that an $O(nk)$ blockwise decomposable offline algorithm exists using $O(n)$ space

when run on a block of width n and height $m \leq n$. Therefore by Theorem 3.9, the result follows. □

3.3.3. The split and feed translation algorithm (Proof of Theorem 3.9)

Throughout this section we will consider an arbitrary approximate pattern matching problem defined by a recurrence relation $D[j, i]$ on a pattern P and a text T arriving online as defined above. We assume that this relation is k-bounded (Definition 2) and that an offline *blockwise decomposable* (Definition 3) algorithm, A, running in time $T(W, H)$ and using $S(W, H)$ space is provided. Here W and H are the width and height respectively of the dynamic programming block being computed. We give an non-amortised online algorithm using A as a black box which computes $D[m-1, i]$ as $T[i]$ arrives in time $O((T(n, m)/n) \log m + k)$ per character.

Consider the $m \times n$ dynamic programming table underlying the problem. Our algorithm splits the table by row into a number of *levels*, defined by a sequence of dividing rows, $r(\rho) = (1 - 4^{-\rho})m$ for $1 \leq \rho < s$. Here s is the largest integer such that $m/4^{s+1} > 6k$. For simplicity we let $r(0) = 0$ and $r(s) = m - 1$. In the case that $s < 3$, we compute each new column of the dynamic programming table directly in $O(m)$ time per arriving character online and non-amortised. As $s < 3$ we have that $k > m/(6 \times 4^4)$ and hence $O(m) \subseteq O((T(n, m)/n) \log m + k)$ as required by Theorem 3.9. Therefore wlog. we assume that $s \geq 3$.

Each level $0 < \rho < s$ computes row $r(\rho)$ of the dynamic programming table using the values of cells in row $r(\rho - 1)$ (and the relevant pattern and text sections). We show that computation is scheduled so that any cell value required by level ρ is either outputted by level $\rho - 1$ before it is needed or cannot affect the output (due to the k-bound). Each of these $O(\log m)$ levels will run at most eight copies of A in parallel. The final level s will compute all rows $r(s-1) = (1 - 4^{-(s-1)})m$ to $r(s) = m - 1$ online directly of which there are $O(k)$ by definition. The cell values on dividing rows are stored explicitly.

The algorithm for level 0. The zero-th level computes the values on row 1 of the dynamic programming table. As f can be computed in constant time, level 0 can compute cell $D[0, i]$ as symbol $T[i]$ arrives in constant time.

The algorithm for level $0 < \rho < s$. Each level splits the dynamic programming table further by columns into a number of rectangles, defined by a sequence of dividing columns given by

$$c_\rho(j) = \frac{7}{6} \times \frac{j}{8} \times \frac{m}{4^{\rho-1}} \text{ for all } j \geq 0\,.$$

The algorithm for level ρ operates as follows: Whenever a text character $T[c_\rho(j)]$ corresponding to a dividing column $c_\rho(j)$ arrives an instance of algorithm A is begun. The instance is run on the block $[r(\rho - 1) \ldots r(\rho)] \times [c_\rho(j - 8) \ldots c_\rho(j)]$ of the dynamic programming table. The cell values on the left block edge, $D[r(\rho - 1) \ldots r(\rho), c_\rho(j - 8)]$ are ignored and set to ∞ in the input to A. Also any cell values on the top block edge, $D[r(\rho - 1), c_\rho(j - 8) \ldots c_\rho(j)]$

that have not been outputted when computation begins are set to ∞ also. We show below that this does not affect correctness. Analogously to the local method, the computation is distributed evenly so that computation ends when $T[c_{\rho+1}(4j+1)]$ is received for levels $\rho \neq s-1$. For level $s-1$, computation ends when $T[h(4j)]$ is received (see level s algorithm description). Only the values of cells $D[r(\rho), c_\rho(j-1)\dots c_\rho(j)]$ outputted by A are kept (as others may be incorrect). Due to the overlap of the blocks, the value for each cell on row $r(\rho)$ is outputted by some block on level ρ.

