Order-Preserving Pattern Matching with k Mismatches

Paweł Gawrychowski[1] and Przemysław Uznański[2,⋆]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[2] LIF, CNRS and Aix-Marseille Université, Marseille, France

Abstract. We study a generalization of the order-preserving pattern matching recently introduced by Kubica et al. (Inf. Process. Let., 2013) and Kim et al. (submitted to Theor. Comp. Sci.), where instead of looking for an exact copy of the pattern, we only require that the relative order between the elements is the same. In our variant, we additionally allow up to k mismatches between the pattern of length m and the text of length n, and the goal is to construct an efficient algorithm for small values of k. Our solution detects an order-preserving occurrence with up to k mismatches in $\mathcal{O}(n(\log\log m + k\log\log k))$ time.

1 Introduction

Order-preserving pattern matching, recently introduced in [9] and [10], and further considered in [4], is a variant of the well-known pattern matching problem, where instead of looking for a fragment of the text which is identical to the given pattern, we are interested in locating a fragment which is order-isomorphic with the pattern. Two sequences over integer alphabet are *order-isomorphic* if the relative order between any two elements at the same positions in both sequences is the same. Similar problems have been extensively studied in a slightly different setting, where instead of a fragment, we are interested in a (not necessarily contiguous) subsequence. For instance, pattern avoidance in permutations was of much interest.

For the order-preserving pattern matching, both [9] and [10] present an $\mathcal{O}(n + m\log m)$ time algorithm, where n is the length of the text, and m is the length of the pattern. Actually, the solution given by [10] works in $\mathcal{O}(n+\mathrm{sort}(m))$ time, where $\mathrm{sort}(m)$ is the time required to sort a sequence of m numbers. Furthermore, efficient algorithms for the version with multiple patterns are known [4]. Also, a generalization of suffix trees in the order-preserving setting was recently considered [4], and the question of constructing a forward automaton allowing efficient pattern matching and developing an average-case optimal pattern matching algorithm was studied [1].

Given that the complexity of the exact order-preserving pattern matching seems to be already settled, a natural direction is to consider its approximate version.

⋆ This work was started while the author was a PhD student at Inria Bordeaux Sud-Ouest, France.

A.S. Kulikov, S.O. Kuznetsov, and P. Pevzner (Eds.): CPM 2014, LNCS 8486, pp. 130–139, 2014.

Such direction was successfully investigated for the related case of *parametrized pattern matching* in [6], where an $\mathcal{O}(nk^{1.5} + mk \log m)$ time algorithm was given for parametrized matching with k mismatches.

We consider a relaxation of order-preserving pattern matching, which we call *order-preserving pattern matching with k mismatches*. Instead of requiring that the fragment we seek is order-isomorphic with the pattern, we are allowed to first remove k elements at the corresponding positions from the fragment and the pattern, and then check if the remaining two sequences are order-isomorphic. In such setting, it is relatively straightforward to achieve running time of $\mathcal{O}(nm \log \log m)$, where n is the length of the text, and m is the length of the pattern. Such complexity might be unacceptable for long patterns, though, and we aim to achieve complexity of the form $\mathcal{O}(nf(k))$. In other words, we would like our running time to be close to linear if the bound on the number of mismatches is very small. We construct a deterministic algorithm with $\mathcal{O}(n(\log \log m + k \log \log k))$ running time. At a very high level, our solution is similar to the one given in [6]. We show how to filter the possible starting positions so that a position is either eliminated in $\mathcal{O}(f(k))$ time, or the structure of the fragment starting there is simple, and we can verify the occurrence in $\mathcal{O}(f(k))$ time. The details are quite different in our setting, though.

A different variant of approximate order-preserving pattern matching could be that we allow to remove k elements from the fragment, and k elements from the pattern, but don't require that they are at the same positions. Then we get order-preserving pattern matching with k errors. Unfortunately, such modification seems difficult to solve in polynomial time: even if the only allowed operation is removing k elements from the fragment, the problem becomes NP-complete [3].

