Measuring the Propagation of Information in Partial Evaluation

Henning Korsholm Rohde

BRICS Report Series RS-05-26

ISSN 0909-0878 August 2005

Measuring the Propagation of Information in Partial Evaluation

Henning Korsholm Rohde
BRICS[*], Department of Computer Science
University of Aarhus, Denmark
hense@brics.dk

August 2005

Abstract

We present the first measurement-based analysis of the information propagated by a partial evaluator. Our analysis is based on measuring implementations of string-matching algorithms, based on the observation that the sequence of character comparisons accurately reflects maintained information. Notably, we can easily prove matchers to be different and we show that they display more variety and finesse than previously believed. As a consequence, we are able to pinpoint differences and inaccuracies in many results previously considered equivalent.

Our analysis includes a framework that lets us obtain string matchers – notably the family of Boyer-Moore algorithms – in a systematic formalism-independent way from a few information-propagation primitives. By leveraging the existing research in string matching, we show that the landscape of information propagation is non-trivial in the sense that small changes in information propagation may dramatically change the properties of the resulting string matchers. We thus expect that this work will prove useful as a test and feedback mechanism for information propagation in the development of advanced program transformations, such as GPC or Supercompilation.

[*]Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

Contents

1 Introduction

1.1 Partial evaluation and the Knuth-Morris-Pratt algorithm

Pattern matching in strings has been a traditional catalyst for developments in partial evaluation since Futamura in 1987 challenged existing partial evaluators. To illustrate the power of Generalized Partial Computation, notably the propagation of run-time (dynamic) information as compile-time (static) predicates, he generated the search phase of the ingenious Knuth-Morris-Pratt algorithm from a naive, inefficient algorithm [19]. This example quickly became a prime test of strength for partial evaluators.

String-matching frameworks The success of the Knuth-Morris-Pratt example soon inspired a line of work that focused on string matching. In this line of work, *parameterized* string matchers were constructed, whose instantiations could be specialized into several string matchers. The conjecture was that both the Knuth-Morris-Pratt algorithm and the equally ingenious Boyer-Moore algorithm could be derived naturally from a common framework, with a simple *left-to-right* traversal of the pattern giving rise to the Knuth-Morris-Pratt algorithm and the corresponding *right-to-left* traversal giving rise to the Boyer-Moore algorithm. This "duality" conjecture quickly became folklore [13].

Information propagation and the KMP-test The Knuth-Morris-Pratt example was also used to compare advanced program transformations (including Supercompilation and GPC), thereby coining the term "KMP-test" [41]. A transformation is said to pass the KMP-test if it can generate a linear-time program from a naive – so-called quadratic-time – string matcher. The point was that certain differences in the *information propagated* by each algorithm was distinguished by the test. Hence, techniques that were known to propagate *positive* information about the text (from outcomes of successful character comparisons) were found to pass; techniques that did not, were found to fail.

1.2 Motivation

Under the usual emphasis on improving efficiency (specified by time complexity), an established practice is to distinguish string matchers by traversal-order and search-phase time complexity only. Consequently, only three classes of string matchers are usually considered, although the frameworks attempt to draw finer distinctions:

(1) Naive $O(mn)$ matchers	("Brute-Force-like")
(2) Left-to-right $O(f(m)+n)$ matchers	("Knuth-Morris-Pratt-like")
(3) Right-to-left $O(f(m)+n)$ matchers	("Boyer-Moore-like")

Unfortunately, this convenient but coarse view invites trouble: the classes are not homogeneous with respect to information propagation. Not only does each class include an infinite number of matchers using different amounts of propagated information [2], but also matchers with other quite different theoretical and practical properties. Furthermore, Knuth-Morris-Pratt-like algorithms have been generated from Brute-Force-like ones by techniques phrased in a variety of formalisms and languages, additionally blurring a direct comparison of methods.

One of the merits of the KMP-test was, however, precisely to use a formalism-independent property of string matchers (namely time complexity) as a basis for comparison. Unfortunately, it still leaves us with the heterogeneous classes: knowing only that a transformation passes the test does not tell us much about what information it propagates – only that it must propagate *something*. Consequently, the flurry of successful results are still just as difficult to compare, undermining the benefits of a standard example. In particular, the consequences of propagating so-called *negative* information are not well understood.

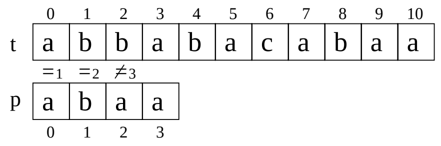
1.3 Overview of this work

The purpose of this work is to better understand information propagation of advanced transformation techniques and its consequences. Our approach is to unify ideas from the KMP-test [41], string matching frameworks [6, 7, 36], and recent analyses [1, 2, 15, 24]. In this overview, we will gloss over some of the technical contributions.

