Single and multiple consecutive permutation motif search

Djamal Belazzougui[⋆], Adeline Pierrot[⋆⋆], Mathieu Raffinot[⋆ ⋆ ⋆], and Stéphane Vialette[†]

Abstract: Let t be a permutation (that shall play the role of the *text*) on $[n]$ and a motif p be a sequence of m distinct integer(s) of $[n]$, $m \leq n$. The motif p occurs in t in position i if and only if $p_1 \dots p_m$ is order-isomorphic to $t_i \dots t_{i+m-1}$, that is, for all $1 \leq k < \ell \leq m$, $p_k > p_\ell$ if and only if $t_{i+k-1} > t_{i+\ell-1}$. Searching for a motif p in a text t consists in identifying all occurrences of p in t. We first present a forward automaton which allows us to search for p in t in $O(m^2 \log\log m + n)$ time. We then introduce a Morris-Pratt automaton representation of the forward automaton which allows us to reduce this complexity to $O(m \log\log m + n)$ at the price of an additional amortized constant term. The latter automaton occupies $O(m)$ space. We then extend the problem to search for a set of motifs and exhibit a specific Aho-Corasick like algorithm. Next we present a sub-linear average case search algorithm running in $O\left(\frac{m \log m}{\log\log m} + \frac{n \log m}{m \log\log m}\right)$ time, that we eventually prove to be optimal on average.

1 Introduction

Two sequences of distinct integers are *order-isomorphic* if the permutations required to sort them are the same. A sequence p is said to be a *motif* (or *occurs*) within a sequence t if t has a *subsequence* that is order-isomorphic to p. Motif involvement in *permutations* and sequences has now become a very active area of research [9]. However, only few results on the complexity of finding motifs in permutations and sequences are known. It appears to be a difficult problem to decide given two permutations π and σ whether σ occurs in π, and in this generality the problem is NP-complete [7]. Let $[n]$ be the set of all integers from 1 to n and let S_n be the set of all permutations on $[n]$. For $\sigma \in S_m$ and $\pi \in S_n$, the $O(n^m)$ time brute-force algorithm was improved to $O(n^{0.47m+o(m)})$ time in [2]. There are several ways in which this notion of permutation motifs may be generalized, and we focus here on *consecutive motifs* (*i.e.* the match is required to consist of contiguous elements) [9]. A sequence p is said to be a *consecutive motif* or *consecutively occurs* within a sequence t if t has a *substring* that is order-isomorphic to p. Searching for a motif p in a text t consists in identifying all occurrences of p in t. Recently, using a modification of the classical Knuth-Morris-Pratt string matching algorithm, a $O(n + m \log m)$ time algorithm has

[⋆] Helsinki Institute for Information Technology HIIT, Department of Computer Science, University of Helsinki, Finland
[⋆⋆] Institute of Discrete Mathematics and Geometry, TU Wien, Wiedner Hauptstrasse 8-10, 1040 Wien, Austria. Most of the work was done when the author was in LIAFA.
[⋆ ⋆ ⋆] LIAFA, Univ. Paris Diderot - Paris 7, 75205 Paris Cedex 13, France
[†] LIGM CNRS UMR 8049, Université Paris-Est, France. vialette@univ-mlv.fr

been proposed for checking if a given sequence t of length n contains a substring which is order-isomorphic to a given motif p of length m [10]. The time complexity reduces to $O(n+m)$ time under the assumption that the symbols of the motif can be sorted in $O(m)$ time.

Let t be a permutation of length n and p be a sequence of $m \leq n$ distinct integers in $[n]$. First we present a forward automaton which allows us to search for p in t in $O(m^2 \log\log m + n)$ time. Next, we introduce a Morris-Pratt automaton representation [11] of the forward automaton which allows us to reduce this complexity to $O(m \log\log m + n)$ at the price of an additional amortized constant term for each symbol of the text. The latter automaton occupies $O(m)$ space while the former occupies $O(m^2)$ space. We then extend the problem to search for a set of motifs and exhibit a specific Aho-Corasick like algorithm. Finally we present a sub-linear average case search algorithm running in $O\left(\frac{m \log m}{\log\log m} + \frac{n \log m}{m \log\log m}\right)$ time that we eventually prove to be optimal on average. Both lower and upper bounds assume all text permutations to be equiprobable and all integer values in a motif to be distinct.