2 Overview of the Algorithm

Given a text $(t_1, \ldots, t_n)$ and a pattern $(p_1, \ldots, p_m)$, we want to locate an order-preserving occurrence with at most k mismatches of the pattern in the text. Such occurrence is a fragment $(t_i, \ldots, t_{i+m-1})$ with the property that if we ignore the elements at some up to corresponding k positions in the fragment and the pattern, the relative order of the remaining elements is the same in both of them.

The above definition of the problem is not very convenient to work with, hence we start with characterising k-isomorphic sequences using the language of subsequences in Lemma 1. This will be useful in some of the further proofs and also gives us a polynomial time solution for the problem, which simply considers every possible i separately. To improve on this naive solution, we need a way of quickly eliminating some of these starting positions. For this we define the signature $S(a_1, \ldots, a_m)$ of a sequence $(a_1, \ldots, a_m)$, and show in Lemma 3 that the Hamming distance between the signatures of two k-isomorphic sequences cannot be too large. Hence such distance between $S(t_i, \ldots, t_{i+m-1})$ and $S(p_1, \ldots, p_m)$ can be used to filter some starting positions where a match cannot happen.

In order to make the filtering efficient, we need to maintain $S(t_i, \ldots, t_{i+m-1})$ as we increase i, i.e., move a window of length m through the text. For this we

first provide in Lemma 4 a data structure which, for a fixed word, allows us to maintain a word of a similar length under changing the letters, so that we can quickly generate the first k mismatches between subwords of the current and the fixed word. The structure is based on representing the current word as a concatenation of subwords of the fixed word. Then we observe that increasing i by one changes the current signature only slightly, which allows us to apply the aforementioned structure to maintain $S(t_i, \ldots, t_{i+m-1})$ as shown in Lemma 5. Therefore we can efficiently eliminate all starting positions for which the Hamming distance between the signatures is too large.

For all the remaining starting positions, we reduce the problem to computing the heaviest increasing subsequence, which is a weighted version of the well-known longest increasing subsequence, in Lemma 6. The time taken by the reduction depends on the Hamming distance, which is small as otherwise the position would be eliminated in the previous step. Finally, such weighted version of the longest increasing subsequence can be solved efficiently as shown in Lemma 7. Altogether these results give an algorithm for order-preserving pattern matching with k with the cost of processing a single i depending mostly on k.

An implicit assumption in this solution is that there are no repeated elements in neither the text nor the pattern. The assumption can be removed without increasing the time complexity by carefully modifying all parts of the algorithm. Some of these changes are not completely trivial, for example the definition of a signature becomes more involved, which in turn makes the proofs more complicated. Because of the limited space we defer the details (and a few proofs) to the full version.

3 Preliminaries

We consider strings over an integer alphabet, or in other words sequences of integers. Two such sequences are *order-isomorphic* (or simply *isomorphic*), denoted by $(a_1, \ldots, a_m) \sim (b_1, \ldots, b_m)$, when $a_i \leq a_j$ iff $b_i \leq b_j$ for all i, j. We will also use the usual equality of strings. Whenever we are talking about sequences, we are interested in the relative order between their elements, and whenever we are talking about strings consisting of characters, the equality of elements is of interest to us. For two strings s and t, their *Hamming distance* $\mathrm{H}(s, t)$ is simply the number of positions where the corresponding characters differ.

Given a text $(t_1, \ldots, t_n)$ and a pattern $(p_1, \ldots, p_m)$, the *order-preserving pattern matching* problem is to find i such that $(t_i, \ldots, t_{i+m-1}) \sim (p_1, \ldots, p_m)$. We consider its approximate version, i.e., order-preserving pattern matching with k mismatches. We define two sequences *order-isomorphic with k mismatches*, denoted by $(a_1, \ldots, a_m) \overset{k}{\sim} (b_1, \ldots, b_m)$, when we can select (up to) k indices $1 \leq i_1 < \ldots < i_k \leq m$, and remove the corresponding elements from both sequences so that the resulting two new sequences are isomorphic, i.e., $a_j \leq a_{j'}$ iff $b_j \leq b_{j'}$ for any $j, j' \notin \{i_1, \ldots, i_k\}$. In *order-preserving pattern matching with k mismatches* we want i such that $(t_i, \ldots, t_{i+m-1}) \overset{k}{\sim} (p_1, \ldots, p_m)$, see Fig. 1.

