Algorithms **2011**, *4*, 285-306; doi:10.3390/a4040285

OPEN ACCESS

algorithms

ISSN 1999-4893

www.mdpi.com/journal/algorithms

Article

An Algorithm to Compute the Character Access Count Distribution for Pattern Matching Algorithms

Tobias Marschall [1,⋆] and Sven Rahmann [2,3,⋆]

[1] Centrum Wiskunde & Informatica (CWI), Science Park 123, 1098 XG Amsterdam, The Netherlands

[2] Genome Informatics, Faculty of Medicine, University of Duisburg-Essen, Hufelandstr. 55, 45122 Essen, Germany

[3] Bioinformatics, Computer Science XI, TU Dortmund, 44221 Dortmund, Germany

⋆ Authors to whom correspondence should be addressed; E-Mails: T.Marschall@cwi.nl (T.M.); Sven.Rahmann@tu-dortmund.de (S.R.); Tel./Fax: +31(0)20 592 4132 ext. 4199.

Received: 14 October 2011; in revised form: 26 October 2011 / Accepted: 26 October 2011 / Published: 31 October 2011

Abstract: We propose a framework for the exact probabilistic analysis of window-based pattern matching algorithms, such as Boyer–Moore, Horspool, Backward DAWG Matching, Backward Oracle Matching, and more. In particular, we develop an algorithm that efficiently computes the distribution of a pattern matching algorithm's running time cost (such as the number of text character accesses) for any given pattern in a random text model. Text models range from simple uniform models to higher-order Markov models or hidden Markov models (HMMs). Furthermore, we provide an algorithm to compute the exact distribution of *differences* in running time cost of two pattern matching algorithms. Methodologically, we use extensions of finite automata which we call *deterministic arithmetic automata* (DAAs) and *probabilistic arithmetic automata* (PAAs) [1]. Given an algorithm, a pattern, and a text model, a PAA is constructed from which the sought distributions can be derived using dynamic programming. To our knowledge, this is the first time that substring- or suffix-based pattern matching algorithms are analyzed exactly by computing the whole distribution of running time cost. Experimentally, we compare Horspool's algorithm, Backward DAWG Matching, and Backward Oracle Matching on prototypical patterns of short length and provide statistics on the size of minimal DAAs for these computations.

Keywords: pattern matching; analysis of algorithms; finite automaton; minimization; deterministic arithmetic automaton; probabilistic arithmetic automaton

Classification: MSC 68W32; 68W40

1. Introduction

The basic pattern matching problem is to find all occurrences of a *pattern* string in a (long) *text* string, with few character accesses, where a *character access* is the act of retrieving one character of the input string from memory. For many pattern matching algorithms, this is equivalent to speaking of character *comparisons*, as every accessed character is immediately compared to a character in the pattern. However, for some algorithms (e.g., the Knuth–Morris–Pratt algorithm [2]), each character access triggers a table lookup rather than a comparison. Thus, we discuss character accesses rather than character comparisons in the remainder of this article.

Let n be the text length and m be the pattern length. The well-known Knuth–Morris–Pratt algorithm [2] reads each text character exactly once from left to right and hence needs exactly n character accesses for any text of length n, after preprocessing the pattern in $\Theta(m)$ time. In contrast, the Boyer–Moore [3], Horspool [4], Sunday [5], Backward DAWG Matching (BDM, [6]) and Backward Oracle Matching (BOM, [7]) algorithms move a length-m search window across the text and first compare its *last* character to the last character of the pattern. This often allows to move the search window by more than one position (at best, by m positions if the last window character does not occur in the pattern at all), for a best case of n/m, but a worst case of mn character accesses. The worst case can often be improved to $\Theta(m + n)$, but this makes the code more complicated and seldom provides a speed-up in practice. For practical pattern matching applications, the most important algorithms are Horspool, BDM (often implemented as Backward Nondeterministic DAWG Matching, BNDM, via a non-deterministic automaton that is simulated in a bit-parallel fashion), and BOM, depending on alphabet size, text length and pattern length; see [8] for an experimental map.

A question that has apparently so far not been investigated is about the exact probability distribution of the number of required character accesses X_n^p when matching a given pattern p against a random text of finite length n (non-asymptotic case), even though related questions have been answered in the literature. For example, [9,10] analyze the expected value of X_n^p for the Horspool algorithm. In [11] it is further shown that for the Horspool algorithm, X_n^p is asymptotically normally distributed for random texts with independent and identically distributed (i.i.d.) characters, and [12] extends this result to Markovian text models. In [13], a method to compute mean and variance of these distributions is given.

In contrast to these results that are special to the Horspool algorithm, we use a general framework called *probabilistic arithmetic automata* (PAAs), introduced at CPM'08 [1], to compute the exact distribution of X_n^p for any window-based pattern matching algorithm. In [1], PAAs were introduced in order to compute the distribution of occurrence counts of patterns, a purpose for which multiple other researchers have also proposed to combine finite automata with probabilistic text models [14–17]. Especially the early approach of Nicodéme *et al.* [14] has shown how to derive generating functions and perform asymptotic analysis of occurrence distributions.

Here, we show that a similar idea can be applied to the analysis of pattern matching algorithms by constructing an automaton that encodes the behavior of such an algorithm and then combining it with a text model. The PAA framework allows doing this in a natural way, which further highlights its utility. The random text model can be quite general, from simple i.i.d. uniform models to high-order Markov models or HMMs. The approach is applied to the following pattern matching algorithms in the non-asymptotic regime (short patterns, medium-length texts): Horspool, B(N)DM, BOM. We do not treat BDM and BNDM separately as, in terms of text character accesses, they are indistinguishable (see Section 2.2).

This paper is organized as follows. In the next section, we give a brief review of the Horspool, B(N)DM and BOM algorithms. In Section 3, we define *deterministic arithmetic automata* (DAAs). In Section 4, we present a simple general DAA construction for the analysis of window-based pattern matching algorithms. We also show that the state space of the DAA can be considerably reduced by adapting DFA minimization to DAAs. In Section 5, we summarize the PAA framework with its generic algorithms, define finite-memory text models and connect DAAs to PAAs. Given a pattern p, an algorithm, and a random text model, this framework allows constructing a PAA that mimics the algorithms' behavior. By applying dynamic programming to this PAA we obtain an algorithm to compute the distribution of X_n^p for any finite text length n. Section 6 introduces *difference DAAs* by a product construction that allows to compare two algorithms on a given pattern. Results on the comparison of several algorithms for example patterns can be found in Section 7. There, we also provide statistics on automata sizes for different algorithms and pattern lengths. Section 8 contains a concluding discussion.

An extended abstract of this work has been presented at LATA'10 [18] with two alternative DAA constructions. In contrast to that version, the DAA construction in the present paper can be seen as a combination of both of those, and is much simpler. Additionally, the DAA minimization introduced in the present paper allows the analysis of much longer patterns in practice. While [18] was focused on Horspool's and Sunday's algorithms, here, we give a general construction scheme applicable to any window-based pattern matching algorithm and discuss the most relevant algorithms, namely Horspool, BOM, and B(N)DM, as examples.

Notation Throughout this paper, Σ denotes a finite alphabet, $p \in \Sigma^*$ is an arbitrary but fixed pattern, and $s \in \Sigma^*$ is the text to be searched for p. Furthermore, $m := |p|$ and $n := |s|$. Indexing generally starts at zero, that is, $s = s[0] \dots s[|s|-1]$ for all $s \in \Sigma^*$. The prefix, suffix, and substring of a string s are written $s[..i] := s[0] \dots s[i]$, $s[i..] := s[i] \dots s[|s|-1]$, and $s[i \dots j] := s[i] \dots s[j]$, respectively. By $\overleftarrow{p}$, we denote the reverse pattern $p[m-1] \dots p[0]$. For a random variable X, its distribution (law) is denoted by $\mathcal{L}(X)$. Iverson brackets are written $[\![\cdot]\!]$, *i.e.*, $[\![A]\!] = 1$ if the statement A is true and $[\![A]\!] = 0$ otherwise.

2. Algorithms

In the following, we summarize the Horspool, B(N)DM and BOM algorithms; algorithmic details can be found in [8].

We do not discuss the Knuth–Morris–Pratt algorithm because its number of text character accesses is constant: Each character of the text is looked at exactly once. Therefore, $\mathcal{L}(X_n^p)$ is the Dirac distribution on n, *i.e.*, $\mathbb{P}(X_n^p = n) = 1$.

We also do not discuss the Boyer–Moore algorithm, since it is never the best one in practice because of its complicated code to achieve optimal asymptotic running time. In contrast to our earlier paper [18], we also skip the Sunday algorithm, as it is almost always inferior to Horspool's. Instead, we focus on those algorithms that are fastest in practice according to [8].

The Horspool, B(N)DM and BOM algorithms have the following properties in common: They maintain a search window w of length $m = |p|$ that initially starts at position 0 in the text s, such that its rightmost character is at position $t = m - 1$. The right window position t grows in the course of the algorithm; we always have $w = s[(t-m+1)\dots t]$. The two properties of an algorithm that influence our analysis are the following: For a pattern $p \in \Sigma^m$, each window $w \in \Sigma^m$ determines

(1) its cost $\xi^p(w)$, e.g., the number of text character accesses required to analyze this window,
(2) its shift $\textit{shift}^p(w)$, which is the number of characters the window is advanced after it has been examined.