The algorithm for level s. Level s will output $D[m-1, i]$ as $T[i]$ arrives. It will achieve this by computing the dynamic programming table from $r(s-1) = (1-4^{-(s-1)})m$ to $r(s) = m-1$ directly. Recall that s is the largest integer such that $m/4^s > 6k$. Therefore $r(s) - r(s-1) = m - (1-4^{-s})m = m/4^s$ and hence we can compute each sub-column in $O(k)$ time. However, level $(s-1)$ does not produce its output online so we must schedule the work carefully.

The final level again splits the dynamic programming table further by two sequences of dividing columns given by

$$c_s(j) = \frac{7}{6} \times \frac{j}{8} \times \frac{m}{4^{s-1}} \quad \text{and} \quad h(j) = \frac{7}{6} \times \frac{(2j+1)}{16} \times \frac{m}{4^{s-1}} \qquad \text{for all } j \geq 0\,.$$

Note that $c_s(j)$ is defined as before and $h(j)$ is half way between $c_s(j)$ and $c_s(j+1)$. The algorithm proceeds as follows: Whenever a text character $T[h(j-2)]$ corresponding to a dividing column $h(j-2)$ arrives we begin directly computing the dynamic programming table block $[r(s-1)\dots r(s)] \times [c_s(j-8)\dots c_s(j)]$. As each character from $T[h(j-2)]$ until $T[c_s(j-1)-1]$ arrives, we compute 14 sub-columns of the table block so that we finish computing $[r(s-1)\dots r(s)] \times [c_s(j-8)\dots c_s(j-1)-1]$ before $T[c_s(j-1)]$ arrives. When any characters $T[i]$ between $T[c_s(j-1)]$ and $T[c_s(j)]$ arrives we compute the sub-column $D[r(s-1)\dots r(s), i]$ and output $D[m-1, i]$ online. In the summary we refer to these two computation speeds as *quick* and *slow* respectively. Throughout we set all cells on the left boundary, $D[r(s-1)\dots r(s), c_s(j-8)]$ as well as any cells on the top boundary, $D[r(s), c_s(j-8)\dots c_s(j)]$ that have not been computed, to ∞.

Algorithm summary. We can now give a summary of the overall structure of the algorithm when character $T[i]$ arrives:

1. *Level* 0: Compute $D[0, i]$ in real-time.
2. *For all levels* $0 < \rho < s$:
   (a) If $i = c_\rho(j)$ for some j: Begin computation on the block, $D[r(\rho-1)\dots r(\rho), c_\rho(j-8)\dots c_\rho(j)]$
   (b) Continue computation on each of the at most 8 blocks which are currently being computed by level ρ.
   (c) If either ($\rho < s-1$ and $i = c_{\rho+1}(4j+1)$) or ($\rho = s-1$ and $i = h(4j)$): The block which began computation at $c_i(j)$ is completed.
3. *Level s*:

(a) If $i = h(j-2)$ for some j: Begin quickly computing the block, $D[r(s-1)\dots r(s), c_s(j-8)\dots c_s(j)]$
(b) Compute 14 sub-columns of each of the at most 7 blocks which are currently being computed *quickly* by level s.
(c) If $i = c_s(j-1)$ for some j: Change to slowly computing the block, $D[r(s-1)\dots r(s), c_s(j-8)\dots c_s(j)]$
(d) Compute the sub-column $D[r(s-1)\dots r(s), i]$ of the block being computed slowly and output $D[m-1, i]$ online.