Our solution works in the word RAM model, where n integers can be sorted in $\mathcal{O}(n \log \log n)$ time [5], and we can implement dynamic dictionaries using

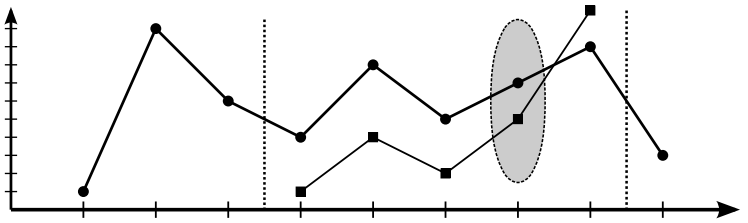

Fig. 1. $[1, 4, 2, 5, 11]$ occurs in $[1, 10, 6, 4, 8, 5, 7, 9, 3]$ (at position 4) with 1 mismatch

van Emde Boas trees. In the restricted comparison model, where we can only compare the integers, all $\log \log$ in our complexities increase to $\log$.

We assume that integers in any sequence are all distinct. Such assumption was already made in one of the papers introducing the problem [9], with a justification that we can always perturb the input to ensure this (or, more precisely, we can consider pairs consisting of a number and its position). In some cases this can change the answer, though[1]. Nevertheless, using a more complicated argument, given in the full version, we can generalize our solution to allow the numbers to repeat. Another simplifying assumption that we make in designing our algorithm is that $n \leq 2m$. We can do so using a standard trick of cutting the text into overlapping fragments of length $2m$ and running the algorithm on each such fragment separately, which preserves all possible occurrences.

4 The Algorithm

First we translate k-isomorphism into the language of subsequences.

Lemma 1. $(a_1, \ldots, a_m) \overset{k}{\nsim} (b_1, \ldots, b_m)$ *iff there exist* $i_1, \ldots, i_{m-k}$ *such that* $a_{i_1} < \ldots < a_{i_{m-k}}$ *and* $b_{i_1} < \ldots < b_{i_{m-k}}$.

The above lemma implies an inductive interpretation of k-isomorphism useful in further proofs and a fast method for testing k-isomorphism.

Proposition 1. *If* $(a_1, \ldots, a_m) \overset{k+1}{\sim} (b_1, \ldots, b_m)$ *then there exists* $(a'_1, \ldots, a'_m)$ *such that* $(a_1, \ldots, a_m) \overset{1}{\sim} (a'_1, \ldots, a'_m)$ *and* $(a'_1, \ldots, a'_m) \overset{k}{\nsim} (b_1, \ldots, b_m)$.

Lemma 2. $(a_1, \ldots, a_m) \overset{k}{\nsim} (b_1, \ldots, b_m)$ *can be checked in time* $\mathcal{O}(m \log \log m)$.

Proof. Let π be the sorting permutation of $(a_1, \ldots, a_m)$. Such permutation can be found in time $\mathcal{O}(m \log \log m)$. Let $(b'_1, \ldots, b'_m)$ be a sequence defined by setting $b'_i := b_{\pi(i)}$. Then, by Lemma 1, $(a_1, \ldots, a_m) \overset{k}{\nsim} (b_1, \ldots, b_m)$ iff there exists an increasing subsequence of b' of length $m - k$. Existence of such a subsequence can be checked in time $\mathcal{O}(m \log \log m)$ using a van Emde Boas tree [7]. □

[1] More precisely, it might make two non-isomorphic sequences isomorphic, but not the other way around.

By applying the above lemma to each of the possible occurrences separately, we can already solve order-preserving pattern matching with k mismatches in time $\mathcal{O}(nm \log\log m)$. However, our goal is to develop a faster $\mathcal{O}(nf(k))$ time algorithm. For this we cannot afford to verify every possible position using Lemma 2, and we need a closer look into the structure of the problem.