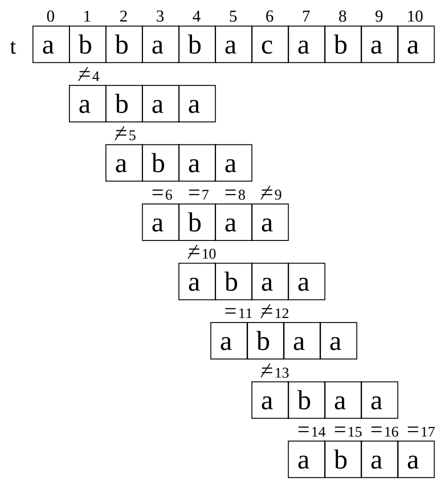
1.3.1 A measure for information propagation

The fundamental question is how to distinguish between string matchers. Inspired by recent analyses, our answer is a domain-specific measure, namely the sequence of text character comparisons during matching. Note that such comparisons are both what advanced program transformations attempt to optimize away by propagating information and a key concern in string matching. As an additional benefit here, the measure conveniently factors out representation issues and how information is actually propagated. When we wish to emphasize these omissions, we will talk about the propagation of *characteristic* information. In contrast to time complexity, the sequence of comparisons is *observable*, that is, we can use test examples to prove that two string matchers are different, as illustrated below.

In this article, string matchers find the first occurrence of a pattern string in a text string. Consider a naive left-to-right ("Brute-Force") algorithm's search for the pattern p=`abaa` in the text t=`abbabacabaa`. The first round of comparisons are:



These three comparisons ended in a mismatch, so the algorithm simply tries successive alignments until an occurrence is found or the text is exhausted:



The first occurrence is found using 17 comparisons between the pattern and the text.

The outcomes of the comparisons can be listed as follows:

p[0]=t[0], p[1]=t[1], p[2]≠t[2], p[0]≠t[1], p[0]≠t[2], p[0]=t[3],
p[1]=t[4], p[2]=t[5], p[3]≠t[6], p[0]≠t[4], p[0]=t[5], p[1]≠t[6],
p[0]≠t[6], p[0]=t[7], p[1]=t[8], p[2]=t[9], p[3]=t[10].

The equalities represent *positive* information, and the inequalities *negative*.

In this context, advanced transformations *propagate* text information and decide the outcome of certain comparisons between the pattern and the text *at transformation time*, using comparisons between the pattern itself. For positive information `p[i]=t[j]`, we can substitute later occurrences of `t[j]` with `p[i]`; then comparisons before involving `t[j]` can be now be performed using the pattern only. For negative information `p[i]`≠`t[j]`, using that if also `p[i]=p[i']` then `p[i']`≠`t[j]`, we can decide applicable later occurrences of `p[i']`≠`t[j]` by `p[i]=p[i']`.

Consider the above example, when we propagate positive information. The comparisons become (underlining decided ones):

p[0]=t[0], p[1]=t[1], p[2]≠t[2], $\underline{\texttt{p[0]}\neq\texttt{p[1]}}$, p[0]≠t[2], p[0]=t[3],
p[1]=t[4], p[2]=t[5], p[3]≠t[6], $\underline{\texttt{p[0]}\neq\texttt{p[1]}}$, $\underline{\texttt{p[0]=p[2]}}$, p[1]≠t[6],
p[0]≠t[6], p[0]=t[7], p[1]=t[8], p[2]=t[9], p[3]=t[10].

If we additionally propagate negative information, we decide even more:

p[0]=t[0], p[1]=t[1], p[2]≠t[2], $\underline{\texttt{p[0]}\neq\texttt{p[1]}}$, $\underline{\texttt{p[0]=p[2]}}$, p[0]=t[3],
p[1]=t[4], p[2]=t[5], p[3]≠t[6], $\underline{\texttt{p[0]}\neq\texttt{p[1]}}$, $\underline{\texttt{p[0]=p[2]}}$, p[1]≠t[6],
$\underline{\texttt{p[0]=p[3]}}$, p[0]=t[7], p[1]=t[8], p[2]=t[9], p[3]=t[10].

Performing the underlined comparisons at transformation time (or, equivalently, in a preprocessing phase) is the key to efficiency: we can now tabulate each sequence of underlined comparisons indexed by the pattern index of the *preceding* mismatch, performing them once and for all.

Doing so leads to two distinct Knuth-Morris-Pratt-like string matchers.