2.1. Horspool

First, the rightmost characters of window and pattern are compared; that means, $a := w[m-1] = s[t]$ is compared with $p[m-1]$. If they match, the remaining $m-1$ characters are compared until either the first mismatch is found or an entire match has been verified. This comparison can happen right-to-left, left-to-right, or in an arbitrary order that may depend on p. In our analysis, we focus on the right-to-left case for concreteness, but the modifications for the other cases are straightforward. Therefore, the cost of window w is

$$\xi^p(w) = \begin{cases} m & \text{if } p = w, \\ \min\{i : 1 \le i \le m,\ p[m-i] \ne w[m-i]\} & \text{otherwise.} \end{cases}$$

In any case, the rightmost window character a is used to determine how far the window can be shifted for the next iteration. The shift function ensures that no match can be missed by moving the window such that a becomes aligned to the rightmost a in p (not considering the last position). If a does not occur in p (or only at the last position), it is safe to shift by m positions. Formally, we define

$$\begin{aligned} \textit{right}^p(a) &:= \max\left[\{i \in \{0,\dots,m-2\} : p[i] = a\} \cup \{-1\}\right], \\ \texttt{shift}[a] &:= (m-1) - \textit{right}^p(a)\,, \text{ assuming } p \text{ fixed,} \\ \textit{shift}^p(w) &:= \texttt{shift}[w[m-1]]\,. \end{aligned}$$

For concreteness, we state Horspool's algorithm and how we count text character accesses as pseudocode in Algorithm 1. Note that after a shift, even when we know that a now matches its corresponding pattern character, the corresponding position is compared again and counts as a text access. Otherwise the additional bookkeeping would make the algorithm more complicated; this is not worth the effort in practice. The lookup in the `shift`-table does not count as an additional access, since we can remember `shift`$[a]$ as soon as the last window character has been read.

The main advantage of the Horspool algorithm is its simplicity. Especially, a window's shift value depends only on its last character, and its cost is easily computed from the number of consecutive matching characters at its right end. The Horspool algorithm does not require any advanced data structure and can be implemented in a few lines of code.

Algorithm 1 HORSPOOL-WITH-COST

```
Input: text $s \in \Sigma^*$, pattern $p \in \Sigma^m$
Output: pair (number $occ$ of occurrences of $p$ in $s$, number $cost$ of accesses to $s$)
 1: pre-compute table shift$[a]$ for all $a \in \Sigma$
 2: $(occ, cost) \leftarrow (0, 0)$
 3: $t \leftarrow m - 1$
 4: while $t < |s|$ do
 5:    $i \leftarrow 0$
 6:    while $i < m$ do
 7:       $cost \leftarrow cost + 1$
 8:       if $s[t-i] \neq p[(m-1)-i]$ then break
 9:       $i \leftarrow i + 1$
10:    if $i = m$ then $occ \leftarrow occ + 1$
11:    $t \leftarrow t +$ shift$[s[t]]$
12: return $(occ, cost)$
```

2.2. Backward (Nondeterministic) DAWG Matching, B(N)DM

The main idea of the BDM algorithm is to build a deterministic finite automaton (in this case, a suffix automaton, which is a directed acyclic word graph or DAWG) that recognizes all substrings of the reversed pattern, accepts all suffixes of the reversed pattern (including the empty suffix), and enters a FAIL state if a string has been read that is not a substring of the reversed pattern.

The suffix automaton processes the window right-to-left. As long as the FAIL state has not been reached, we have read a substring of the reversed pattern. If we are in an accepting state, we have even found a suffix of the reversed pattern (*i.e.*, a prefix of p). Whenever this happens before we have read m characters, the last such event marks the next potential window start that may contain a match with p, and hence determines the shift. When we are in an accepting state after reading m characters, we have found a match, but this does not influence the shift.

So, $\xi^p(w)$ is the number of characters read when entering FAIL (including the FAIL-inducing character), or m if $p = w$. Let $I^p(w) \subseteq \{0, \dots, m-1\}$ be the set defined by $i \in I^p(w)$ if and only if the suffix automaton of $\overleftarrow{p}$ is in an accepting state after reading i characters of w. Then

$$\mathit{shift}^p(w) = \min\left\{m - i \,\middle|\, i \in I^p(w)\right\}.$$

Note that $I^p(w)$ is never empty as the suffix automaton accepts the empty string and, thus, $0 \in I^p(w)$ for all windows w.

The advantage of BDM is that it makes long shifts, but its main disadvantage is the necessary construction of the suffix automaton, which is possible in $O(m)$ time via the suffix tree of the reversed pattern, but too expensive in practice to compete with other algorithms unless the search text is extremely long.

Constructing a nondeterministic finite automaton (NFA) instead of the deterministic suffix automaton is much simpler. However, processing a text character then does not take constant, but $O(m)$ time. However, the NFA can be efficiently simulated with bit-parallel operations such that processing a text character takes $O(m/W)$ time, where W is the machine word size. For many patterns in practice, this is as good as $O(1)$. The resulting algorithm is then called BNDM.

From the "text character accesses" analysis point of view, BDM and BNDM are equivalent, as they have the same shift and cost functions.

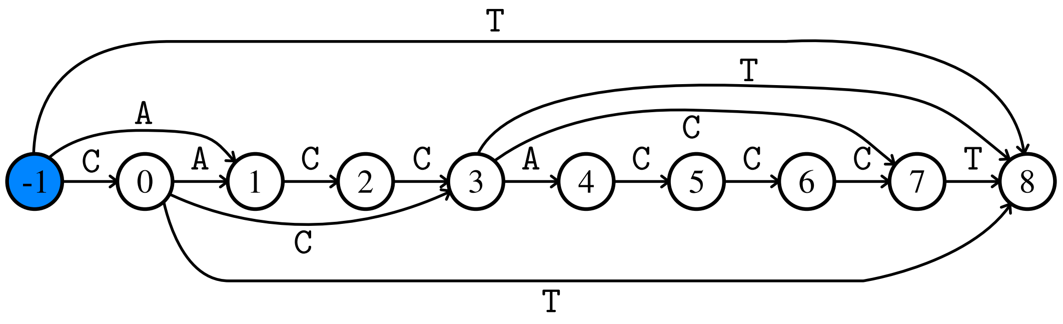
2.3. Backward Oracle Matching, BOM

BOM is similar to B(N)DM, but the suffix automaton of the reversed pattern is replaced by a simpler deterministic automaton, the factor oracle [8]. The factor oracle of a string x (which corresponds to the reversed pattern $\overleftarrow{p}$) of length m has the following properties.

(1) If y is a factor (substring) of x, then there exists a path spelling y from the start state to some state which is not the FAIL state; we say that y is *recognized.*
(2) The only string of length m recognized is x.
(3) It has the minimal number of states $(m+1)$ necessary for recognizing x (omitting the FAIL state).
(4) It has between m and $2m-1$ transitions (omitting those into the FAIL state).

It may recognize more strings than the substrings of x (although in practice not many more), but is easier to construct. It still guarantees that, once the FAIL state is reached, the sequence of read characters is not a substring of x. We refer to [8] for the construction details and further properties of the oracle; an example is shown in Figure 1.

Figure 1. Factor Oracle for $x =$ CACCACCCT, corresponding to pattern $p =$ TCCCACCAC. Omitted edges lead into the omitted FAIL state. The string ACCT is recognized (states 1,2,3,8), although it is not a substring of x.



The cost function $\xi^p(w)$ is the number of characters read when entering FAIL (including the FAIL-inducing character), or m if $p = w$. The shift function is based on the principle that the window can be safely shifted beyond the FAILed substring; so *shift*$^p(w)$ is defined as m minus the number of successfully read characters in w if $w \neq p$, and *shift*$^p(p) := 1$ (although this special case for $w = p$ can be improved by examining the pattern).

By construction, BOM never gives longer shifts than B(N)DM. The main advantage of BOM over BDM is reduced space usage and preprocessing time; the factor oracle only has $m+1$ states and can be constructed faster than a suffix automaton.

3. Deterministic Arithmetic Automata

In this section, we introduce deterministic arithmetic automata (DAAs). They extend ordinary deterministic finite automata (DFAs) by performing a computation while one moves from state to state. Even though DAAs can be shown to be formally equivalent to families of DFAs on an appropriately defined larger state space, they are a useful concept before introducing probabilistic arithmetic automata (PAAs) and allow us to construct PAAs for the analysis of pattern matching algorithms in a simpler way. By using the PAA framework, we emphasize the connection between the problems discussed in the present article and those solved before using the same formalism: Other applications in biological sequence analysis include the exact computation of clump size distributions and *p*-values of sequence motifs [19], and the determination of seed sensitivity for pairwise sequence alignment algorithms based on filtering [20].

Definition 1 (Deterministic Arithmetic Automaton, DAA) *A* deterministic arithmetic automaton *is a tuple*

$$\mathcal{D} = \big(\mathcal{Q}, q_0, \Sigma, \delta, \mathcal{V}, v_0, \mathcal{E}, (\eta_q)_{q\in\mathcal{Q}}, (\theta_q)_{q\in\mathcal{Q}}\big),$$

where $\mathcal{Q}$ is a finite set of states, $q_0 \in \mathcal{Q}$ is the start state, Σ is a finite alphabet, $\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ is a transition function, $\mathcal{V}$ is a finite or countable set of values, $v_0 \in \mathcal{V}$ is called the start value, $\mathcal{E}$ is a finite set of emissions, $\eta_q \in \mathcal{E}$ is the emission associated to state q, and $\theta_q : \mathcal{V} \times \mathcal{E} \to \mathcal{V}$ is a binary operation associated to state q.

Informally, a DAA starts with the state-value pair (q_0, v_0) and reads a sequence of symbols from Σ. Being in state q with value v, upon reading $\sigma \in \Sigma$, the DAA performs a state transition to $q' := \delta(q, \sigma)$ and updates the value to $v' := \theta_{q'}(v, \eta_{q'})$ using the operation and emission of the new state q'.

Further, we define the associated joint transition function

$$\hat{\delta} : (\mathcal{Q} \times \mathcal{V}) \times \Sigma \to (\mathcal{Q} \times \mathcal{V}), \qquad \hat{\delta}\big((q, v), \sigma\big) := \big(\delta(q, \sigma)\,,\, \theta_{\delta(q,\sigma)}(v, \eta_{\delta(q,\sigma)})\big).$$

As usual, we extend the definition of $\hat{\delta}$ inductively from Σ to Σ^* in its second argument by $\hat{\delta}\big((q, v), \varepsilon\big) := (q, v)$ for the empty string ε and $\hat{\delta}\big((q, v), x\sigma\big) := \delta\big(\hat{\delta}((q, v), x), \sigma\big)$ for all $x \in \Sigma^*$ and $\sigma \in \Sigma$.