The first step is to define the *signature* of a sequence $(a_1, \ldots, a_m)$. Let $\mathrm{pred}(i)$ be the position where the predecessor of a_i among $\{a_1, \ldots, a_m\}$ occurs in the sequence (or 0, if a_i is the smallest element). Then the signature $S(a_1, \ldots, a_m)$ is a new sequence $(1 - \mathrm{pred}(1), \ldots, m - \mathrm{pred}(m))$ (a simpler version, where the new sequence is $(\mathrm{pred}(1), \ldots, \mathrm{pred}(m))$, was already used to solve the exact version). The signature clearly can be computed in time $\mathcal{O}(m \log\log m)$ by sorting. While looking at the signatures is not enough to determine if two sequences are k-isomorphic, in some cases it is enough to detect that they are not.

Lemma 3. *If* $(a_1, \ldots, a_m) \overset{k}{\sim} (b_1, \ldots, b_m)$*, then the Hamming distance between* $S(a_1, \ldots, a_m)$ *and* $S(b_1, \ldots, b_m)$ *is at most* $3k$*.*

Proof. We apply induction on the number of mismatches k.

For $k = 0$, $(a_1, \ldots, a_m) \sim (b_1, \ldots, b_m)$ iff $S(a_1, \ldots, a_m) = S(b_1, \ldots, b_m)$, so the Hamming distance is clearly zero.

Now we proceed to the inductive step. If $(a_1, \ldots, a_m) \overset{k+1}{\sim} (b_1, \ldots, b_m)$, then due to Proposition 1, there exists $(a'_1, \ldots, a'_m)$, such that $(a'_1, \ldots, a'_m) \overset{k}{\sim} (b_1, \ldots, b_m)$ and $(a_1, \ldots, a_m) \overset{1}{\sim} (a'_1, \ldots, a'_m)$. Second constraint is equivalent (by application of Lemma 1) to existence of such i, that $(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_m) \sim (a'_1, \ldots, a'_{i-1}, a'_{i+1}, \ldots, a'_m)$.

We want to upperbound the Hamming distance between $S(a_1, \ldots, a_m)$ and $S(a'_1, \ldots, a'_m)$. Let j, j' be indices such that a_j is the direct predecessor of a_i and $a_{j'}$ is the direct successor of a_i, both taken from the set $\{a_1, \ldots, a_m\}$. Similarly, let ℓ, ℓ' be such indices, that a'_ℓ is the direct predecessor, and $a'_{\ell'}$ is the direct successor of a'_i, both taken from the set $\{a'_1, \ldots, a'_m\}$. That is,

$$\ldots < a_j < a_i < a_{j'} < \ldots$$

is the sorted version of $(a_1, \ldots, a_m)$, and

$$\ldots < a'_\ell < a'_i < a'_{\ell'} < \ldots$$

is the sorted version of $(a'_1, \ldots, a'_m)$. The signatures $S(a_1, \ldots, a_m)$ and $S(a'_1, \ldots, a'_m)$ differ on at most 3 positions: j', ℓ', and i. Thus $\mathrm{H}(S(a_1, \ldots, a_m), S(b_1, \ldots, b_m))$ can be upperbounded by

$$\mathrm{H}(S(a_1, \ldots, a_m), S(a'_1, \ldots, a'_m)) + \mathrm{H}(S(a'_1, \ldots, a'_m), S(b_1, \ldots, b_m)) \leq 3k + 3,$$

which ends the inductive step. □

Our algorithm iterates through $i = 1, 2, 3, \ldots$ maintaining the signature of the current $(t_i, \ldots, t_{i+m-1})$. Hence the second step is that we develop in the next two lemmas a data structure, which allows us to store $S(t_i, \ldots, t_{i+m-1})$, update it efficiently after increasing i by one, and compute its Hamming distance to $S(p_1, \ldots, p_m)$.