Let us define some notations. Let $\Sigma_n = [n]$. Abusing notations, we consider in this paper permutations of S_n as strings without symbol repetition, and we denote by Σ_n^* the set of all strings without symbol repetition (including the empty string), where each symbol is an integer in $[n]$. A *prefix* (resp. *suffix*, *factor*) u of p is a string such that $p = uw, w \in \Sigma_n^*$ (resp. $p = wu, w \in \Sigma_n^*$, $p = wuz, w, z \in \Sigma_n^*$). We also denote by $|w|$ the number of integer(s) in a string $w, w \in \Sigma_n^*$. We eventually denote by p^r the reverse of p, that is, the string formed by the symbols of p read in the reverse order. We denote by $p^{\equiv}$ the set of words of Σ_n^* which are order-isomorphic to p.

The following property is useful for designing automaton transitions.

Property 1. Let $p = p_1 \dots p_m \in \Sigma_n^*$ and $w = w_1 \dots w_\ell \in \Sigma_n^*$, $\ell < m$, such that w is order-isomorphic to $p_1 \dots p_\ell$, and let $\alpha \in [n]$ s.t. $w\alpha \in \Sigma_n^*$. Testing if $w\alpha$ is order-isomorphic to $p_1 \dots p_\ell p_{\ell+1}$ can be performed in constant time using only a pair of integers.

Proof. The pair of integers (x_1, x_2) is determined as follows: $x_1 \leq \ell$ is the position of the largest number p_{x_1} in $p_1..p_\ell$ which is smaller than $p_{\ell+1}$, if any. Otherwise, we fix x_1 arbitrarily to $-\infty$. Let $x_2 \leq \ell$ be the position of the smallest integer p_{x_2} in $p_1..p_\ell$ which is larger than $p_{\ell+1}$, if any. Otherwise, we fix x_2 to $+\infty$. Now, it suffices to test if $w_{x_1} < \alpha < w_{x_2}$ to check whether $w\alpha$ is order-isomorphic to $p_1 \dots p_{\ell+1}$ $\square$

We define a function $\text{rep}(p = p_1 \dots p_m, j)$ which returns a pair of integers (x_1, x_2) that represents the pair defined in Property 1 for the prefix of length j of the motif p.

2 Tools

Before proceeding, we first describe some useful data structures we shall use as basic subroutines of our algorithms. The problem called *predecessor search* *problem* is defined as follows: given a set $S = \{x_1, x_2, \ldots, x_n\} \subset [u]$ (u is called the size of the universe), we support the following query: given an integer y return its predecessor in the set S, namely the only element x_i such that $x_i \leq y < x_{i+1}$ [1]. In addition, in the dynamic case, we also support updates: add or remove an element from the set S. The standard data structures to solve the predecessor search are the balanced binary search trees [1,5]. They use linear space and support queries and updates in worst-case $O(\log n)$ time. However, there exist better data structures that take advantage of the structure of the integers to get better query and update time. Specifically, the Van-Emde-Boas tree [13] supports queries and updates in (worst-case) time $O(\log\log u)$ using $O(u)$ space. Using randomization, the y-fast trie achieves $O(n)$ space with queries supported in time $O(\log\log u)$ and updates supported in randomized $O(\log\log u)$ time. The problem has received series of improvements which culminated with Andersson and Thorup's result [4]. They achieve $O(n)$ space with queries and updates supported in $O(\min(\log\log u, \sqrt{\frac{\log n}{\log\log n}}))$ (the update time is still randomized).

A special case occurs when space u is available and the set of keys S is known to be smaller than $\log^c u$ for some constant c. In this case all operations are supported in worst-case constant time using the atomic-heap [14].

3 Forward search automaton

The problem we consider is to search for a motif p in a permutation t without preprocessing the text itself. By analogy to the simpler case of the direct search of a word p in text t, we build an automaton that recognizes $(\Sigma_n^*) \cdot p^{\equiv}$.

We formally define our forward search automaton $\mathcal{FD}(p)$ built on $p = p_1 \ldots p_m$ as follows (see Figure 1 for an example): *(i)* $m+1$ states corresponding to each prefix (including the empty prefix) of p, state 0 is initial, state m is terminal; *(ii)* m forward transitions from state j to $j+1$ labeled by rep$(p, j+1)$; *(iii)* some backward transitions $\delta(x, [i,j])$, where x numbers a state, $0 \leq x \leq m$, $i \in \{1, \ldots, x\} \cup \{-\infty\}$, $j \in \{1, \ldots, x\} \cup \{+\infty\}$, defined the following way: $\delta(x, [i,j]) = q$ if and only if for all $p_i < \alpha < p_j$ (resp. $\alpha < p_j$ if $i = -\infty$, $p_i < \alpha$ if $j = +\infty$), the longest prefix of p that is order-isomorphic to a suffix of $p_1 \ldots p_x \alpha$ is $p_1 \ldots p_q$. We also impose some constraints on outgoing transitions:

Let x be the state corresponding to the prefix $p_1 \ldots p_x$. Let us sort all $p_i, 1 \leq i \leq x$ and consider the resulting order $p_{i_0} = -\infty < p_{i_1} < \ldots < p_{i_k} < +\infty = p_{i_{k+1}}$. We build one outgoing transition for each interval $[p_{i_j}, p_{i_{j+1}}]$, except if $p_{i_{j+1}} = p_{i_j} + 1$. Also we merge transitions that start in the same state and end if the same state whenever they are labeled by contiguous intervals.

It is obvious that the resulting automaton recognizes a given motif in a permutation by reading one by one each integer and choosing the appropriate transition. The main result on the forward automaton is the following.

Lemma 1. *Searching for a consecutive motif $p = p_1 \ldots p_m$ in a permutation $t = t_1 \ldots t_n$ using the forward automaton $\mathcal{FD}(p)$ built on p takes $O(n)$ time.*

[1] By convention, if all the elements of S are larger than y, then return $-\infty$ and if no elements is larger than y then return x_n.

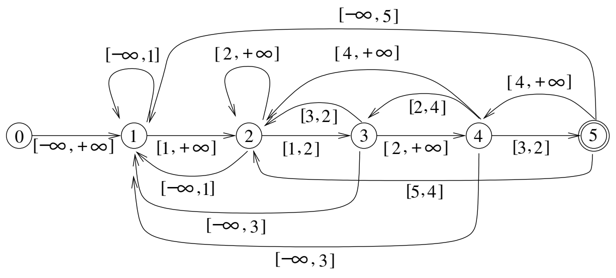

Fig. 1. Forward automaton built on $p = (4, 12, 6, 16, 10)$. State 0 is initial and state 5 is terminal.

We can build the forward automaton in $O(m^2 \log\log m)$ time. However, we defer the proof of this construction for the following reason. This $O(m^2 \log\log m)$ complexity might be too large for long motifs. Nevertheless, we show below that we can compute in a first step a type of Morris-Pratt coding of this automaton which can either (a) be directly used for the search for the motif in the text and will preserve the linear time complexity at the cost of an amortized constant term (we take more time for each text symbol), or (b) be developed to build the whole forward automaton structure.

Therefore we present and build a new automaton $\mathcal{M}P$ that is a Morris-Pratt representation of the forward automaton. The idea is to avoid building all backward transitions by only considering a special backward single transition from each state $x, x > 0$ named *failure* transition. We formally define our automaton $\mathcal{M}P(p)$ built on $p = p_1 \dots p_m$ the following way (see Figure 2 for an example):
(i) $m + 1$ states corresponding to each prefix (including the empty prefix) of p, state 0 is initial, state m is terminal;
(ii) m forward transitions from state j to $j + 1$ labeled by $\text{rep}(p, j + 1)$;
(iii) m failure (non labeled) transitions which connect a state $j > 0$ to a state $k < j$ if and only if $p_1 \dots p_k$ is the longest order-isomorphic border of $p_1 \dots p_j$:

Definition 1. *Let $p \in \Sigma_n^*$. A* border *of p is a word $w \in \Sigma_n^*, |w| < |p|$ that is order-isomorphic to a suffix of p but also order-isomorphic to a prefix of p.*

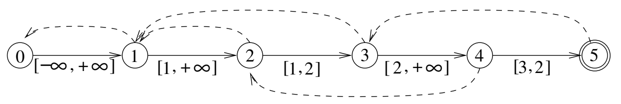

Fig. 2. $\mathcal{M}P$ automaton built on $p = (4, 12, 6, 16, 10)$. State 0 is initial and state 5 is terminal. Backward transitions are failure transitions.

Reading a text t through the $\mathcal{M}P$ representation of the forward automaton is performed the following way. Let us assume we reached state $x < m$ and we read a symbol t_i at position i of the text. Let $[k, \ell] = \text{rep}(p, x + 1)$. If $t_i \in [t_{i-x-1+k}, t_{i-x-1+\ell}]$ we follow the forward transition and the new current state is $x + 1$. Otherwise, we *fail* reading t_i from x and we retry from state $q = \text{fail}(x)$ and so-on until (a) either q is undefined, in which case we start again from state 0, (b) or a forward transition from q to $q + 1$ works, in which case the next current state is $q + 1$.