When $\hat{\delta}\big((q_0, v_0), s\big) = (q, v)$ for some $q \in \mathcal{Q}$ and $s \in \Sigma^*$, we say that $\mathcal{D}$ computes value v for input s and define $\mathit{value}_{\mathcal{D}}(s) := v$.

For each state q, the emission η_q is fixed and could be dropped from the definition of DAAs. In fact, one could also dispense with values and operations entirely and define a DFA over state space $\mathcal{Q} \times \mathcal{V}$, performing the same operations as a DAA. However, we intentionally include values, operations, and emissions to emphasize the connection to PAAs (which are defined in Section 5).

As a simple example for a DAA, take a standard DFA $(\mathcal{Q}, q_0, \Sigma, \delta, F)$ with $F \subset \mathcal{Q}$ being a set of final (or accepting) states. To obtain a DAA that counts how many times the DFA visits an accepting state when reading $s \in \Sigma^*$, let $\mathcal{E} := \{0, 1\}$ and define $\eta_q := 1$ if $q \in F$, and $\eta_q := 0$ otherwise. Further define $\mathcal{V} = \mathbb{N}$ with $v_0 := 0$, and let the operation in each state be the usual addition: $\theta_q(v, e) := v + e$ for all q. Then $\mathit{value}_{\mathcal{D}}(s)$ is the desired count.

4. Constructing DAAs for Pattern Matching Analysis

For a given algorithm and pattern $p \in \Sigma^m$ with known shift and cost functions, $\mathit{shift}^p : \Sigma^m \to \{1, \ldots, m\}$, $w \mapsto \mathit{shift}^p(w)$ and $\xi^p : \Sigma^m \to \mathbb{N}$, $w \mapsto \xi^p(w)$, we construct a DAA that upon reading a text $s \in \Sigma^*$ computes the total cost, defined as the sum of costs over all examined windows. (Which windows are examined depends of course on the shift values of previously examined windows.) Slightly abusing notation, we write $\xi^p(s)$ for the total cost incurred on s.

While different constructions are possible (see also [18]), the construction presented here has the advantage that it is simple to describe and implement and processes only one text character at a time. This property allows the construction of a product DAA that directly compares two algorithms as detailed in Section 6.

Definition 2 (DAA encoding a pattern matching algorithm) *Given a window-based pattern matching algorithm A, a pattern $p \in \Sigma^m$, and the associated shift and cost functions, $\mathit{shift}^p : \Sigma^m \to \{1, \ldots, m\}$ and $\xi^p : \Sigma^m \to \mathbb{N}$, the* DAA encoding algorithm A *is defined by*

- $\mathcal{Q} := \Sigma^m \times \{0, \ldots, m\}$,
- $q_0 := (p, m)$,

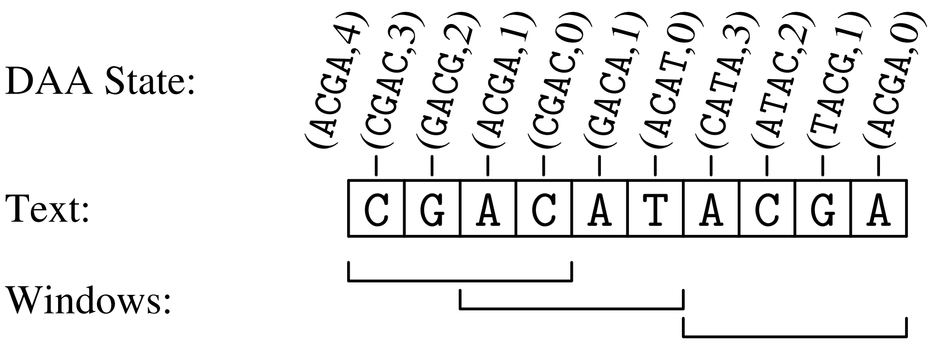
where informally, a state $q = (w, x)$ means that the last m read characters spell w and that x more characters need to be read to get to the end of the current window. For the start state $q_0 = (p, m)$, the component p is arbitrary, as we need to read m characters to reach the end of the first window.

The remaining components are defined as

- $\mathcal{V} := \mathbb{N}$,
- $v_0 := 0$,
- $\mathcal{E} := \{1, \ldots, m\}$,
- $\eta_{(w,x)} := \begin{cases} 0 & \text{if } x > 0, \\ \xi^p(w) & \text{if } x = 0, \end{cases}$
- $\theta_q : (v, e) \mapsto v + e$ *for all* $q \in \mathcal{Q}$ *(addition),*
- $\delta : \big((w, x), \sigma\big) \mapsto \begin{cases} (w'\sigma,\ x - 1) & \text{if } x > 0, \\ (w'\sigma,\ \mathit{shift}^p(w) - 1) & \text{if } x = 0, \end{cases}$
  where w' is the length-$(m-1)$ suffix of w, i.e., $w' := w[1] \ldots w[m-1]$.

Figure 2 shows an example of how a DAA for Horspool's algorithm moves from state to state. The value accumulates the cost of examined windows. Therefore, the operation is a simple addition in each state, and the emission of state (w, x) specifies the cost to add. Consequently, the emission is zero if the state does not correspond to an examined window ($x > 0$), and the emission equals the window cost $\xi^p(w)$ if $x = 0$. The transition function δ specifies how to move from one state to the next when reading the next text character $\sigma \in \Sigma$: In any case, the window content is updated by forgetting the first character and appending the read σ. If the end of the current window has not been reached ($x > 0$), the counter x is decremented. Otherwise, the window's shift value is used to compute the number of characters till the next window aligns.

Figure 2. Illustration of the DAA encoding the behavior of Horspool's algorithm when searching the text $s = CGACATACGA$ for the pattern $p = \texttt{ACGA}$. On top, one sees the state the DAA takes after reading the character below. The leftmost state is the start state. At the bottom, the windows considered by Horspool's algorithm are indicated, illustrating that the second component of the current state encodes the distance to the right end of the next window.



Theorem 1 *Let $\mathcal{D}$ be a DAA as given by Definition 2. Then, value$_\mathcal{D}(s) = \xi^p(s)$ for all $s \in \Sigma^*$.*

Proof The total cost $\xi^p(s)$ can be written as the sum of costs of all processed windows: $\xi^p(s) = \sum_{i \in I_s} \xi^p(s[i-m+1 \dots i])$, where I_s is the set of indices giving the processed windows, *i.e.*, $I_s \subset \{m-1, \dots, |s|-1\}$ such that

$$i \in I_s \quad :\Longleftrightarrow \quad i = m-1 \quad \text{or} \quad \exists j \in I_s : i = j + \textit{shift}^p(s[j-m+1 \dots j]).$$

We have to prove that the DAA computes this value for $s \in \Sigma^*$.

Let (w_i, x_i) be the DAA state active after reading $s[..i]$. Observe that the transition function δ ensures that the w_i-component of (w_i, x_i) reflects the rightmost length-m window of $s[..i]$, which can immediately be verified inductively. Thus, the emission on reading the last character $s[i]$ of $s[..i]$ with $i \geq m-1$ is, by definition of $\eta_{(w_i,x_i)}$, either $\xi^p(s[i-m+1 \dots i])$ or zero, depending on the second component of (w_i, x_i). As the operation is an addition for all states, *value*$_\mathcal{D}(s) = \sum_{i \in I'_s} \xi^p(s[i-m+1 \dots i])$ for

$$I'_s := \{i \in \{0, \dots, |s|-1\} : x_i = 0\}.$$

It remains to show that $I_s = I'_s$. To this end, note that by δ, we have $x_{i+1} = x_i - 1$ if $x_{i+1} > 0$ and $x_{i+1} = \textit{shift}^p(w_i) - 1$ if $x_{i+1} = 0$. As $q_0 = (p, m)$, it follows that $m-1 \in I'_s$. Using $w_i = s[i-m+1 \dots i]$ for $i \geq m-1$, we conclude that whenever $x_i = 0$, it follows that $x_j = 0$ for $j = i + \textit{shift}^p(s[i-m+1 \dots i])$ and that $x_{j'} > 0$ for $i < j' < j$. Hence we obtain that $i \in I'_s$ implies that $i + \textit{shift}^p(s[i-m+1 \dots i]) \in I'_s$ and $i + k \notin I'_s$ for $0 < k < \textit{shift}^p(s[i-m+1 \dots i])$, which completes the proof. □

DAA Minimization. The size of the constructed DAA's state space is $(m+1)|\Sigma|^m$ and grows exponentially with the pattern length, making the application for long patterns infeasible in practice. However, depending on the particular circumstances (*i.e.*, algorithm and pattern analyzed), the constructed DAA can often be substantially reduced by state space minimization [21]. For example, for B(N)DM, both cost and shift of an examined window depend only on the longest factor of p that is a suffix of the window. Since there are only $O(m^2)$ different factors, it is reasonable that $|\Sigma|^m$ can be replaced by $O(m^2)$, for a total state space of size $O(m^3)$. Therefore, for each algorithm, a specialized construction may exist that directly constructs the minimal state space whose size may only grow polynomially with m. For the Horspool algorithm, it is known that the state space has a size of only $O(m^2)$, as the construction of Tsai [13] can be adapted to construct a DAA according to our definition. However, we have been unable to provide a direct construction of the minimal DAA applicable to all window-based pattern matching algorithms.

Hopcroft's algorithm [21] minimizes a DFA in $O(|\mathcal{Q}| \log |\mathcal{Q}|)$ time by iteratively refining a partition of the state set. In the beginning, all states are partitioned into two distinct sets: one containing the accepting states, and the other containing the non-accepting states. This partition is iteratively refined whenever a reason for non-equivalence of two states in the same set is found. Upon termination, the states are partitioned into sets of equivalent states. Refer to [22] for an in-depth explanation of Hopcroft's algorithm.

The algorithm can straightforwardly be adapted to minimize DAAs by choosing the initial state set partition appropriately. In our case, each DAA state is associated with the same operation. The only differences in state's behavior thus stem from different emissions. Therefore, Hopcroft's algorithm can be initialized by the partition induced by the emissions and then continued as usual.