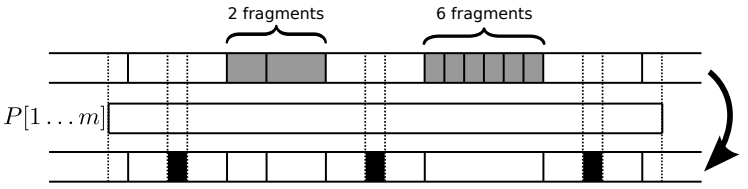

Fig. 2. Updating the representation. Black boxes represent mismatches, gray areas are full fragments between mismatches. Fragments are either left untouched (on the left), or compressed into a single new one (on the right).

Lemma 4. *Given a string $S^P[1..m]$, we can maintain a string $S^T[1..2m]$ and perform the following operations:*

1. *replacing any character $S^T[x]$ in amortized time $\mathcal{O}(\log\log m)$,*
2. *generating the first $3k$ mismatches between $S^T[i..(i+m-1)]$ and $S^P[1..m]$ in amortized time $\mathcal{O}(k+\log\log m)$.*

The structure is initialized in time $\mathcal{O}(m\log\log m)$.

Proof. We represent the current $S^T[1..2m]$ as a concatenation of a number of fragments. Each fragment is a subword of S^P (possibly single letter) or a special character \$ not occurring in S^P. The starting positions of the fragments are kept in a van Emde Boas tree, and additionally each fragment knows its successor and predecessor. In order to bound the amortized complexity of each operation, we maintain an invariant that every element of the tree has 2 credits available, with one credit being worth $\mathcal{O}(\log\log m)$ time. We assume that given any two substrings of S^P, we can compute their longest common prefix in $\mathcal{O}(1)$ time. This is possible after $\mathcal{O}(m)$ preprocessing [2,8].

We initialize the structure by partitioning S^T into $2m$ single characters. The cost of initialization, including allocating the credits, is $\mathcal{O}(m\log\log m)$.

Replacing $S^T[x]$ with a new character c starts with locating the fragment w containing the position i using the tree. If w is a single character, we replace it with the new one. If w is a longer subword $w[i..j]$ of S^P, and we need to replace its ℓ-th character, we first split w into three fragments $w[i..(i+\ell-1)]$, $w[i+\ell]$, $w[(i+\ell+1)..j]$. In both cases we spend $\mathcal{O}(\log\log m)$ time, including the cost of inserting the new elements and allocating their credits.

Generating the mismatches begins with locating the fragment corresponding to the position i. Then we scan the representation from left to right starting from there. Locating the fragment takes $\mathcal{O}(\log\log m)$ time, but traversing can be done in $\mathcal{O}(1)$ time per each step, as we can use the information about the successor of each fragment. We will match S^P with the representation of S^T while scanning. This is done using constant time longest common prefix queries. Each such query allows us to either detect a mismatch, or move to the next fragment. Whenever we find a mismatch, if the part of the text between the previous mismatch (or the beginning of the window) and the current mismatch contains at least 3 full

fragments, we replace them with a single fragment, which is the corresponding subword of S^P. If there are less than 3 full fragments, we keep the current representation intact, see Fig. 2. We stop the scanning after reaching $(3k+1)$-th mismatch, or after the whole window was processed, whichever comes first. By a standard argument, the amortized cost of processing a single mismatch is $\mathcal{O}(1)$, so we need $\mathcal{O}(k + \log\log m)$ time to generate all the mismatches. □

Lemma 5. *Given a pattern $(p_1, \ldots, p_m)$ and a text $(t_1, \ldots, t_{2m})$, we can maintain an implicit representation of the current signature $S(t_i, \ldots, t_{i+m-1})$ and perform the following operations:*

1. *increasing i by one in amortized time $\mathcal{O}(\log\log m)$,*
2. *generating the first $3k$ mismatches between $S(p_1, \ldots, p_m)$ and $S(t_i, \ldots, t_{i+m-1})$ in time $\mathcal{O}(k + \log\log m)$.*

The structure is initialized in time $\mathcal{O}(m \log\log m)$.