It will now be instructive to consider the real Knuth-Morris-Pratt algorithm [32] (see Appendix A). It performs the following comparisons on the example (underlining comparisons performed in the preprocessing phase):

$$\underline{\texttt{p[1]}\neq\texttt{p[0]}},\ \underline{\texttt{p[1]}\neq\texttt{p[0]}},\ \underline{\texttt{p[2]=p[0]}},\ \underline{\texttt{p[2]=p[0]}},\ \underline{\texttt{p[3]}\neq\texttt{p[1]}},\ \texttt{p[0]=t[0]},$$
$$\texttt{p[1]=t[1]},\ \texttt{p[2]}\neq\texttt{t[2]},\ \texttt{p[0]=t[3]},\ \texttt{p[1]=t[4]},\ \texttt{p[2]=t[5]},\ \texttt{p[3]}\neq\texttt{t[6]},$$
$$\texttt{p[1]}\neq\texttt{t[6]},\ \texttt{p[0]}\neq\texttt{t[6]},\ \texttt{p[0]=t[7]},\ \texttt{p[1]=t[8]},\ \texttt{p[2]=t[9]},\ \texttt{p[3]=t[10]}.$$

Even ignoring underlined comparisons, this sequence differs from the previous ones.

However, if we restricted the additional propagation of negative information to just one character (i.e., not propagating `p[i]`≠`t[j]` beyond the next occurrence of `p[i']`≠`t[j]`, applicable or not), we obtain:

$$\texttt{p[0]=t[0]},\ \texttt{p[1]=t[1]},\ \texttt{p[2]}\neq\texttt{t[2]},\ \underline{\texttt{p[0]}\neq\texttt{p[1]}},\ \underline{\texttt{p[0]=p[2]}},\ \texttt{p[0]=t[3]},$$
$$\texttt{p[1]=t[4]},\ \texttt{p[2]=t[5]},\ \texttt{p[3]}\neq\texttt{t[6]},\ \underline{\texttt{p[0]}\neq\texttt{p[1]}},\ \underline{\texttt{p[0]=p[2]}},\ \texttt{p[1]}\neq\texttt{t[6]},$$
$$\texttt{p[0]}\neq\texttt{t[6]},\ \texttt{p[0]=t[7]},\ \texttt{p[1]=t[8]},\ \texttt{p[2]=t[9]},\ \texttt{p[3]=t[10]}.$$

For every pattern and text, in fact, such a string matcher performs the exact same sequence of non-underlined comparisons as the Knuth-Morris-Pratt algorithm [2]. We can thus view the Knuth-Morris-Pratt algorithm as one that propagates text information in this particular way, linking information propagation strategies with real algorithms.

We notice that picking out the sequence of text indices of the comparisons, which we will call the *trace*, gives us a concise way of measuring string matchers (and scales to bounded static variation [15]). The set of traces is called the *behavior* of a matcher. In summary,

Information propagation strategy	Trace for the example
none	0 1 2 1 2 3 4 5 6 4 5 6 6 7 8 9 10
positive	0 1 2 2 3 4 5 6 6 6 7 8 9 10
positive and one character of negative	0 1 2 3 4 5 6 6 6 7 8 9 10
positive and negative	0 1 2 3 4 5 6 6 7 8 9 10

In particular, the matchers differ in the number of comparisons.

All in all, we have a concrete way of comparing string matchers and in turn transformation techniques, regardless of underlying formalism (requiring only that the sequence of text character comparisons is well-defined). It supplements time complexity, allowing a classification of string matchers along two dimensions: "efficiency" (time complexity) and "information propagation" (behavior).

1.3.2 A framework for information propagation

We are now able to expand the treatment of information propagation, using ideas from string matching frameworks. We generalize the "information-propagation strategies" to a specification language for characteristic information propagation, dropping the inconvenient (implicit) dependence on the Brute-Force string matcher. The language is given an interpretation and provides concise specifications for a wide variety of behaviors.

The real payoff is that by measuring the (implicit) information propagation of *real* string matching algorithms, we can introduce sound canonical naming of specifications and behaviors. Consistent naming is a prerequisite for sound comparison and exchange of existing ideas that involve Knuth-Morris-Pratt-like matchers. Furthermore, we also have plenty of data points for controlling and *guiding* exploration and for analyzing advanced program transformations and frameworks.

1.3.3 An analysis of information propagation

We have now established a sound setting for information propagation (measure and framework), in which we – with little effort – can analyze string matchers and, in turn, program transformations, as inspired by the KMP-test.

Notably, we show that neither Supercompilation nor GPC generate the exact Knuth-Morris-Pratt string matcher by transforming a naive string matcher. In general, too much negative information is propagated which in turn increases the sizes of generated matchers. Although the number of comparisons is reduced, the increase in size may be a major obstacle for linear-time preprocessing by self-application [1].