Lemma 2. *Searching for a motif p in a text $t_1 \dots t_n$ using the Morris-Pratt representation $\mathcal{M}P(p)$ of the forward automaton built on p takes $O(n)$ time.*
In order to prove Lemma 2 we need to focus on the classical notion of border that we have extended to our framework in Definition 1.

The construction of the forward automaton relies on the maximal border of each prefix that is followed by an appropriate integer in the motif. The Morris-Pratt approach is based on the following property:

Property 2. A border of a border is a border.

This property allows us to replace the direct transition of the forward algorithm by a search along the borders, from the longest to the smallest, to identify the longest one that is followed by the appropriate integer. We state now that we can build the Morris-Pratt representation of the forward automaton efficiently.

Lemma 3. *Building a Morris-Pratt representation of the forward automaton on a consecutive motif $p = p_1 \dots p_m$ can be performed in (worst-case) $O(m \log \log m)$ time.*

Lemma 2 and 3 allow us to state the main theorem of this section.

Theorem 1. *Searching for a consecutive motif $p = p_1 \dots p_m$ in a permutation $t = t_1 \dots t_n$ can be done in $O(m \log \log m + n)$ time.*

The Morris-Pratt representation of the forward automaton permits to search directly in the text at the price of larger amortized complexity (considering the constant hidden by the O notation) than that required by searching with the forward automaton directly. If the real time cost of the search phase is an issue, the forward automaton can be built from its Morris-Pratt representation.

Property 3. Building the forward automaton of a consecutive motif $p = p_1 \dots p_m$ can be performed in $O(m^2 \log \log m)$ time.

An interesting point is that the construction of the forward automaton from its Morris-Pratt representation can also be performed in a lazy way, that is, when reading the text. The missing transitions are then built *on the fly* when needed.

4 Multiple worst case linear motif searching

We can extend the previous problem defined for a single motif to a set of motifs S. We denote by d the number of motifs, by m the total length of the motifs and by r the length of the longest motif. For this problem we adapt the Aho-Corasick automaton [3] (or $\mathcal{A}C$ automaton for short). We first recall the classical construction of the $\mathcal{A}C$ automaton (for regular motifs). The $\mathcal{A}C$ automaton is a generalization of the $\mathcal{M}P$ automaton to a set of multiple motifs. We denote by P the set of prefixes of strings in S. In order to simplify the description we will assume that the set of motifs S is prefix-free. That is, we will assume that no motif is prefix of another. Extending the algorithm to the case where S is non-prefix free, should not pose any particular issue. The states of the $\mathcal{AC}$ automaton are defined in the same way as in the $\mathcal{MP}$ automaton. Each state t in the $\mathcal{AC}$ automaton corresponds uniquely to a string $q \in P$. The forward transitions are defined as follows: there exists a forward transition connecting state s corresponding to a prefix q to each state corresponding to an element $qc \in P$ (where c is a single symbol). Thus this definition of the forward transitions matches essentially the definition of the forward transitions in the $\mathcal{MP}$ automaton. The failure transitions are defined as follows: the failure transition from the state s corresponding to a prefix q goes to the state s' corresponding to the longest string q' such that $q' \in P$, q' is a suffix of q and $q' \neq q$. The matching using the $\mathcal{AC}$ automaton is done in the same way as in the $\mathcal{MP}$ automaton using the forward and failure transitions.

Our extension of the $\mathcal{AC}$ automaton. We could use exactly the same algorithm as the one used previously for our variant of the $\mathcal{MP}$ automaton with few differences. We describe our modification to the $\mathcal{AC}$ automaton to adapt it to the case of consecutive permutation matching (a similar result which has been independently discovered is described in [8]). An important observation is that we could have two or more elements of P that are both of the same length and order-isomorphic. Those two elements should have a single corresponding state in the $\mathcal{AC}$ automaton. Thus, if two or more elements of P are order-isomorphic we keep only one of them. For the forward transitions, we can associate a pair of positions (x_1, x_2) to each forward transition. Then we can check which transition is the right one from a state corresponding to a string q by checking the condition $t_{i-|q|-1+x_1} < t_i < t_{i-|q|-1+x_2}$ for every pair (x_1, x_2) and take the corresponding transition. As any state can have up to d outgoing transitions, the time taken to choose the transition would grow to $O(d)$. We reduce the time to $O(\log d)$ by organizing the forward transitions outgoing from the same state into a balanced binary search tree. That is we put at the root of the balanced binary search tree the pair of positions (x_1, x_2), where x_1 is the median of all transition pairs (sorting the pairs by p_{x_1} values), and then on the left (resp. right) subtree all transitions whose corresponding pairs (x_1 component) point to (resp. larger) smaller values in the motif.