Proof. First we construct $S(p_1, \ldots, p_m)$ in time $\mathcal{O}(m \log\log m)$ by sorting. Whenever we increase i by one, just a few characters of $S(t_i, \ldots, t_{i+m-1}) = (s_1, \ldots, s_m)$ need to be modified. The new signature can be created by first removing the first character s_1, appending a new character s_{m+1}, and then modifying the characters corresponding to the successors of t_i and t_{i+m}. By maintaining all $t_i, \ldots, t_{i+m-1}$ in a van Emde Boas tree (we can rename the elements so that $t_i \in \{1, \ldots, 2m\}$ by sorting) we can calculate both s_{m+1} and the characters which needs to be modified in $\mathcal{O}(\log\log m)$ time. Current $S(t_i, \ldots, t_{i+m-1})$ is stored using Lemma 4. We initialize $S^T[1..2m]$ to be $S(t_1, \ldots, t_m)$ concatenated with m copies of, say, 0. After increasing i by one, we replace $S^T[i]$, $S^T[i+m]$ and possibly two more characters in $\mathcal{O}(\log\log m)$ time. Generating the mismatches is straightforward using Lemma 4. □

Now our algorithm first uses Lemma 5 to quickly eliminate the starting positions i such that the Hamming distance between the corresponding signatures is large. For the remaining starting positions, we reduce checking if $(t_i, \ldots, t_{i+m-1}) \overset{k}{\sim} (p_1, \ldots, p_m)$ to a weighted version of the well-known longest increasing subsequence problem on at most $3(k+1)$ elements. In the weighted variant, which we call *heaviest increasing subsequence*, the input is a sequence $(a_1, \ldots, a_\ell)$ and weight w_i of each element a_i, and we look for an increasing subsequence with the largest total weight, i.e., for $1 \le i_1 < \ldots < i_s \le \ell$ such that $a_{i_1} < \ldots < a_{i_s}$ and $\sum_j w_{i_j}$ is maximized.

Lemma 6. *Assuming random access to $(a_1, \ldots, a_m)$, the sorting permutation π_b of $(b_1, \ldots, b_m)$, and the rank of every b_i in $\{b_1, \ldots, b_m\}$, and given ℓ positions where $S(a_1, \ldots, a_m)$ and $S(b_1, \ldots, b_m)$ differ, we can reduce in $\mathcal{O}(\ell \log\log \ell)$ time checking if $(a_1, \ldots, a_m) \overset{k}{\sim} (b_1, \ldots, b_m)$ to computing the heaviest increasing subsequence on at most $\ell + 1$ elements.*

Proof. Let $d_1, \ldots, d_\ell$ be the positions where $S(a_1, \ldots, a_m)$ and $S(b_1, \ldots, b_m)$ differ. From the definition of a signature, for any other position i the predecessors

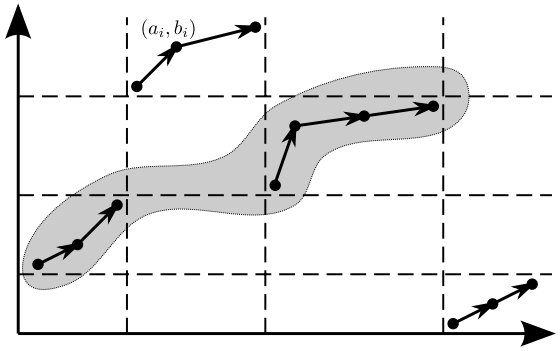

Fig. 3. Partition into maximal paths. The heaviest increasing subsequence is marked.

of a_i and b_i in their respective sequences are at the same position j, which we denote by $j \to i$. For any given i, $j \to i$ for at most one j. Similarly, for any given j, $j \to i$ for at most one i, because the only such i corresponds to the successor of, say, a_j in its sequence. Consider a partition of the set of all positions into maximal *paths* of the form $j_1 \to \ldots \to j_k$ (see Fig. 3). Such partition is clearly unique, and furthermore the first element of every path is one of the positions where the signatures differ (except one possible path starting with the position corresponding to the smallest element). Hence there are at most $\ell + 1$ paths, and we denote by I_j the path starting with d_j. If the smallest element occurs at the same position in both sequences, we additionally denote this position by d_0, and call the path starting there I_0 (we will assume that this is always the case, which can be ensured by appending $-\infty$ to both sequences).