The Knuth-Morris-Pratt algorithm has received the lion's share of attention in program transformation, despite the diversity of string matchers. For example, the Knuth-Morris-Pratt algorithm – despite its optimal time complexity – does not actually improve over the Brute-Force algorithm in practice (see, e.g., [42, Section 2.1.4]). The Boyer-Moore/*bad-character-shift* variants on the other hand – notably Horspool's asymptotically inefficient algorithm – are considered to be the fastest string matchers in practice. This work sheds light on the situation, allowing program transformations to better tune the quality of outcomes with respect to other properties than time complexity.

Prerequisites A standard introduction to exact string matching algorithms can be found in Appendix A. The core results require some understanding of partial evaluation [13,31]. Familiarity with advanced transformations or frameworks will be useful, in particular Sørensen et al.'s work [41].

The rest of this article is organized as follows: Section 2 describes the measure – or *behavior*– of string matchers; Section 3 describes the string matching framework used for analysis; and Section 4 analyzes advanced program transformations and existing frameworks. Note that the analysis itself does not require an understanding of the details of the framework's construction. Section 5 concludes.

2 A measure for information propagation

The fundamental concept is the observable measure called the *trace.* For any string matcher M, pattern p, and text t, we define *the trace of* M *on* p *and* t – denoted $\|M(p,t)\|_{tr}$ – as the sequence of indices of text character accesses during successful matching.[1] Traces indirectly measure the propagation of *characteristic* information.

We consider only string matchers and formalisms where traces are well-defined. The deterministic, imperative Algol60-like language in which string matching algo-

[1]The "successful" restriction on the traces simply circumvents the inessential detail that overall failure is detected in different ways by different matchers: some stop when pattern no longer fits – others only when a pointer reaches the end of the text. Since we can always just append the pattern to any text, we do not lose precision.

rithms are traditionally phrased is one such formalism, and traces are the sequences of values for `k` in the executions of `text[k]` (see Appendix A). For inspection purposes, it is thus straightforward to wrap `printf`s around each such `k`.

We define the *behavior* of a string matcher M as

$$\|M\| = \lambda(p,t).\, \|M(p,t)\|_{tr}.$$

The behavior of a string matcher is a total function that we cannot observe in its entirety. Hence, we cannot observe that two string matchers are behaviorally equal. We can, however, observe that two string matchers are behaviorally different, by comparing traces for concrete examples. Note that behaviors form an equivalence relation.

It is important to note that the standard properties about text-character comparisons (their number and the delay) outlined in Appendix A are all subsumed by our notion of behavior.[2] Consequently, any existing analysis, such as the accurate bounds on the number of comparisons, will automatically apply to any behaviorally-equivalent string matcher. The distinction between "characteristic information propagation" (behavior) and "efficiency" (time complexity) is essential, as depicted in Figure 2. Similar distinctions have been mentioned informally in the literature.

We can thus substantiate "to *obtain* a string matcher" to mean to generate one with the same behavior and the same (best) worst-case time complexity. When discussing information propagation, we leave "characteristic" implicit.

Methodology For practical purposes, we will only speak of behavioral equality relative to a fixed underlying *test suite* of examples (i.e., pattern and text pairs), which currently consists of 170 examples. We use an implementation of the framework to ensure that all treated matchers' traces coincide with those of their authoritative implementations [10] for this test suite and that no counter-examples are known. This assurance is not as weak as it may seem: given any string matcher, we can prove it different from all but at most one known expressible string matcher, using at most 2 chosen examples (p=`abaa` and t=`abbababacabaa` is the prime one – and adequate for Knuth-Morris-Pratt-like matchers). Minimally, e.g., when analyzing transformations or frameworks by hand, just the above example often suffices.

3 A framework for information propagation

In this section, we present a new string matching framework that foremost establishes consistency in the use of names for generated matchers. Here, only the behavior of string matchers is considered.

3.1 The framework

The framework is a translation, $\mathcal{W}[\![\cdot]\!] : S \to L$, from an inductive specification language to a trace-compliant functional language. The image of the translation will be

[2]Note also that text-character comparisons represent the only *observable* search-phase property. The notion of behavior is thus a natural choice that supplements (worst-case) time complexity, although they are not quite orthogonal.