As we exemplify in Section 7, this leads to a considerable reduction of the number of states.

5. Probabilistic Arithmetic Automata

This section introduces finite-memory random text models and explains how to construct a *probabilistic arithmetic automaton* (PAA) from a (minimized) DAA and a random text model. PAAs were introduced in [1], where they are used to compute pattern occurrence count distributions. Further examples for the utility of PAAs are discussed in [19] and [20].

5.1. Random Text Models

Given an alphabet Σ, a random text is a stochastic process $(S_t)_{t\in\mathbb{N}_0}$, where each S_t takes values in Σ. A text model $\mathbb{P}$ is a probability measure assigning probabilities to (sets of) strings. It is given by (consistently) specifying the probabilities $\mathbb{P}(S_0 \dots S_{|s|-1} = s)$ for all $s \in \Sigma^*$. We only consider finite-memory models in this article which are formalized in the following definition.

Definition 3 (Finite-memory text model) *A finite-memory text model is a tuple $(\mathcal{C}, c_0, \Sigma, \varphi)$, where $\mathcal{C}$ is a finite state space (called* context space*), $c_0 \in \mathcal{C}$ a start context, Σ an alphabet, and $\varphi : \mathcal{C} \times \Sigma \times \mathcal{C} \to [0,1]$ a transition function with $\sum_{\sigma\in\Sigma, c'\in\mathcal{C}} \varphi(c, \sigma, c') = 1$ for all $c \in \mathcal{C}$. The random variable giving the text model state after t steps is denoted C_t with $C_0 :\equiv c_0$. A probability measure is now induced by stipulating*

$$\mathbb{P}(S_0 \dots S_{n-1} = s, C_1 = c_1, \dots, C_n = c_n) := \prod_{i=0}^{n-1} \varphi(c_i, s[i], c_{i+1})$$

for all $n \in \mathbb{N}_0$, $s \in \Sigma^n$, and $(c_1, \dots, c_n) \in \mathcal{C}^n$.

The idea is that the model given by $(\mathcal{C}, c_0, \Sigma, \varphi)$ generates a random text by moving from context to context and emitting a character at each transition, where $\varphi(c, \sigma, c')$ is the probability of moving from context c to context c' and thereby generating the letter σ.

Note that the probability $\mathbb{P}(S_0 \dots S_{|s|-1} = s)$ is obtained by marginalization over all context sequences that generate s. This can be efficiently done, using the decomposition of the following lemma.

Lemma 1 *Let $(\mathcal{C}, c_0, \Sigma, \varphi)$ be a finite-memory text model. Then,*

$$\mathbb{P}(S_0 \dots S_n = s\sigma, C_{n+1} = c) = \sum_{c' \in \mathcal{C}} \mathbb{P}(S_0 \dots S_{n-1} = s, C_n = c') \cdot \varphi(c', \sigma, c)$$

for all $n \in \mathbb{N}_0$, $s \in \Sigma^n$, $\sigma \in \Sigma$ and $c \in \mathcal{C}$.

Proof We have

$$\begin{aligned}
&\mathbb{P}(S_0 \dots S_n = s\sigma, C_{n+1} = c) \\
&= \sum_{c_1, \dots, c_n} \mathbb{P}(S_0 \dots S_n = s\sigma, C_1 = c_1, \dots, C_n = c_n, C_{n+1} = c) \\
&= \sum_{c_1, \dots, c_n} \prod_{i=0}^{n-1} \varphi(c_i, s[i], c_{i+1}) \cdot \varphi(c_n, \sigma, c) \\
&= \sum_{c_n \in \mathcal{C}} \left(\sum_{c_1, \dots, c_{n-1}} \prod_{i=0}^{n-1} \varphi(c_i, s[i], c_{i+1}) \right) \cdot \varphi(c_n, \sigma, c) \\
&= \sum_{c_n \in \mathcal{C}} \mathbb{P}(S_0 \dots S_{n-1} = s, C_n = c_n) \cdot \varphi(c_n, \sigma, c)
\end{aligned}$$

Renaming c_n to c' yields the claimed result. $\square$

Similar text models are used in [23], where they are called probability transducers. In the following, we refer to a finite-memory text model $(\mathcal{C}, c_0, \Sigma, \varphi)$ simply as text model, as all text models considered in this article are special cases of Definition 3.

For an i.i.d. model, we set $\mathcal{C} = \{\varepsilon\}$ and $\varphi(\varepsilon, \sigma, \varepsilon) = p_\sigma$ for each $\sigma \in \Sigma$, where p_σ is the occurrence probability of letter σ (and ε may be interpreted as an empty context). For a Markovian text model of order r, the distribution of the next character depends on the r preceding characters (fewer at the beginning); thus $\mathcal{C} := \bigcup_{i=0}^{r} \Sigma^i$. This notion of text models also covers variable order Markov chains as introduced in [24], which can be converted into equivalent models of fixed order. Text models as defined above have the same expressive power as character-emitting HMMs, that means, they allow to construct the same probability distributions.

5.2. *Basic PAA Concepts*

Probabilistic arithmetic automata (PAAs), as introduced in [1], are a generic concept useful to model probabilistic chains of operations. In this section, we sum up the definition and basic recurrences needed in this article.

Definition 4 (Probabilistic Arithmetic Automaton, PAA) *A* probabilistic arithmetic automaton *is a tuple $\mathcal{P} = \big(\mathcal{Q}, q_0, T, \mathcal{V}, v_0, \mathcal{E}, \mu = (\mu_q)_{q \in \mathcal{Q}}, \theta = (\theta_q)_{q \in \mathcal{Q}}\big)$, where $\mathcal{Q}$, q_0, $\mathcal{V}$, v_0, $\mathcal{E}$ and θ have the* *same meaning as for a DAA, each μ_q is a state-specific probability distribution on the emissions $\mathcal{E}$, and $T : \mathcal{Q} \times \mathcal{Q} \to [0,1]$ is a transition function, such that $T(q,q')$ gives the probability of a transition from state q to state q', i.e., $\big(T(q,q')\big)_{q,q' \in \mathcal{Q}}$ is a stochastic matrix.*

A PAA induces three stochastic processes: (1) the state process $(Q_t)_{t\in\mathbb{N}}$ with values in $\mathcal{Q}$, (2) the emission process $(E_t)_{t\in\mathbb{N}}$ with values in $\mathcal{E}$, and (3) the value process $(V_t)_{t\in\mathbb{N}}$ with values in $\mathcal{V}$ such that $\mathcal{V}_0 :\equiv v_0$ and $\mathcal{V}_t := \theta_{Q_t}(V_{t-1}, E_t)$.

We now restate the PAA recurrences from [1] to compute the state-value distribution after t steps. For the sake of a shorter notation, we define $f_t(q,v) := \mathbb{P}(Q_t = q, V_t = v)$. Since we are generally only interested in the value distribution, note that it can be obtained by marginalization: $\mathbb{P}(V_t = v) = \sum_{q\in\mathcal{Q}} f_t(q,v)$.

Lemma 2 (State-value recurrence, [1]) *The state-value distribution is given by $f_0(q,v) = 1$ if $q = q_0$ and $v = v_0$, and $f_0(q,v) = 0$ otherwise. For $t \geq 0$,*

$$f_{t+1}(q,v) = \sum_{q'\in\mathcal{Q}} \sum_{(v',e)\in\theta_q^{-1}(v)} f_t(q',v') \cdot T(q',q) \cdot \mu_q(e), \tag{1}$$

where $\theta_q^{-1}(v)$ denotes the inverse image set of v under θ_q.

The recurrence in Lemma 2 resembles the Forward recurrences known from HMMs.

Note that the range of V_t is finite for each t, even when $\mathcal{V}$ is infinite, as V_t is a function of the states and emissions up to time t, and state set $\mathcal{Q}$ and emission set $\mathcal{E}$ are finite. We define $\mathcal{V}_t :=$ range V_t and $\vartheta_n := \max_{0\leq t\leq n} |\mathcal{V}_t|$. Clearly $\vartheta_n \leq (|\mathcal{Q}| \cdot |\mathcal{E}|)^n$. Therefore all actual computations are on finite sets. When analyzing the number of character accesses of a pattern matching algorithm, we have $\mathcal{V}_t \subset \{0,\ldots,m(n-m+1)\}$, as at most $(n-m+1)$ search windows are processed, each causing at most m character accesses. Thus, $\vartheta_n \in O(n \cdot m)$.

5.3. Constructing a PAA from a DAA and a Text Model

We now formally state how to combine a DAA and a text model into a PAA that allows us to compute the distribution of values produced by the DAA when processing a random text.

Definition 5 (PAA induced by DAA and text model) *Let a text model $M = (\mathcal{C}, c_0, \Sigma, \varphi)$ and a DAA $\mathcal{D} = \big(\mathcal{Q}^{\mathcal{D}}, q_0^{\mathcal{D}}, \Sigma, \delta, \mathcal{V}, v_0, \mathcal{E}, (\eta_q)_{q\in\mathcal{Q}^{\mathcal{D}}}, (\theta_q^{\mathcal{D}})_{q\in\mathcal{Q}^{\mathcal{D}}}\big)$ over the same alphabet Σ be given. Then, we define the* PAA induced by $\mathcal{D}$ and M *by giving*

- *a state space $\mathcal{Q} := \mathcal{Q}^{\mathcal{D}} \times \mathcal{C}$,*
- *a start state $q_0 := (q_0^{\mathcal{D}}, c_0)$,*
- *transition probabilities*

$$T\big((q,c),(q',c')\big) := \sum_{\sigma\in\Sigma:\, \delta(q,\sigma)=q'} \varphi(c,\sigma,c'), \tag{2}$$

- *(deterministic) emission probability vectors $\mu_{(q,c)}(e) := [\![e = \eta_q]\!]$ for all $(q,c) \in \mathcal{Q}$,*
- *operations $\theta_{(q,c)}(v,e) := \theta_q^{\mathcal{D}}(v,e)$ for all $(q,c) \in \mathcal{Q}$.*

Note that states having zero probability of being reached from q_0 may be omitted from $\mathcal{Q}$ and T without changing the PAA's state, emission or value process. The next lemma states that the PAA given by Definition 3 indeed reflects the probabilistic behavior of the input DAA acting on a random text generated by the text model. Furthermore, it gives the runtime required to compute the distribution of DAA values via dynamic programming.