Altenatively we can use a different approach based on a dynamic balanced binary search tree (or more sophisticated dynamic predecessor data structure). With the use of a binary search tree, we can achieve $O(\log r)$ time to decide which transition to take. More precisely, each time we read t_i we insert the pair (t_i, i) into the binary search tree. The insertion uses the number t_i as the key. Now suppose that we only pass through forward transitions. Then a transition at step i is uniquely determined by: (1) the current state s corresponding to an element $q \in P$; (2) the position of the predecessor of t_i among $t_{i-|q|} \dots t_{i-1}$.

To determine the predecessor of t_i among $t_{i-|q|} \dots t_{i-1}$, the dynamic binary search tree should contain precisely the $|q|$ pairs corresponding to $t_{i-|q|} \dots t_{i-1}$.

In order to maintain the dynamic binary search tree we must do the following actions while passing through a failure or a forward transition: (1) whenever we pass through a forward transition at a step i we insert the pair (t_i, i); (2) whenever we pass through a failure transition from a state corresponding to a prefix q_1 to a state corresponding to a prefix q_2, then we should remove from the binary tree all the pairs corresponding to the symbols $t_{i-|q_1|} \dots t_{i-|q_2|}$.

It should be noted that each removal or insertion of a pair into the binary search tree takes $O(\log r)$ time. The upper bound $O(\log r)$ comes from the fact that we never insert more than r elements in the binary search tree. Since in overall we are doing $O(n)$ insertions or removals, the amortized time should simplify to $O(n \log r)$. Finally if we replace binary search tree with a more efficient predecessor data structure, we will be able to achieve randomized time $O(n\cdot\tau)$ where $\tau = \min(\log\log n, \sqrt{\frac{\log r}{\log\log r}})$ is the time needed to do an operation on the predecessor data structure (see Section 2 for details). We use the linear space version of the predecessor data structure which guarantees only randomized performance but uses $O(r) \leq O(m)$ additional space only. We thus have the following theorem :

Theorem 2. *Searching in a text of size n for a set of d consecutive motifs whose $\mathcal{AC}$ automaton has been built and where the longest motif is of length r can be done in randomized $O(n \cdot \tau)$ time, where $\tau = \min(\log\log n, \sqrt{\frac{\log r}{\log\log r}}, \log d)$.*

Preprocessing. We now show that the preprocessing phase can be done in worst-case $O(m \log\log r)$ time. As before our starting point will be to sort all the motifs and reduce the range of symbols of each motif of length ℓ from range $[n]$ to the range $[1..\ell]$. This takes worst-case time $O(m \log\log r)$.

Recall that two or more elements of P of the same length and order-isomorphic should be associated with the same state in the $\mathcal{AC}$ automaton. In order to identify the order-isomorphic elements of P, we will carry a first step called normalization. It consists in normalizing each motif. A motif p is normalized by replacing each symbol p_j by the pair $\text{rep}(p = p_1 \dots p_{j-1}, j)$ (consisting in the positions of the predecessor and successor among symbols $p_1 \dots p_{j-1}$). This can be done for all motifs in total $O(m \log\log r)$ time. In the next step, we build a trie on the set of normalized motifs. This takes linear time. The trie naturally determines the forward transitions. More precisely any node in the trie will represent a state of the automaton and the labeled trie transitions will represent follow transitions.

Note that unlike the forward automaton (or the $\mathcal{MP}$ automaton) there could be more than one outgoing forward transition from each node. In order to encode the outgoing transitions from each node, we will make use of a hash table that stores all the transitions outgoing from that node. More precisely for each transition labeled by the pair $\text{rep}(p = p_1 \dots p_{j-1}, j)$ and directed to a state q, the hash table will associate the key p_1 associated with the value q. If we want to achieve complexity $O(\log d)$ per transition, then we organize the transition in a balanced binary search tree instead of a hash table. Now that the next transitions have been successfully built, the final step will be to build the failure transitions and this takes more effort. The construction of the failure transitions can also be done in worst-case $O(m \log \log r)$ time, but for lack of space we defer the details to the extended version [6].