Recall that our goal is to check if $(a_1, \ldots, a_m) \overset{k}{\sim} (b_1, \ldots, b_m)$. For this we need to check if there exist $i_1, \ldots, i_{m-k}$ such that $a_{i_1} < \ldots < a_{i_{m-k}}$ and $b_{i_1} < \ldots < b_{i_{m-k}}$. Alternatively, we could compute the largest s for which there exist a solution $i_1, \ldots, i_s$ such that $a_{i_1} < \ldots < a_{i_s}$ and $b_{i_1} < \ldots < b_{i_s}$. We claim that one can assume that for each path I either none of its elements are among $i_1, \ldots, i_s$, or all of its elements are there. We prove this in two steps.

1. If $i_k \in I$ and $i_k \to j$, then without losing the generality $i_{k+1} = j$. Assume otherwise, so $i_{k+1} \neq j$ or $k = s$. Recall that it means that a_j is the successor of a_{i_k} and b_j is the successor of b_{i_k}. Hence $a_{i_k} < a_j$ and $b_{i_k} < b_j$. If $k = s$ we can extend the current solution by appending j. Otherwise $a_j < a_{i_{k+1}}$ and $b_j < b_{i_{k+1}}$, so we can extend the solution by inserting j between i_k and i_{k+1}.
2. If $i_k \in I$ and $j \to i_k$, then without losing the generality $i_{k-1} = j$. Assume otherwise, so $i_{k-1} \neq j$ or $k = 1$. Similarly as in the previous case, a_j is the predecessor of a_{i_k} and b_j is the predecessor of b_{i_k}. Hence $a_j < a_{i_k}$ and $b_j < b_{i_k}$. If $k = 1$ we can extend the current solution by prepending j. Otherwise $a_{i_{k+1}} < a_j$ and $b_{i_{k+1}} < b_j$, so we can insert j between i_{k-1} and i_k.

Now let the weight of a path I be its length $|I|$. From the above reasoning we know that the optimal solution contains either no elements from a path, or all

of its elements. Hence if we know which paths contain the elements used in the optimal solution, we can compute s as the sum of the weights of these paths. Additionally, if we take such optimal solution, and remove all but the first element from every path, we get a valid solution. Hence s can be computed by choosing some solution restricted only to $d_0, \ldots, d_\ell$, and then summing up weights of the corresponding paths. It follows that computing the optimal solution can be done, similarly as in the proof of Lemma 2, by finding an increasing subsequence. We define a new weighted sequence $(a'_0, \ldots, a'_\ell)$ by setting $a'_j = b_{\pi_b(d_j)}$ and choosing the weight of a'_j to be $|I_j|$. Then an increasing subsequence of $(a'_0, \ldots, a'_\ell)$ corresponds to a valid solution restricted to $d_0, \ldots, d_\ell$, and moreover the weight of the heaviest such subsequence is exactly s. In other words, we can reduce our question to computing the heaviest increasing subsequence.

Finally, we need to analyze the complexity of our reduction. Assuming random access to both $(a_1, \ldots, a_m)$ and π_b, we can construct $(a'_0, \ldots, a'_\ell)$ in time $\mathcal{O}(\ell)$. Computing the weight of every a'_j is more complicated. We need to find every $|I_j|$ without explicitly constructing the paths. For every d_j we can retrieve the rank r_j of its corresponding element in $\{b_1, \ldots, b_m\}$. Then I_j contains d_j and all i such that the predecessor of b_i among $\{b_{d_0}, \ldots, b_{d_\ell}\}$ is b_{d_j}. Hence $|I_j|$ can be computed by counting such i. This can be done by locating the successor $b_{d_{j'}}$ of b_{d_j} in $\{b_{d_0}, \ldots, b_{d_\ell}\}$ and returning $r_{d_{j'}} - r_{d_j} - 1$ (if the successor does not exist, $m - r_{d_j}$). To find all these successors, we only need to sort $\{b_{d_0}, \ldots, b_{d_\ell}\}$, which can, again, be done in time $\mathcal{O}(\ell \log\log \ell)$. □