Lemma 3 (Properties of PAA induced by DAA and text model) *Let a text model $M = (\mathcal{C}, c_0, \Sigma, \varphi)$ and a DAA $\mathcal{D} = \big(\mathcal{Q}^\mathcal{D}, q_0^\mathcal{D}, \Sigma, \delta, \mathcal{V}, v_0, \mathcal{E}, (\eta_q)_{q\in\mathcal{Q}^\mathcal{D}}, (\theta_q^\mathcal{D})_{q\in\mathcal{Q}^\mathcal{D}}\big)$ be given and let $\mathcal{P} = \big(\mathcal{Q}, q_0, T, \mathcal{V}, v_0, \mathcal{E}, \mu = (\mu_q)_{q\in\mathcal{Q}}, \theta = (\theta_q)_{q\in\mathcal{Q}}\big)$ be the PAA given by Definition 5. Then,*

(1) $\mathcal{L}(V_t) = \mathcal{L}\big(value_\mathcal{D}(S_0 \dots S_{t-1})\big)$ for all $t \in \mathbb{N}_0$, where S is a random text according to the text model M,

(2) the value distribution $\mathcal{L}(V_n)$ can be computed with $O(n \cdot |\mathcal{Q}^\mathcal{D}| \cdot |\mathcal{C}|^2 \cdot |\Sigma| \cdot \vartheta_n)$ operations using $O(|\mathcal{Q}^\mathcal{D}| \cdot |\mathcal{C}| \cdot \vartheta_n)$ space, and

(3) if for all $c \in \mathcal{C}$ and $\sigma \in \Sigma$, there exists at most one $c' \in \mathcal{C}$ such that $\varphi(c, \sigma, c') > 0$, then the runtime is bounded by $O(n \cdot |\mathcal{Q}^\mathcal{D}| \cdot |\mathcal{C}| \cdot |\Sigma| \cdot \vartheta_n)$.

Proof As in Section 5.2, we define $f_t(q, v) := \mathbb{P}(Q_t = q, V_t = v)$. To prove 1, we show that

$$f_t\big((q^\mathcal{D}, c), v\big) = \sum_{s\in\Sigma^t} \big[\!\big[\hat{\delta}\big((q_0^\mathcal{D}, v_0), s\big) = (q^\mathcal{D}, v)\big]\!\big] \cdot \mathbb{P}(S_0 \dots S_{t-1} = s, C_t = c) \tag{3}$$

for all $q^\mathcal{D} \in \mathcal{Q}^\mathcal{D}$, $c \in \mathcal{C}$, $v \in \mathcal{V}$, and $t \in \mathbb{N}_0$. For $t = 0$, Equation (3) is satisfied by definitions of PAAs, DAAs and text models. For $t > 0$ we prove it inductively. Assume Equation (3) to be correct for all t' with $0 \le t' < t$. Then

$$f_t\big(\underbrace{(q^\mathcal{D}, c)}_{=:q}, v\big) \tag{4}$$

$$= \sum_{q'\in\mathcal{Q}} \sum_{(v',e)\in\theta_q^{-1}(v)} f_{t-1}(q', v') \cdot T(q', q) \cdot \mu_q(e) \tag{5}$$

$$= \sum_{q'\in\mathcal{Q}} \sum_{(v',e)\in\mathcal{V}\times\mathcal{E}} \big[\!\big[\theta_{q^\mathcal{D}}^\mathcal{D}(v', e) = v\big]\!\big] \cdot f_{t-1}(q', v') \cdot T(q', q) \cdot \big[\!\big[\eta_{q^\mathcal{D}} = e\big]\!\big] \tag{6}$$

$$\begin{aligned}= \sum_{q'^\mathcal{D}\in\mathcal{Q}^\mathcal{D}} \sum_{c'\in\mathcal{C}} \sum_{(v',e)\in\mathcal{V}\times\mathcal{E}} & \big[\!\big[\theta_{q^\mathcal{D}}^\mathcal{D}(v', e) = v\big]\!\big] \cdot \big[\!\big[\eta_{q^\mathcal{D}} = e\big]\!\big] \cdot f_{t-1}(q', v') \\ & \cdot \sum_{\sigma\in\Sigma} \big[\!\big[\delta(q'^\mathcal{D}, \sigma) = q^\mathcal{D}\big]\!\big] \cdot \varphi(c', \sigma, c)\end{aligned} \tag{7}$$

$$\begin{aligned}= \sum_{s\in\Sigma^{t-1}} \sum_{\sigma\in\Sigma} \sum_{q'^\mathcal{D}\in\mathcal{Q}^\mathcal{D}} \sum_{c'\in\mathcal{C}} \sum_{(v',e)\in\mathcal{V}\times\mathcal{E}} & \big[\!\big[\theta_{q^\mathcal{D}}^\mathcal{D}(v', e) = v\big]\!\big] \cdot \big[\!\big[\eta_{q^\mathcal{D}} = e\big]\!\big] \\ & \cdot \big[\!\big[\delta(q'^\mathcal{D}, \sigma) = q^\mathcal{D}\big]\!\big] \cdot \big[\!\big[\hat{\delta}\big((q_0^\mathcal{D}, v_0), s\big) = (q'^\mathcal{D}, v')\big]\!\big] \\ & \cdot \mathbb{P}(S_0 \dots S_{t-2} = s, C^{t-1} = c') \cdot \varphi(c', \sigma, c)\end{aligned} \tag{8}$$

$$\begin{aligned}= \sum_{s\sigma\in\Sigma^t} \sum_{q'^\mathcal{D}\in\mathcal{Q}^\mathcal{D}} \sum_{(v',e)\in\mathcal{V}\times\mathcal{E}} & \big[\!\big[\theta_{q^\mathcal{D}}^\mathcal{D}(v', e) = v\big]\!\big] \cdot \big[\!\big[\eta_{q^\mathcal{D}} = e\big]\!\big] \cdot \big[\!\big[\hat{\delta}\big((q_0^\mathcal{D}, v_0), s\big) = (q'^\mathcal{D}, v')\big]\!\big] \\ & \cdot \big[\!\big[\delta(q'^\mathcal{D}, \sigma) = q^\mathcal{D}\big]\!\big] \cdot \mathbb{P}(S_0 \dots S_{t-1} = s\sigma, C_t = c)\end{aligned} \tag{9}$$

$$= \sum_{s\sigma \in \Sigma^t} [\![\hat{\delta}\big((q_0^{\mathcal{D}}, v_0), s\sigma\big) = (q^{\mathcal{D}}, v)]\!] \cdot \mathbb{P}(S_0 \ldots S_{t-1} = s\sigma, C_t = c). \tag{10}$$

In the above derivation, step (4)→(5) follows from (1). Step (5)→(6) follows from the definitions of θ_q and μ_q. Step (6)→(7) uses the definitions of T and $\mathcal{Q}$ in Lemma 3. Step (7)→(8) uses the induction assumption. Step (8)→(9) uses Lemma 1. The final step (9)→(10) follows by combining the four Iverson brackets summed over $q'^{\mathcal{D}}$ and (v', e) into a single Iverson bracket.

To compute the table f_n containing $f_n(q, v)$ for all $q \in \mathcal{Q}$ and $v \in \mathcal{V}$, we start with f_0 and perform n update steps. The runtime bounds given in 2. and 3. can be verified by considering a "push" algorithm: When computing f_{t+1}, we initialize the table with zeros and iterate over all $q \in \mathcal{Q}$, $v \in \mathcal{V}$ and $q' \in \{q'' \in \mathcal{Q} : T(q, q'') > 0\}$; for each combination of q, v, and q' with $T(q, q') > 0$, we add $f_t(q, v) \cdot T(q, q')$ to $f_{t+1}\big(q', \theta_{q'}(v, \eta_{q'})\big)$. □

As a direct consequence of the above lemma and of the DAA construction from Section 4, we arrive at our main theorem.

Theorem 2 *Let a finite-memory text model $(\mathcal{C}, c_0, \Sigma, \varphi)$, a window-based pattern matching algorithm A, a pattern p with $|p| = m$, and the functions shift^{A,p} and $\xi^{A,p}$ be given. Then, the cost distribution $\mathcal{L}(X_n^{A,p})$ can be computed using $O(n^2 \cdot m \cdot |\mathcal{Q}^{\mathcal{D}}| \cdot |\mathcal{C}|^2 \cdot |\Sigma|)$ time and $O(|\mathcal{Q}^{\mathcal{D}}| \cdot |\mathcal{C}| \cdot n \cdot m)$ space. If for all $c \in \mathcal{C}$ and $\sigma \in \Sigma$, there exists at most one $c' \in \mathcal{C}$ such that $\varphi(c, \sigma, c') > 0$, a factor of $|\mathcal{C}|$ can be dropped from the runtime bounds.*

Using optimal algorithm-dependent DAA constructions schemes (e.g., the $O(m^2)$ construction for the Horspool algorithm by Tsai [13]) allows to replace $|\mathcal{Q}^{\mathcal{D}}|$ by a polynomial in m, instead of $O(m|\Sigma|^m)$.

6. Comparing Algorithms with Difference DAAs

Computing the cost distribution for two algorithms allows us to compare their performance characteristics. One natural question, however, cannot be answered by comparing these two (one-dimensional) distributions: What is the probability that algorithm A needs more text accesses than algorithm B to scan the same random text? The answer will depend on the correlation of algorithm performances: Do the same instances lead to long runtimes for both algorithms or are there instances that are easy for one algorithm but difficult for the other? This section answers these questions by constructing a PAA to compute the distribution of *cost differences* of two algorithms. That means, we calculate the probability that algorithm A needs v text accesses *more* than algorithm B for all $v \in \mathbb{Z}$.