We thus have the following theorem:

Theorem 3. *Building the $\mathcal{AC}$ automaton for a set of d consecutive motifs of total length m and where the longest motif is of length r can be done in worst-case $O(m \log \log r)$ time.*

5 Single sublinear average-case motif searching

Algorithm forward takes $O(n + m \log \log m)$ time in the worst case but also on average. We present now a very simple and efficient average case-algorithm which takes $O(\frac{m \log m}{\log \log m} + n \frac{\log m}{m \log \log m})$ time.

In order to search for a motif p in t, we first build a tree T of all isomorphic-order factors of length $b = \lceil \frac{3.5 \log m}{\log \log m} \rceil$ of p^r (the reverse of p). T is built by inserting each such factor one after the other in a tree and building the corresponding path if it does not already exist. The construction of this tree requires $O(\frac{m \log m}{\log \log m})$ time (details are given below). The matching phase is performed through a window of size m that is shifted along the text. For each position of this window, b symbols are read backward from the end of the window in the tree T. Two cases may occur: *(i)* either the factor is not recognized as a factor of p^r. This means that no occurrence of p might overlap this factor and we can safely shift the search window past the first symbol of this factor;
(ii) or the factor is recognized, in which case we simply check if the motif is present using a naive $O(m)$ algorithm, and we repeat this test for the next $m - b$ symbols. This might require $O(m^2)$ steps in the worst case.
In both cases we then shift the window of $m - b + 1$ symbols.

Let us analyze the average complexity of our algorithm, in the following model: all text permutations are considered to be equiprobable, all integer values in a motif are distinct.

We count the average number of symbol comparisons required to shift the search window of $m - b + 1$ symbols to the right. As there are $n/(m - b + 1)$ such segments of length $m - b + 1$ symbols in n, we will simply multiply the resulting complexity of the matching phase by $n/(m - b + 1) = O(n/m)$ to get the whole average complexity of our algorithm (assuming T is already built).

There might be $O(b!)$ distinct motifs that could appear in the text while this number is bounded by $m - b + 1$ in the motif (one by position). Thus, with a probability bounded by $\frac{m-b+1}{b!}$ we will recognize the segment of the text as a factor of p and enter case 2. In which case, moving the search window of $m - b + 1 = O(m)$ symbols to the right using the naive algorithm will require $O(m^2)$ worst case time.

In the other case which occurs with probability at least $1 - \frac{m-b+1}{b!}$, shifting the search window by $m - b + 1$ symbols to the right only requires reading b numbers.

The average complexity (in terms of number of symbol reading and comparisons) for shifting by $m - b + 1$ symbols is thus (upper) bounded by $A =$

$O((m^2)\frac{m-b+1}{b!} + b(1 - \frac{m-b+1}{b!}))$ and the whole complexity by $O((n/m)A)$. By expanding and simplifying A we get that $A = O(b + O(m^3/b!))$. Now using the famous Stirling approximation $\ln(k!) = k\ln k - k + O(\ln k)$, it is not difficult to prove that $b! = 2^{b\log b - b\log e + O(\log b)} = \Omega(m^3)$ and thus $A = O(b)$ and the whole average time complexity (in terms of number of symbol reading and comparisons) turns out to be $O(\frac{n\log m}{m\log\log m})$.

Implementation details. The tree T can actually be built in $O(\frac{m\log m}{\log\log m})$ time by using appropriate data structures. Recall that the tree T recognizes all the factors of p^r of length $\lceil\frac{3.5\log m}{\log\log m}\rceil$. To implement T, we use the same $\mathcal{AC}$ automaton presented in previous section to build the tree T, but with two differences: we only need forward transitions and the length of any motif is bounded by $\frac{\log m}{\log\log m}$. Thus the cost is upper bounded by $O(\frac{m\log m}{\log\log m}\cdot\tau)$, where τ is the time needed to do an operation on the predecessor data structure (maximum of the times needed for inserts/deletes and searches). We now turn our attention to the cost of the matching phase. From the previous section, we know that the total complexity in terms of number of symbol reading and comparisons is $O(\frac{n\log m}{m\log\log m})$. The total cost of the matching phase is dominated by the multiplication of the total number of text symbols read multiplied by the cost of a transition in the $\mathcal{AC}$ automaton which itself is dominated by $\tau = O(\min(\log\log n, \sqrt{\frac{\log r}{\log\log r}}, \log d))$, the time to do an operation on a predecessor data structure (or traversing a balanced binary search tree of size $O(d)$). The total cost of the matching phase is thus $O(\frac{n\log m}{m\log\log m}\cdot\tau)$.