Lemma 7. *Given a sequence of ℓ weighted elements, we can compute its heaviest increasing subsequence in time $\mathcal{O}(\ell \log\log \ell)$.*

Proof. Let the sequence be $(a_1, \ldots, a_\ell)$, and denote the weight of a_i by w_i. We will describe how to compute the weight of the heaviest increasing subsequence, reconstructing the subsequence itself will be straightforward. At a high level, for each i we want to compute the weight r_i of the heaviest increasing subsequence ending at a_i. Observe that $r_i = w_i + \max\{r_j : j < i \text{ and } a_j < a_i\}$, where we assume that $a_0 = -\infty$ and $r_0 = 0$. We process $i = 1, \ldots, \ell$, so we need a dynamic structure where we could store all already computed results r_j so that we can select the appropriate one efficiently. To simplify the implementation of this structure, we rename the elements in the sequence so that $a_i \in \{1, \ldots, \ell\}$. This can be done in $\mathcal{O}(\ell \log\log \ell)$ time by sorting. Then the dynamic structure needs to store n values $v_1, \ldots, v_n$, all initialized to $-\infty$ in the beginning, and implement two operations:

1. increase any v_k,
2. given k, return the maximum among $v_1, \ldots, v_k$.

Then to compute r_i we first find the maximum among $v_1, \ldots, v_{a_i - 1}$, and afterwards update v_{a_i} to be r_i. Both operations can be implemented in amortized time $\mathcal{O}(\log\log \ell)$ using a van Emde Boas tree. □

By combining the above ingredients (and, as mentioned before, cutting the input into overlapping fragments of length $2m$) we obtain the following result.

Theorem 1. *Order-preserving pattern matching with k mismatches can be solved in time $\mathcal{O}(n(\log\log m + k\log\log k))$, where n is the length of the text and m is the length of the pattern.*

5 Conclusions

Recall that the complexity of our solution is $\mathcal{O}(n(\log\log m + k\log\log k))$. Given that it is straightforward to prove a lower bound of $\Omega(n + m\log m)$ in the comparison model, and that for $k = 0$ one can achieve $\mathcal{O}(n + \mathrm{sort}(m))$ time [10], a natural question is whether achieving $\mathcal{O}(nf(k)) + \mathcal{O}(m\ \mathrm{polylog}(m))$ time is possible. Finally, even though the version with k errors seems hard (see the introduction), there might be an $\mathcal{O}(nf(k))$ time algorithm, with $f(k)$ being an exponential function.

References

1. Belazzougui, D., Pierrot, A., Raffinot, M., Vialette, S.: Single and multiple consecutive permutation motif search. In: Cai, L., Cheng, S.-W., Lam, T.-W. (eds.) ISAAC 2013. LNCS, vol. 8283, pp. 66–77. Springer, Heidelberg (2013)
2. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
3. Bose, P., Buss, J.F., Lubiw, A.: Pattern matching for permutations. Inf. Process. Lett. 65(5), 277–283 (1998)
4. Crochemore, M., et al.: Order-preserving incomplete suffix trees and order-preserving indexes. In: Kurland, O., Lewenstein, M., Porat, E. (eds.) SPIRE 2013. LNCS, vol. 8214, pp. 84–95. Springer, Heidelberg (2013)
5. Han, Y.: Deterministic sorting in $O(n\log\log n)$ time and linear space. In: Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing, STOC 2002, pp. 602–608. ACM, New York (2002)
6. Hazay, C., Lewenstein, M., Sokol, D.: Approximate parameterized matching. ACM Transactions on Algorithms 3(3) (2007)
7. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest common subsequences. Commun. ACM 20(5), 350–353 (1977)
8. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM 53(6), 918–936 (2006)
9. Kim, J., Eades, P., Fleischer, R., Hong, S.H., Iliopoulos, C.S., Park, K., Puglisi, S.J., Tokuyama, T.: Order preserving matching. CoRR abs/1302.4064 (2013)
10. Kubica, M., Kulczyński, T., Radoszewski, J., Rytter, W., Waleń, T.: A linear time algorithm for consecutive permutation pattern matching. Inf. Process. Lett. 113(12), 430–433 (2013)