We start by giving a general construction of a DAA that computes the difference of the sum of emission of two given DAAs.

Definition 6 (Difference DAA)
Let a finite alphabet Σ and two DAAs $\mathcal{D}^1 = \big(\mathcal{Q}^1, q_0^1, \Sigma, \delta^1, \mathcal{V}^1, v_0^1, \mathcal{E}^1, (\eta_q^1)_{q\in\mathcal{Q}^1}, (\theta_q^1)_{q\in\mathcal{Q}^1}\big)$ and $\mathcal{D}^2 = \big(\mathcal{Q}^2, q_0^2, \Sigma, \delta^2, \mathcal{V}^2, v_0^2, \mathcal{E}^2, (\eta_q^2)_{q\in\mathcal{Q}^2}, (\theta_q^2)_{q\in\mathcal{Q}^2}\big)$ be given with $\mathcal{V}^1 = \mathcal{V}^2 = \mathbb{N}$, $v_0^1 = v_0^2 = 0$, $\mathcal{E}^1, \mathcal{E}^2 \subset \mathbb{N}$, and all operations are additions of previous value and current emission. The difference DAA of $\mathcal{D}^1$ and *$\mathcal{D}^2$ is defined as*

$$\mathit{DiffDAA}(\mathcal{D}^1, \mathcal{D}^2) := \big(\mathcal{Q}, q_0, \Sigma, \delta, \mathcal{V}, v_0, \mathcal{E}, (\eta_q)_{q\in\mathcal{Q}}, (\theta_q)_{q\in\mathcal{Q}}\big)\,,$$

where

- $\mathcal{Q} := \mathcal{Q}^1 \times \mathcal{Q}^2$ *and* $q_0 := (q_0^1, q_0^2)$,
- $\mathcal{V} := \mathbb{Z}$ *and* $v_0 := 0$,
- $\mathcal{E} := \mathcal{E}^1 \times \mathcal{E}^2$ *and* $\eta_{(q^1,q^2)} := \left(\eta^1_{q^1}, \eta^2_{q^2}\right)$,
- $\delta : \left((q^1, q^2), \sigma\right) \mapsto \left(\delta^1(q^1, \sigma), \delta^2(q^2, \sigma)\right)$,
- $\theta_q : \left(v, (e^1, e^2)\right) \mapsto v + e^1 - e^2$.

Lemma 4 *Let* $\mathcal{D}^1$ *and* $\mathcal{D}^2$ *be DAAs meeting the criteria given in Definition 6 and* $\mathcal{D} :=$ *DiffDAA*$(\mathcal{D}^1, \mathcal{D}^2)$*. Then, value*$_{\mathcal{D}}(s) =$ *value*$_{\mathcal{D}^1}(s) -$ *value*$_{\mathcal{D}^2}(s)$ *for all* $s \in \Sigma^*$.

Proof Follows directly from Definition 6. □

Lemma 4 can now be applied to the DAAs constructed for the analysis of two algorithms as described in Section 4. Since the above construction builds the product of both state spaces, it is advisable to minimize both DAAs before generating the product. Furthermore, in an implementation, only reachable states of the product automaton need to be constructed. Before being used to build a PAA (by applying Lemma 3), the product DAA should again be minimized.

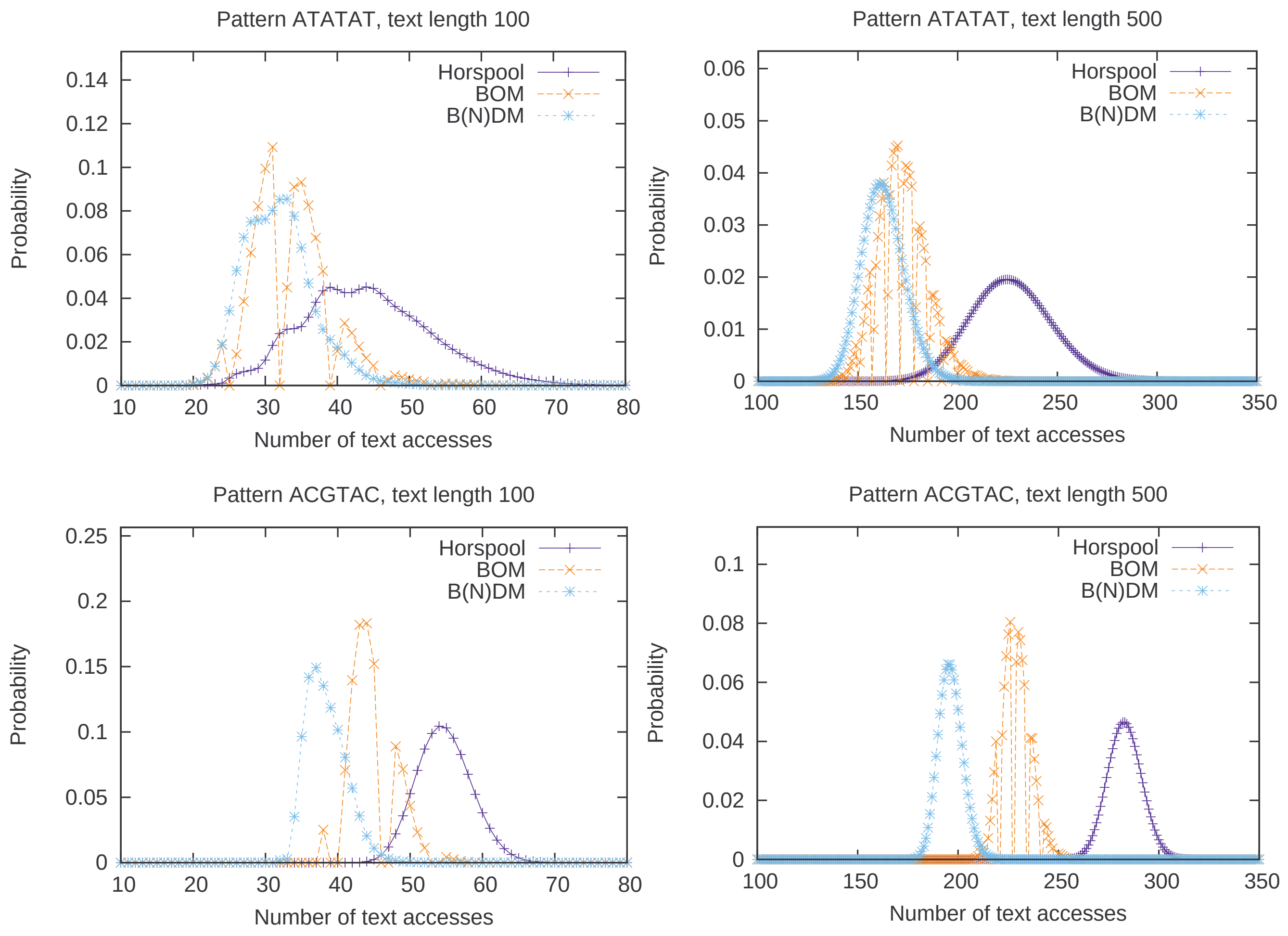
As discussed in Section 5.2, at most $m(n - m + 1)$ character accesses can result from scanning a text of length n for a pattern of length m. Thus, the difference of costs for two algorithms lies in the range $\{-m(n - m + 1), \ldots, m(n - m + 1)\}$ and, hence, $\vartheta_n \in O(n \cdot m)$.

7. Case Studies

In Section 2, we considered three practically relevant algorithms, namely Horspool's algorithm, backward oracle matching (BOM), and backward (non)-deterministic DAWG matching (B(N)DM). Now, we compare the distributions of running time costs of these algorithms for several patterns over the DNA alphabet {`A`, `C`, `G`, `T`}. Figure 3 shows these distributions for the patterns `ATATAT` and `ACGTAC` for text lengths 100 and 500 under a second order Markovian text model estimated from the human genome. For text length 500, the distributions for Horspool and B(N)DM resemble the shape of normal distributions. In fact, for Horspool's algorithm it has been proven that the distribution is asymptotically normal [12]. For smaller text lengths (e.g., 100, as shown in left column of Figure 3), the distributions are less smooth than for longer texts.

It is remarkable that for BOM we find zero probabilities with a fixed period. The period equals $m + 1$ which is 7 in the shown examples. This behavior is caused by the factor-based nature of BOM; when a suffix of the search window has been recognized as not being a factor (substring) of the pattern, the window is just moved far enough to exclude this substring, creating the relation *shift*$^p(w) = m - \xi^p(w) + 1$ between cost and shift of a window w. As the following lemma shows, this property is a sufficient condition for the observed zero probabilities.

Figure 3. Exact distributions of character access counts for patterns `ATATAT` (top) and `ACGTAC` (bottom) for text length 100 (left) and text length 500 (right). A second order Markovian text model estimated from the human genome is used.



Lemma 5 *Let a window-based pattern matching algorithm A, a pattern p with $|p| = m$, and the functions shift^{A,p} and $\xi^{A,p}$ be given such that shift$^{A,p}(w) = m - \xi^{A,p}(w) + 1$ for all $w \in \Sigma^m$. Then,*

$$\xi^{A,p}(s) + n + 1 \not\equiv 0 \mod (m+1)$$

for all $n \in \mathbb{N}$ and all $s \in \Sigma^n$.