Correctness of level $0 < \rho < s$. Consider an arbitrary block on level ρ, which begins computation at column $c_\rho(j)$. First observe that the text characters required have arrived and the pattern is known in advance. Therefore we only need to show that the values of cells $D[r(\rho), c_\rho(j-1)\dots c_\rho(j)]$ are independent of the cells incorrectly set to ∞ in the input. We begin by determining which cells on the top edge, $D[r(\rho-1), c_\rho(j-8)\dots c_\rho(j)]$, have not been outputted when $c_{\rho+1}(j)$ is received. In the case of level 1, all cells on this edge have trivially been outputted so we focus on a level $\rho > 1$. From the algorithm description we observe that a level $\rho-1$ block completes computation as $c_\rho(j)$ is received iff $j \mod 4 = 1$. Therefore in the worst case where $j \mod 4 = 0$, a level $\rho-1$ block ended computation at $c_\rho(j-3)$, outputting values, $D[r(\rho-1), c_\rho(j-6)\dots c_\rho(j-4)]$. Further, cells in the range $D[r(\rho-1), 1\dots c_\rho(j-6)]$ have already been outputted by previously computed blocks. Therefore, the correctness is concluded by Lemma 8 which shows that the values of cells $D[r(\rho), c_\rho(j-1)\dots c_\rho(j)]$ are independent of all other top and left block boundary cells by the k-boundedness of the recurrence relation. Note that the lemma shows that the output is also independent of $D[r(\rho-1), c_\rho(j-5)\dots c_\rho(j-4)]$ which is required for level s.

Lemma 8. *The values of cells* $D[r(\rho), c_\rho(j-1)\dots c_\rho(j)]$ *are independent of the values of cells* $D[r(\rho-1), c_\rho(j-5)\dots c_\rho(j)]$ *and* $D[r(\rho-1)\dots r(\rho), c_\rho(j-8)]$.

Proof. Consider an arbitrary cell $D[r(\rho), \alpha]$ with $c_\rho(j-1) \le \alpha \le c_\rho(j)$. Firstly, by k-boundedness, $D[r(\rho), \alpha]$ is independent of $D[r(\rho-1)\dots r(\rho), \alpha - k - y]$ for all $y > r(\rho) - r(\rho-1)$. Therefore, $D[r(\rho), \alpha]$ is independent of $D[r(\rho-1)\dots r(\rho), c_\rho(j-1)-k+r(\rho-1)-r(\rho)-y']$ for all $y' > 0$. However by definition, $r(\rho) - r(\rho-1) < m/4^{\rho-1}$. Further, as $m/4^{\rho+1} \ge m/4^{s+1} > 6k$, we have that $c_\rho(j-1) + r(\rho-1) - r(\rho) - k > c_\rho(j-1) - (97/96) \times (m/4^{\rho+1}) > c_\rho(j-8)$. Hence, as required, $D[r(\rho), \alpha]$ is independent of $D[r(\rho-1)\dots r(\rho), c_\rho(j-8)]$. Secondly, by k-boundedness, $D[r(\rho), \alpha]$ is independent of $D[r(\rho) - k - y, c_\rho(j-5)\dots c_\rho(j)]$ for all $y > \alpha - c_\rho(j-5)$. Therefore, $D[r(\rho), \alpha]$ is independent of $D[r(\rho) - k + c_\rho(j-5) - \alpha - y', c_\rho(j-5)\dots c_\rho(j)]$ for all $y' > 0$. However by definition, $\alpha - c_\rho(j-5) \le c_\rho(j) - c_\rho(j-5) = (7/6) \times (5/8) \times (m/4^{\rho-1})$ and as $k < m/(6\times 4^{\rho+1})$, thus $r(\rho)-k+c_\rho(j-5)-\alpha > r(\rho)-(3/4)\times(m/4^{\rho-1}) \ge r(\rho-1)$. Hence, as required, $D[r(\rho), \alpha]$ is independent of $D[r(\rho-1), c_\rho(j-5)\dots c_\rho(j)]$. □

Running time of level $0 < \rho < s$. Let $W = c_\rho(j) - c_\rho(j-8) + 1 = (7/6) \times (m/4^{\rho-1}) + 1$ and H be the width and height respectively of a block on level