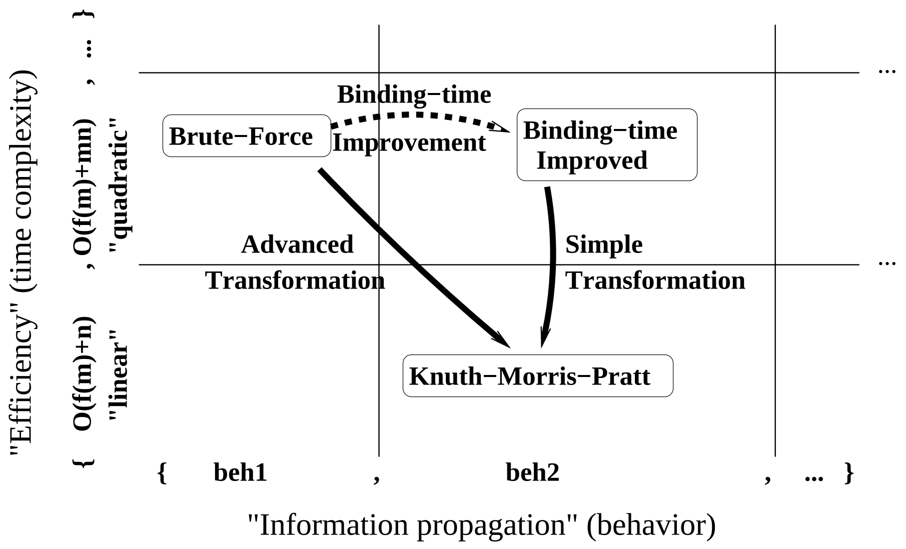


Figure 1: Classification of string matchers

We classify string matchers along two dimensions: "information propagation" (behavior) on the horizontal axis and "efficiency" (best worst-case time complexity) on the vertical axis. The "*f*" in the time complexities should be understood as an arbitrary function, which factors out time consumption during preprocessing or at transformation time. String matchers M are boxed, and a program transformation T from M_1 to M_2 is understood as $T(M_1) = \lambda(p,t).\,T(M_1,p)(t)$, where $T(M_1)$ classify as M_2. Binding-time improvements, in contrast, just map M_1 to M_2.

A more refined classification might take resource consumption during preprocessing and at transformational time into account (see, e.g., Ager et al.'s work [1]). Here we only try to capture *what* information is propagated, glossing over *how*.

string matchers that propagate explicitly-represented information by themselves. We assume a target language with the usual primitives for list and function processing.

The syntax of the specification language is:

$$\begin{array}{rcl}
S & ::= & \textsc{Basic}(O,\ C,\ I^+,\ I^-) \mid \textsc{Backtracking}(S,S) \mid \\
 & & \textsc{Sequential}(S,S) \mid \textsc{Parallel}(S,S) \mid \\
 & & \textsc{Alternate}(S,S) \mid \textsc{Skew}(S,S) \mid \textsc{Fail} \\[6pt]
O & ::= & \textit{left-to-right} \mid \textit{right-to-left} \mid \textit{last-left-to-right} \mid \\
 & & \textit{last-first-middle-rest} \mid \textit{left-to-right-skip-second} \mid \\
 & & \textit{right-to-left-skip-last} \mid \textit{last-only} \mid \textit{second-only} \\
C & ::= & \textit{char} \mid \textit{table} \\
I^+ & ::= & +\textit{all} \mid +\textit{pos} \mid +\textit{none} \\
I^- & ::= & -\textit{all} \mid -\textit{neg} \mid -\textit{none}
\end{array}$$

Elements of S will be called *partial matchers* and will in general not specify a correct string matcher. The specifications that do are called *complete* and simply have orders that "span" the whole pattern. It should be intuitively clear that complete matchers are simply those that trivially guarantee to test all characters in the pattern, before declaring a match (a formal proof of correctness is, however, beyond the scope of this paper, see [7]).

We need some auxiliary notions. Foremost, each Basic node needs a separate representation of propagated text information (positive and negative) – usually called a cache, so we use a *cache tree* to store these and their relative alignments:

$$\begin{aligned} T ::= \;& \textsc{Single}((\Sigma + \Sigma\ \mathit{list})\ \mathit{list}) \mid \textsc{Aligned}(T,T) \mid \\ & \textsc{Non-aligned}(T,T,\mathbb{N}) \mid \textsc{None} \end{aligned}$$

Each partial matcher induces a similarly-shaped cache tree, along with two auxiliary functions (where $\mathbb{B} = \{\mathtt{tt}, \mathtt{ff}\}$ is the set of truth values):

$$\begin{array}{rl} \text{init} & \mathcal{T}[\![s]\!] : \mathbb{N} \to T \\ \text{match} & \mathcal{M}[\![s]\!] : \mathbb{N} \times T \to \mathbb{B} \times \mathbb{N} \times T \\ \text{shift} & \mathcal{S}[\![s]\!] : \mathbb{N} \times T \to T \end{array}$$

Informally, $\mathcal{T}[\![\cdot]\!]$ creates a cache tree; $\mathcal{M}[\![\cdot]\!]$ performs a matching attempt at a given text alignment, updating the cache tree appropriately; and $\mathcal{S}[\![\cdot]\!]$ re-aligns the cache tree after a mismatch.