Now the performance of both matching and building phases crucially depend on the used predecessor data structure. If a binary search tree is used then $\tau = O\left(\log\frac{\log m}{\log\log m}\right) = O(\log\log m)$ and the total matching time becomes $O(n\cdot\tau) = O(n\log\log m)$, and the total building time becomes $O(m\log m)$. However, we can do better if we work in the word-RAM model. Namely, we can use the atomic-heap (see Section 2) which would add additional $o(m)$ words of space and support all operations (queries, inserts and deletes) in constant time on structures of size $\log^{O(1)} m$. In our case, we have structures of maximal size $O(\frac{\log m}{\log\log m})$ and thus the operations can be supported in constant time. We thus have the following theorem:

Theorem 4. *Searching for a consecutive motif $p = p_1 \dots p_m$ in a permutation $t = t_1 \dots t_n$ can be done in average $O(\frac{m\log m}{\log\log m} + \frac{n\log m}{m\log\log m})$ time.*

6 Average optimality

We prove in this section a lower bound on the average complexity of any consecutive motif matching algorithm. The proof of this bound is inspired by that of Yao [15] which proved an average lower bound for matching a (regular) motif of length m in a text of length n. We prove in our case of interest an average lower bound of $\Omega(\frac{n\log m}{m\log\log m})$ considering all permutations over $[n]$ to be equiprobable. As this average complexity is reached by the algorithm we designed in the previous section, this bound is tight.

We begin to circumscribe our problem on small segments of length $2m-1$ of the text into which we search for. Precisely, following [15,12], we divide our text in $\lfloor n/(2m-1)\rfloor$ contiguous and non-overlapping segments s_i, $1 \le i \le \lfloor n/(2m-1)\rfloor$, such that $s_i(t) = t_{(2m-1)(i-1)+1} \ldots t_{(2m-1)i}$. When searching for a motif in t, there might be occurrences overlapping two blocks. But as we are interested in a lower bound, the following lemma allows us to focus on the inside of all segments.

Lemma 4. *A lower bound for finding a motif p inside all segments $s_i(t)$ is also a lower bound to the problem of searching for all occurrences of p in t.*

We now claim that instead of focusing on all segments $s_i(t)$, we can focus on obtaining a lower bound to search p in any single segment. Indeed these segments are non-overlapping and we are searching inside the segments.

Lemma 5. *The average time for searching for p inside all segments $s_i(t)$ is $\lfloor n/(2m-1)\rfloor$ times the average time for searching for p inside any such segment.*

Let $E(m)$ be the average complexity for searching a motif p of size m in any segment of size $2m-1$. Using the two previous lemmas, the whole average complexity is at least $\sum_{i=1}^{\lfloor n/(2m-1)\rfloor} E(m) = \lfloor n/(2m-1)\rfloor E(m) = \Omega(n/m)E(m)$.

It remains only to prove the lower bound $E(m) = \Omega(\frac{\log m}{\log\log m})$ to obtain the claimed lower bound for the whole problem.

Recall that we consider all $m!$ motif of size m to be equiprobable among the set S_m of permutations of length m. For $0 < \ell \le m$, let $\mathcal{P}_m(\ell)$ be the set of motifs of size m that can be searched using a sliding window of size m over a text of size $2m-1$ and checking only ℓ positions in this window. Then S_m is the disjoint union of $\mathcal{P}_m(\ell)$ and $S_m \setminus \mathcal{P}_m(\ell)$, that is the set of motifs that can be searched with only ℓ accesses and the others. For all motif in $\mathcal{P}_m(\ell)$, the average search complexity is counted 1 (lower bound). For any other motif in $S_m \setminus \mathcal{P}_m(\ell)$, the average search complexity is at least $\ell+1$. This leads to the following lemma:

Lemma 6. *For $0 < \ell \le m$, let $C(m,\ell) = \frac{|\mathcal{P}_m(\ell)|+(m!-|\mathcal{P}_m(\ell)|)(\ell+1)}{m!}$. Then $C(m,\ell)$ is a lower bound for the average complexity $E(m)$.*

We want now to maximize our bound in order to get a tight bound. To do so, we can choose ℓ depending on m.