ρ. The work for this block is distributed evenly over text characters $c_\rho(j)$ to $c_{\rho+1}(4j+1)$. However, $c_\rho(j) = c_{\rho+1}(4j)$ and thus $c_{\rho+1}(4j+1) - c_{\rho+1}(4j) + 1 = (7/6) \times (1/8) \times (m/4^\rho) + 1 \geq W/32$. Similarly for $\rho = s-1$, the work is distributed evenly over the at least $W/16$ text arrivals from $c_{s-1}(j)$ to $h(4j)$. As each text character arrives, work is done on a constant number of blocks and thus in both cases, $O(T(W,H)/W)$ work is performed per character. As A must read the entire input, $T(W,H) \in \Omega(W+H)$ and therefore $O(T(W,m)/W) \in O(T(n,m)/n)$. By inspecting the algorithm, observe that to compute a block ending at column $c_\rho(j)$ we only need access to at the last W cells on row $r(\rho)$ and the last W characters of T. Thus we use $O(W + S(W,H))$ space for level ρ where $S(W,H)$ is the space used by algorithm A on a block of size $W \times H$.

Correctness of level s. By Lemma 8, incorrectly setting the left boundary to ∞ cannot affect calculations. Further observe that as before, no text character is required before it is available. From the algorithm description we observe that a level $s-1$ block completes computation as $c_s(j-1)$ is received iff $j \mod 4 = 2$, analogous to before. Consider the last $s-1$ block to complete before $h(j-2)$ is received (and computation begins). In the worst case where $j \mod 4 = 0$, this $s-1$ block ended computation at $h(j-5)$ outputting values $D[r(s-1), c_s(j-7) \dots c_s(j-5)]$. However, by Lemma 8, the values $D[r(s-1), c_s(j-5) \dots c_s(j)]$ are independent of the cells $D[m-1, c_s(j-1) \dots c_s(j)]$ so the algorithm outputs correctly.

Running time of level s. Level s computes a constant number of columns when any character arrives, requiring a total of $O(k)$ time and $O(k) \in O(m)$ space. Summing across all $O(\log m)$ levels, we require at most $O(T(n,m) \log m/n + k)$ time per character non-amortised. As algorithm A must use $\Omega(n+m)$ space (to store the input and output) then our method uses $O(S(m,m))$ space where $S(m,m)$ is the space requirement of A on an m by m block.

4. Acknowledgements.

The authors would like to thank Benny and Ely Porat for many helpful discussions at an early stage of this work. The authors would also like to thank the anonymous reviewer for pointing out the link between self-normalised matching and L_2-rearrangement distances.

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. *J. comput. syst. sci*, 75(6):359–370, 2009.

[2] Amihood Amir, Yonatan Aumann, Moshe Lewenstein, and Ely Porat. Function matching. *SIAM J. Comput.*, 35(5):1007–1022, 2006.

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

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

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

[6] Peter Clifford and Raphaël Clifford. Self-normalised distance with don't cares. In *CPM '07*, pages 63–70, 2007.

[7] Raphaël Clifford, Klim Efremenko, Benny Porat, and Ely Porat. A black box for online approximate pattern matching. In *CPM '08*, pages 143–151, 2008.

[8] Maxime Crochemore, Gad M. Landau, and Michal Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted cost matrices. In *SODA '02*, pages 679–688, 2002.

[9] Michael J. Fischer and Larry J. Stockmeyer. Fast on-line integer multiplication. In *STOC '73*, pages 67–72, New York, NY, USA, 1973. ACM.

[10] Zvi Galil. String matching in real time. *J. ACM*, 28(1):134–149, 1981.

[11] D. E. Knuth, J. H. Morris, and V. B. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:323–350, 1977.

[12] Gad M. Landau, E. W. Myers, and J. P. Schmidt. Incremental string comparison. *SIAM J. Comput.*, 27(2):557–582, 1998.

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

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

[15] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *SODA '90*, pages 319–327, 1990.

[16] Esko Ukkonen. Algorithms for approximate string matching. *Inf. Control*, 64(1-3):100–118, 1985.