The framework induces a simple bounded search. For any complete specification s, we define

$$\mathcal{W}[\![s]\!] = \text{search}(0, \mathcal{T}[\![s]\!](m))$$

where

$$\text{search}(k,t) = \begin{array}{l} \text{case } (k + m < n) \\ \text{of } \begin{cases} \mathtt{tt} \to \text{case } \mathcal{M}[\![s]\!](k,t) \text{ of } \begin{cases} (\mathtt{tt}, j, t') \to k \\ (\mathtt{ff}, j, t') \to \text{search}(k + j, \mathcal{S}[\![s]\!](j, t')) \end{cases} \\ \mathtt{ff} \to -1 \end{cases} \end{array}$$

The function search successively calls $\mathcal{M}[\![s]\!]$ at each text alignment k, stopping only when the text is exhausted (returns -1) or when an occurrence of the pattern has been located in the text (returns k). Note that $\mathcal{M}[\![s]\!]$ explicitly returns the number of text alignments that can be safely skipped, j.

3.1.1 The Basic partial matcher

The Basic partial matcher is the workhorse of the framework and builds directly on Amtoft et al.'s work [7], but supports finer control of propagated text information. It has four components:

- o : the *order* in which to examine the text characters;
- c : the *method of comparison* to be used to extract information;
- i^+ : what information to *propagate* from each comparison; and
- i^- : what information to *prune* after each round of comparisons.

Order: An order produces a list of indices of non-positive entries in the cache. This (possibly incomplete) list is then used to schedule comparisons, where re-comparison of characters already known to be identical is avoided. The orders considered are:

Order	Selection of the indices 0 1 2 ... m-1
left-to-right	$0 :: 1 :: 2 :: 3 :: \cdots :: m\text{-}1$
right-to-left	$m\text{-}1 :: m\text{-}2 :: \cdots :: 0$
last-left-to-right	$m\text{-}1 :: 0 :: 1 :: 2 :: 3 :: \cdots :: m\text{-}2$
last-first-middle-rest	$m\text{-}1 :: 0 :: m/2 :: 1 :: 2 :: \cdots :: m\text{-}2$
left-to-right-skip-second	$2 :: 3 :: \cdots :: m\text{-}1 :: 0$
right-to-left-skip-last	$m\text{-}2 :: m\text{-}3 :: \cdots :: 0$
last-only	m-1
second-only	1

where m is the length of the cache (and thus also the pattern). We denote the selection of indices for any order o by $\overline{o}$.

We define $\mathcal{O}[\![o]\!] : (\Sigma + \Sigma\ \mathit{list})\ \mathit{list} \to \mathbb{N}\ \mathit{list}$ by

$$\mathcal{O}[\![o]\!](c) = \text{smap } (\lambda i.\ \text{case } \mathit{cad}^i r(c) \text{ of } \begin{Bmatrix} \mathit{in}_1(p) \to [] \\ \mathit{in}_2(n) \to [i] \end{Bmatrix})\ \overline{o}$$

where

$$\text{smap } f\ l = \text{case } l \text{ of } \begin{cases} [] & \to [] \\ i :: l' & \to f(i) +\!\!+ (\text{smap } f\ l') \end{cases}$$

Comparison: A comparison determines whether a text character and a pattern character are equal. Additionally, it produces explicit knowledge about the text character, which can be either positive ('is x') or negative ('is not x'), where 'x' is a non-text (i.e., static) character. The standard character comparison, *char*, is indeed what we expect, where 'x' is the character from the pattern.

If the alphabet, Σ, is assumed to be known and finite, we have a much stronger way of acquiring text knowledge, namely the use of *bounded static variation* [31]. This kind of comparison will always produce positive information ('is x') about a text character, where 'x' is taken from the alphabet. Although bounded static variation could be implemented simply by a large **case**-statement over the alphabet, a key observation is that the constant-time character-key tables – such as the table for the *bad-character-shift* heuristic – play the exact same role in real algorithms [15]. For this reason we call it a *table* comparison.