Lemma 7. *There exists $\ell(m)$ s.t. $0 < \ell(m) \le m$ and $C(m,\ell(m)) = \Omega(\frac{\log m}{\log\log m})$.*

We now sketch the proof of Lemma 7. As $C(m,\ell)$ decreases when $\mathcal{P}_m(\ell)$ increases, we search an upper bound for $\mathcal{P}_m(\ell)$. We prove that $|\mathcal{P}_m(\ell)| \le m!\left(1-\frac{1}{\ell!}\right)^{\left\lceil\frac{m-1}{\ell^2}\right\rceil}$ in the same way Yao proved the counting lemma of [15]. Thus we have $C(m,\ell) \ge \ell+1-\ell\cdot\left(1-\frac{1}{\ell!}\right)^{\left\lceil\frac{m-1}{\ell^2}\right\rceil}$

We claim that $\ell = \frac{b\log m}{\log\log m}$ with $b = 1+o(1)$ satisfies $98/100 \le \left(1-\frac{1}{\ell!}\right)^{\left\lceil\frac{m-1}{\ell^2}\right\rceil} \le 99/100$ (Equation (E)). This gives $C(m,\ell(m)) \ge \ell+1-98/100\ell = \Omega(\ell) = \Omega(\frac{\log m}{\log\log m})$, stating the lemma.

The idea to prove our claim :

Let us impose $\lceil \frac{m-1}{\ell^2} \rceil \times \frac{1}{\ell!} \leq 1/10$ (*ineq.*1). This allows us to approximate Equation (E) using the classical formula $(1+x)^a = 1 + ax + \frac{a(a-1)}{2!}x^2 + \ldots + \frac{a!}{n!(a-n)!}x^n = 1 + ax + \gamma$ where $a = \lceil \frac{m-1}{\ell^2} \rceil$, $x = \frac{-1}{\ell!}$ and $\gamma = \sum_{i=2}^{n} \frac{a!}{i!(a-i)!}x^i$. It is easy to see that inequality (1) implies that γ converges and is dominated by its first term which is bounded $\frac{a(a-1)}{2!}x^2 \leq 1/200$. We thus deduce that $(1+x)^a \in [1+ax, 1+ax+1/200]$ which implies that $(1+x)^a - 1/200 \leq 1+ax \leq (1+x)^a$. From $(1+x)^a = \frac{|\mathcal{P}_m(\ell)|}{m!} \in [\frac{98}{100}, \frac{99}{100}]$, we obtain $\frac{98}{100} - \frac{1}{200} \leq 1 + ax \leq \frac{99}{100}$. By replacing a and x in $1+ax$ we get : $\frac{98}{100} - \frac{1}{200} = 195/200 \;\leq\; 1 - \lceil \frac{m-1}{\ell^2} \rceil \times \frac{1}{\ell!} \leq 99/100$. Then we prove that $\ell = \frac{b \log m}{\log \log m}$ with $b = 1 + o(1)$ satisfy these two last inequalities and inequality (1), implying that Equation (E) is satisfied.

Putting all the lemmas of this section together, we have that $\Omega(\frac{n \log m}{m \log \log m})$ is a lower bound of the whole average complexity for searching for a consecutive motif in a permutation.

References

1. M. AdelsonVelskii and E.M. Landis. *An algorithm for the organization of information.* Defense Technical Information Center, 1963.
2. S. Ahal and Y. Rabinovich. On Complexity of the Subpattern Problem. *SJDM*, 22(2):629–649, 2008.
3. A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.
4. A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13, 2007.
5. R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta informatica*, 1(4):290–306, 1972.
6. D. Belazzougui, A. Pierrot, M. Raffinot, and S. Vialette. Single and multiple consecutive permutation motif search. *CoRR*, abs/1301.4952, January 21 2013.
7. P. Bose, J.F.Buss, and A. Lubiw. Pattern matching for permutations. *Information Processing Letters*, 65(5):277–283, 1998.
8. J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama. Order preserving matching. *arXiv preprint arXiv:1302.4064*, 2013.
9. S. Kitaev. *Patterns in Permutations and Words.* EATCS. Springer, 2011.
10. M. Kubica, T. Kulczyński, J. Radoszewski, W. Rytter, and T. Waleń. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 2013.
11. J. H. Morris, Jr and Vaughan R. Pratt. A linear pattern-matching algorithm. Technical report, Univ. of California, Berkeley, 1970.
12. G. Navarro and K. Fredriksson. Average complexity of exact and approximate multiple string matching. *TCS*, 321(2-3):283–290, 2004.
13. P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.
14. D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29(3):1030–1049, December 1999.
15. A. C. Yao. The complexity of pattern matching for a random string. *SIAM Journal on Computing*, 8(3):368–387, 1979.