Proof Let $w_1, \ldots, w_k$ be the sequence of windows examined by algorithm A when processing the text $s \in \Sigma^n$. In the beginning, the rightmost position of the current window is at position $m-1$ and is moved by *shift*$^{A,p}(w_i)$ after processing window w_i. After processing all windows it is beyond the end of the text. Formally,

$$n \leq m - 1 + \sum_{i=1}^{k} \mathit{shift}^{A,p}(w_i) < n + m$$

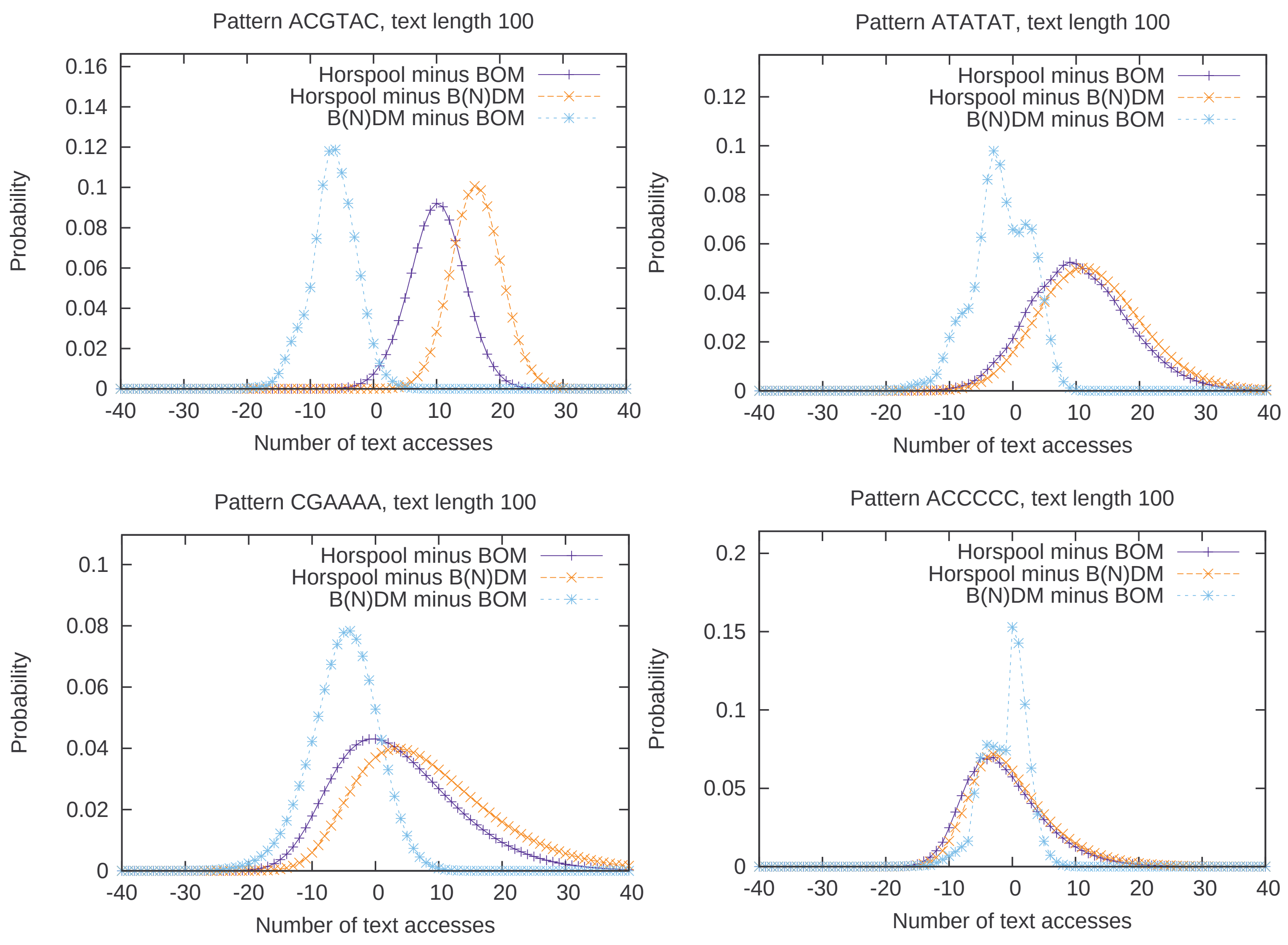
By using the assumption that *shift*$^{A,p}(w) = m - \xi^{A,p}(w) + 1$, we obtain

$$\begin{aligned}
& n \le m - 1 + \sum_{i=1}^{k} \big(m - \xi^{A,p}(w_i) + 1\big) < n + m \\
\iff \quad & n \le m - 1 + k(m+1) - \xi^{A,p}(s) < n + m \\
\iff \quad & k(m+1) < \xi^{A,p}(s) + n + 1 \le k(m+1) + m
\end{aligned}$$

which means that $\xi^{A,p}(s) + n + 1$ lies strictly between k-times and $(k+1)$-times $m+1$ and thus cannot be a multiple of $m+1$. $\square$

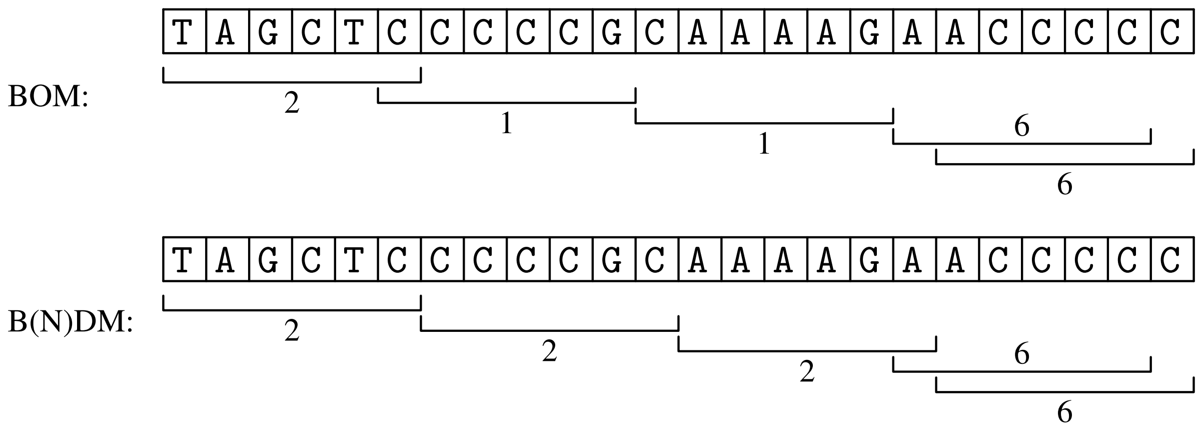
The probability that one pattern matching algorithm is faster than another depends on the pattern. Using the technique introduced in Section 6, we can quantify the strength of this effect. Figure 4 shows distributions of cost *differences* for different patterns and algorithms. That means, the probability that the first algorithm is faster is represented by the area under the curve left of zero. For the pattern ACCCCC and a random text of length 100, for example, there is a 53.9% probability that Horspool's algorithm needs fewer character accesses than B(N)DM (for the same second order Markovian model as used before), while for ACGTAC, the probability is only 0.0016%.

Figure 4. Exact distributions of differences in character access counts for different patterns using a second order Markovian text model estimated from the human genome and random texts of lengths 100.



Worth noting and perhaps surprising is the fact that there is a non-zero probability of BOM being faster than B(N)DM although, $\mathit{shift}^{\text{B(N)DM},p}(w) \geq \mathit{shift}^{\text{BOM},p}(w)$ for all window contents w. The explanation, of course, is that a shorter (say, first) shift for BOM leads to a different window content than for B(N)DM for the second window, which may have a larger shift value. This effect depends on the pattern: For the pattern `ACCCCC`, there is a 52.4% probability that BOM is at least as fast as B(N)DM (in terms of character accesses), while it is 4.9% for `ACGTAC`, again on texts of length 100. An example text where BOM is faster than B(N)DM while searching for the pattern `ACCCCC` is shown in Figure 5. Both algorithms read two characters of the first window but prescribe different shifts. The first window ends on `TC`. BOM recognizes that `TC` is not a substring of the pattern an shifts the window by five positions, just far enough to exclude this substring. In contrast, B(N)DM determines that neither `C` nor `TC` are prefixes of the pattern and shifts the window by six positions. It turns out, however, that a shorter shift of five positions was beneficial in this case as BOM can process the second window with one character access, while B(N)DM uses two character accesses.

Figure 5. Example of a string for which BOM executes less character accesses than B(N)DM when searching for the pattern $p = \texttt{ACCCCC}$. The searched windows are indicated below the text; the nearby number gives the number of character accesses executed when processing this window.

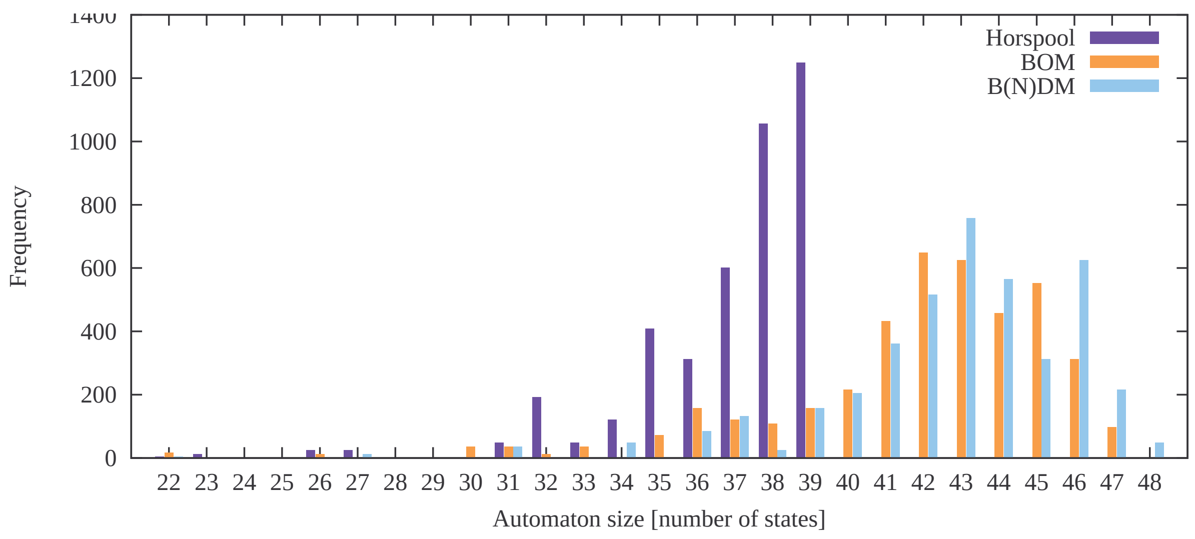

T A G C T C C C C C G C A A A A G A A C C C C C
BOM:
2
1
1
6
6
T A G C T C C C C C G C A A A A G A A C C C C C
B(N)DM:
2
2
2
6
6

To assess the effect of DAA minimization before constructing PAAs, we constructed minimized DAAs for all 21840 patterns of lengths 2 to 7 over $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. The minimum, average, and maximum state counts are shown in Table 1. For length 6, Figure 6 contains a detailed histogram. These statistics show that construction and minimization as given in this article lead to smaller automata (and thus better runtimes) than the constructions given in the conference version of this article [18]. It may be conjectured that the worst-case size of the minimal state space grows only polynomially with m for all of these algorithms, as has been previously proven for the Horspool algorithm [13].