We define $\mathcal{C}[\![c]\!] : \Sigma \times \Sigma \to \mathbb{B} \times (\Sigma + \Sigma)$ by

$$\begin{aligned} \mathcal{C}[\![\mathit{char}]\!](x,y) &= \text{case } (x=y) \text{ of } \begin{cases} \mathtt{tt} \to (\mathtt{tt}, \mathit{in}_1(x)) \\ \mathtt{ff} \to (\mathtt{ff}, \mathit{in}_2(x)) \end{cases} \\ \mathcal{C}[\![\mathit{table}]\!](x,y) &= \text{case } y \text{ of } \begin{cases} a_1 & \to ((x = a_1), \mathit{in}_1(a_1)) \\ & \vdots \\ a_{|\Sigma|} & \to ((x = a_{|\Sigma|}), \mathit{in}_1(a_{|\Sigma|})) \end{cases} \end{aligned}$$

assuming $\Sigma = \{a_1, \ldots, a_{|\Sigma|}\}$ in the latter clause.

Propagator: A propagator determines what collected positive and/or negative information to place in the cache. We use an empty list of negative information to represent "no information". We first define $\overline{i^{+}} : (\Sigma + \Sigma) \to (\Sigma + \Sigma)$ by

$$\overline{+\mathit{all}}(i) = i \qquad \overline{+\mathit{none}}(i) = \mathit{in}_2([]) \qquad \overline{+\mathit{pos}}(i) = \text{case } i \text{ of } \begin{cases} \mathit{in}_1(x) \to i \\ \mathit{in}_2(x) \to \mathit{in}_2([]) \end{cases}$$

We then define $\mathcal{I}^+[\![i^+]\!] : (\Sigma + \Sigma\ list)\ list \times (\Sigma + \Sigma) \times \mathbb{N} \to (\Sigma + \Sigma\ list)\ list$ by

$$\mathcal{I}^+[\![i^+]\!](c, i, k) = \text{upd}\ c\ (\overline{i^+}(i))\ k$$

where upd updates the kth entry of the cache c with information $(\overline{i^+}(i))$:

$$\text{upd}\ (e :: c')\ i\ k = \begin{array}{l} \text{case } (k = 0) \\ \text{of} \begin{cases} \mathtt{tt} \to \text{case } (e, i) \text{ of } \begin{cases} (in_2(l), in_1(p)) \to i :: c' \\ (in_2(l), in_2(l')) \to in_2(l +\!\!+ l') :: c' \end{cases} \\ \mathtt{ff} \to e :: (\text{upd}\ c'\ i\ (k-1)) \end{cases} \end{array}$$

Pruner: A pruner determines what collected positive and/or negative information to erase from the cache after a mismatch. We first define $\overline{i^-} : (\Sigma+\Sigma\ list) \to (\Sigma+\Sigma\ list)$ by

$$\overline{-all}(i) = in_2([]) \qquad \overline{-none}(i) = i \qquad \overline{-neg}(i) = \text{case } i \text{ of } \begin{cases} in_1(x) & \to i \\ in_2(xs) & \to in_2([]) \end{cases}$$

We then define $\mathcal{I}^-[\![i^-]\!] : (\Sigma + \Sigma\ list)\ list \to (\Sigma + \Sigma\ list)\ list$ by

$$\mathcal{I}^-[\![i^-]\!](c) = \text{map}\ (\lambda i.\overline{i^-}(i))\ c$$

Propagators and pruners thus act as filters and control the quantity of propagated and maintained information; comparisons control the quality.

We now define

$$\mathcal{M}[\![\textsc{Basic}(o,\ c,\ i^+,\ i^-)]\!](k, \textsc{Single}(c')) = \text{loop}(\mathcal{O}[\![o]\!](c'), c')$$

where

$$\text{loop}(l, c') = \text{case } l \text{ of } \begin{cases} j :: l' \to \begin{cases} \text{let } (b, i) = \mathcal{C}[\![c]\!](pattern[j], text[k+j]) \\ \qquad c'' \ = \mathcal{I}^+[\![i^+]\!](c', i, j) \\ \text{in case } b \text{ of } \begin{cases} \mathtt{tt} \to \text{loop}(l', c'') \\ \mathtt{ff} \to (\mathtt{ff}, \text{calc}(c''), \textsc{Single}(\mathcal{I}^-[\![i^-]\!](c''))) \end{cases} \end{cases} \\ [] \quad \to (\mathtt{tt}, \text{calc}(c'), \textsc{Single}(\mathcal{I}^-[\![i^-]\!](c'))) \end{cases}$$

$\mathcal{M}[\![\textsc{Basic}(o,\ c,\ i^+,\ i^-)]\!]$ decides whether or not $pattern[j] = text[k+j]$ for all $j \in \overline{o}$, while maintaining exploitable information about the text. It works as follows, given an offset into the text k and a cache of (assumed to be correct) information about the text starting from k:

1. the order $\mathcal{O}[\![o]\!](c')$ produces a list l of indices (relative to k) of text characters that we have not yet gathered sufficient information about to determine whether or not $pattern[j] = text[k+j]$ for all $j \in \overline{o}$;

2. loop then process the elements j of this list.