Table 1. Comparison of DAA sizes for all patterns of length m over $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$.

m	**States unminimized**	**States minimized (min./avg./max.)**		
	$\lvert\Sigma\rvert^m \cdot (m+1)$	**Horspool**	**BOM**	**B(N)DM**
2	48	4 / 4.8 / 5	4 / 4.0 / 4	4 / 4.8 / 5
3	256	7 / 8.3 / 9	7 / 8.3 / 9	7 / 9.6 / 10
4	1280	11 / 14.3 / 15	11 / 15.6 / 18	11 / 17.0 / 19
5	6144	16 / 23.6 / 25	16 / 26.5 / 30	16 / 27.9 / 31
6	28672	22 / 37.0 / 39	22 / 41.8 / 47	22 / 42.8 / 48
7	131072	29 / 55.2 / 58	29 / 62.4 / 70	29 / 62.6 / 70

Figure 6. Histogram on number of states of minimal DAAs over all patterns of length 6 over $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$.



The algorithms were implemented in JAVA and are available as part of the MoSDi software package available at http://mosdi.googlecode.com. They were run on an Intel Core 2 Dual CPU at 2.1 GHz. Computing the distributions shown in Figure 3 took 0.5 to 1.3 seconds for each distribution. Distributions of differences as in Figure 4 were computed in 56 to 97 seconds.

8. Discussion

Using PAAs, we have shown how the exact distribution of the number of character accesses for window-based pattern matching algorithms can be computed algorithmically. The framework admits general finite-memory text models, including i.i.d. models, Markov models of arbitrary order, and character-emitting hidden Markov models. The given construction results in an asymptotic runtime of $O(n^2 \cdot m \cdot |\mathcal{Q}^{\mathcal{D}}| \cdot |\mathcal{C}|^2 \cdot |\Sigma|)$. The number of DAA states $|\mathcal{Q}^{\mathcal{D}}|$ can be as large as $O(m \cdot \Sigma^m)$, but we conjecture that for each reasonable algorithm, the necessary minimal state set $\mathcal{Q}^{\mathcal{D}}_{\min}$ grows only polynomially with m. In particular, we conjecture $O(m^3)$ sizes for B(N)DM and BOM; this is consistent with the numbers in Table 1. For Horspool, a specialized $O(m^2)$ construction is known [13]. Otherwise, in practice, the DAA size can be reduced by DAA minimization, but it remains open if there exists an algorithm to construct the minimal automaton directly in general, *i.e.*, using only $O(|\mathcal{Q}^{\mathcal{D}}_{\min}|)$ time. A proof that this is the case for a broad class of pattern matching algorithms would be an important insight into the nature of these algorithms and therefore certainly warrants further research.

The behavior of BOM deserves further attention: first, periodic zero probabilities are found in its distribution of text character accesses; and second, it may (unexpectedly) need fewer text accesses than B(N)DM on some patterns, although BOM's shift values are never better than B(N)DM's.

We focused on algorithms for single patterns, but the presented techniques also apply to algorithms to search for multiple patterns like the Wu–Manber algorithm [25] or "set backward oracle matching" and "multiple BNDM", as described in [8]. A comparison of the resulting distributions could yield new insights into these algorithms as well.

Other metrics than text character accesses might be of interest and could be easily substituted; for example, just counting the number of windows by defining $\xi^p(w) = 1$ for all $w \in \Sigma^m$.

The given constructions allow us to analyze an algorithm's performance for each pattern individually. While this is desirable for detailed analysis, the cost distribution resulting from randomly choosing text *and* pattern would also be of interest.

The results of this paper were obtained while Tobias Marschall was a PhD student with Sven Rahmann and TU Dortmund. The thesis is available at http://hdl.handle.net/2003/27760.

Acknowledgement

Most work was carried out while both authors were affiliated with TU Dortmund.

References

1. Marschall, T.; Rahmann, S. Probabilistic Arithmetic Automata and Their Application to Pattern Matching Statistics. In *Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching, CPM '08*, Pisa, Italy, 18–20 June 2008; Ferragina, P., Landau, G.M., Eds.; Springer: Berlin, Germany, 2008; Volume 5029, pp. 95–106.
2. Knuth, D.E.; Morris, J.; Pratt, V.R. Fast pattern matching in strings. *SIAM J. Comput.* **1977**, *6*, 323–350.
3. Boyer, R.S.; Moore, J.S. A fast string searching algorithm. *Commun. ACM* **1977**, *20*, 762–772.
4. Horspool, R.N. Practical fast searching in strings. *Softw.-Pract. Exp.* **1980**, *10*, 501–506.
5. Sunday, D.M. A very fast substring search algorithm. *Commun. ACM* **1990**, *33*, 132–142.
6. Crochemore, M.; Czumaj, A.; Gasieniec, L.; Jarominek, S.; Lecroq, T.; Plandowski, W.; Rytter, W. Speeding up two string-matching algorithms. *Algorithmica* **1994**, *12*, 247–267.
7. Allauzen, C.; Crochemore, M.; Raffinot, M. Efficient Experimental String Matching by Weak Factor Recognition. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, CPM '01*, Jerusalem, Israel, 1–4 July 2001; Goos, G., Hartmanis, J., van Leeuwen, J., Eds.; Volume 2089, pp. 51–72.

8. Navarro, G.; Raffinot, M. *Flexible Pattern Matching in Strings*; Cambridge University Press: Cambridge, UK, 2002.
9. Baeza-Yates, R.A.; Gonnet, G.H.; Régnier, M. Analysis of Boyer-Moore-Type String Searching Algorithms. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '90*, San Francisco, CA, USA, 22–24 January 1990; pp. 328–343.
10. Baeza-Yates, R.A.; Régnier, M. Average running time of the boyer-moore-horspool algorithm. *Theor. Comput. Sci.* **1992**, *92*, 19–31.
11. Mahmoud, H.M.; Smythe, R.T.; Régnier, M. Analysis of Boyer-Moore-Horspool string-matching heuristic. *Random Struct. Algorithms* **1997**, *10*, 169–186.
12. Smythe, R.T. The Boyer-Moore-Horspool heuristic with Markovian input. *Random Struct. Algorithms* **2001**, *18*, 153–163.
13. Tsai, T. Average case analysis of the Boyer-Moore algorithm. *Random Struct. Algorithms* **2006**, *28*, 481–498.
14. Nicodème, P.; Salvy, B.; Flajolet, P. Motif statistics. *Theor. Comput. Sci.* **2002**, *287*, 593–617.
15. Nicodème, P. Regexpcount, a symbolic package for counting problems on regular expressions and words. *Fundam. Inform.* **2002**, *56*, 71–88.
16. Nuel, G. Pattern Markov chains: Optimal Markov chain embedding through deterministic finite automata. *J. Appl. Probab.* **2008**, *45*, 226–243.
17. Lladser, M.; Betterton, M.D.; Knight, R. Multiple pattern matching: A Markov chain approach. *J. Math. Biol.* **2008**, *56*, 51–92.
18. Marschall, T.; Rahmann, S. Exact Analysis of Horspool's and Sunday's Pattern Matching Algorithms with Probabilistic Arithmetic Automata. In *Proceedings of the 4th International Conference on Language and Automata Theory and Applications, LATA '10*, Trier, Germany, 24–28 May 2010; Dediu, A.H., Fernau, H., Martín-Vide, C., Eds.; Volume 6031, pp. 439–450.
19. Marschall, T.; Rahmann, S. Efficient exact motif discovery. *Bioinformatics* **2009**, *25*, i356–i364.
20. Herms, I.; Rahmann, S. Computing Alignment Seed Sensitivity with Probabilistic Arithmetic Automata. In *Proceedings of the 8th International Workshop Algorithms in Bioinformatics, WABI '08*, Karlsruhe, Germany, 15–19 September 2008; Crandall, K., Lagergren, J., Eds.; Springer: Berlin, Germany, 2008; Volume 5251, pp. 318–329.
21. Hopcroft, J. An $n \log n$ Algorithm for Minimizing the States in a Finite Automaton. In *The Theory of Machines and Computations*; Kohavi, Z., Paz, A., Eds.; Academic Press: New York, NY, USA, 1971; pp. 189–196.
22. Knuutila, T. Re-describing an algorithm by Hopcroft. *Theor. Comput. Sci.* **2001**, *250*, 333–363.
23. Kucherov, G.; Noé, L.; Roytberg, M. A unifying framework for seed sensitivity and its application to subset seeds. *J. Bioinform. Comput. Biol.* **2006**, *4*, 553–569.
24. Schulz, M.; Weese, D.; Rausch, T.; Döring, A.; Reinert, K.; Vingron, M. Fast and Adaptive Variable Order Markov Chain Construction. In *Proceedings of the 8th International Workshop Algorithms in Bioinformatics, WABI '08*, Karlsruhe, Germany, 15–19 September 2008; Crandall, K.A., Lagergren, J., Eds.; Springer: Berlin, Germany, 2008; Volume 5251, pp. 306–317.

25. Wu, S.; Manber, U. *A Fast Algorithm for Multi-Pattern Searching*, Technical report. University of Arizona: Tucson, AZ, USA, 1994.