   (a) First, $pattern[j]$ and $text[k+j]$ are compared, and any information propagated by $\mathcal{I}^+[\![i^+]\!]$ is inserted in the cache.

   (b) If the comparison succeeds, we continue processing the list.

(c) If the comparison fails, then the current j is a witness that $\mathit{pattern}[j'] = \mathit{text}[k+j']$ for all $j' \in \overline{o}$ does not hold.

However, we might also already be able to decide that the condition cannot hold for some of the subsequent $k' > k$, by exploiting the information in the cache c''; $\mathrm{calc}(c'')$ does so and returns the largest $h > 0$ such that for all $k + h'$, where $h > h' > 0$, we have a witness in the cache that proves that $k + h'$ does not satisfy the condition.

After exploiting the information of the cache, we prune it by $\mathcal{I}^-[\![i^-]\!]$.

3. If the list is or becomes empty, $\mathit{pattern}[j] = \mathit{text}[k+j]$ for all $j \in \overline{o}$ holds. Again we prune the cache, and, for scalability, we also return $\mathrm{calc}(c'')$.

Note that text character accesses (namely $\mathit{text}[k+j]$) are performed in this loop; given a pattern and text, their trace is the sequence of '$k+j$' values.

We also have:

$$\begin{aligned}
\mathcal{T}[\![\textsc{Basic}(o,\ c,\ i^+,\ i^-)]\!](m) &= \textsc{Single}(\overbrace{\mathit{in}_2([]) :: \cdots :: \mathit{in}_2([])}^{m \text{ times}}) \\
\mathcal{S}[\![\textsc{Basic}(o,\ c,\ i^+,\ i^-)]\!](j, \textsc{Single}(c')) &= \textsc{Single}(\mathit{cd}^j\, r(c') +\!\!+ \underbrace{(\mathit{in}_2([]) :: \cdots :: \mathit{in}_2([]))}_{j \text{ times}})
\end{aligned}$$

As examples of Basic matchers, the specifications of the Brute-Force and Knuth-Morris-Pratt algorithms are

Brute-Force	Basic(*left-to-right*, *char*, +*none*, −*none*)
Knuth-Morris-Pratt	Basic(*left-to-right*, *char*, +*all*, −*neg*)

whose information propagation is illustrated in Figures 3.1.1 and 3.1.1.

Note that the four parameters for Basic matchers are not completely orthogonal, so there may be more than one way to specify such matchers. Their virtue is, however, that they concisely capture many matchers' information propagation.

3.1.2 Composite partial matchers

A basic design idea is that simpler partial matchers – in particular Basic partial matchers – can directly specify *heuristics*, such as the *bad-character-shift* heuristic. We call the non-Basic partial matchers for combinators.

Backtracking, Sequential, and Parallel are particularly simple binary combinators. Here, $\mathcal{M}[\![\cdot(s_1, s_2)]\!]$ uses just the boolean outcome of $\mathcal{M}[\![s_1]\!]$ to decide whether or not to use $\mathcal{M}[\![s_2]\!]$. We define:

$$\begin{aligned}
&\mathcal{M}[\![\textsc{Backtracking}(s_1, s_2)]\!](k, \textsc{Aligned}(t_1,t_2)) = \\
&\quad \text{case } \mathcal{M}[\![s_1]\!](k, t_1) \text{ of } \begin{cases} (\mathtt{tt}, j_1, t_1') \to (\mathtt{tt}, j_1, \textsc{Aligned}(t_1',t_2)) \\ (\mathtt{ff}, j_1, t_1') \to \begin{cases} \text{let } (b_2, j_2, t_2') = \mathcal{M}[\![s_2]\!](k, t_2) \\ \text{in } \ (\mathtt{ff}, \max(j_1, j_2), \textsc{Aligned}(t_1',t_2')) \end{cases} \end{cases}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{M}[\![\textsc{Sequential}(s_1, s_2)]\!](k, \textsc{Aligned}(t_1,t_2)) = \\
&\quad \text{case } \mathcal{M}[\![s_1]\!](k, t_1) \text{ of } \begin{cases} (\mathtt{tt}, j_1, t_1') \to \begin{cases} \text{let } (b_2, j_2, t_2') = \mathcal{M}[\![s_2]\!](k, t_2) \\ \text{in } \ (b_2, \max(j_1, j_2), \textsc{Aligned}(t_1',t_2')) \end{cases} \\ (\mathtt{ff}, j_1, t_1') \to (\mathtt{ff}, j_1, \textsc{Aligned}(t_1',t_2)) \end{cases}
\end{aligned}$$

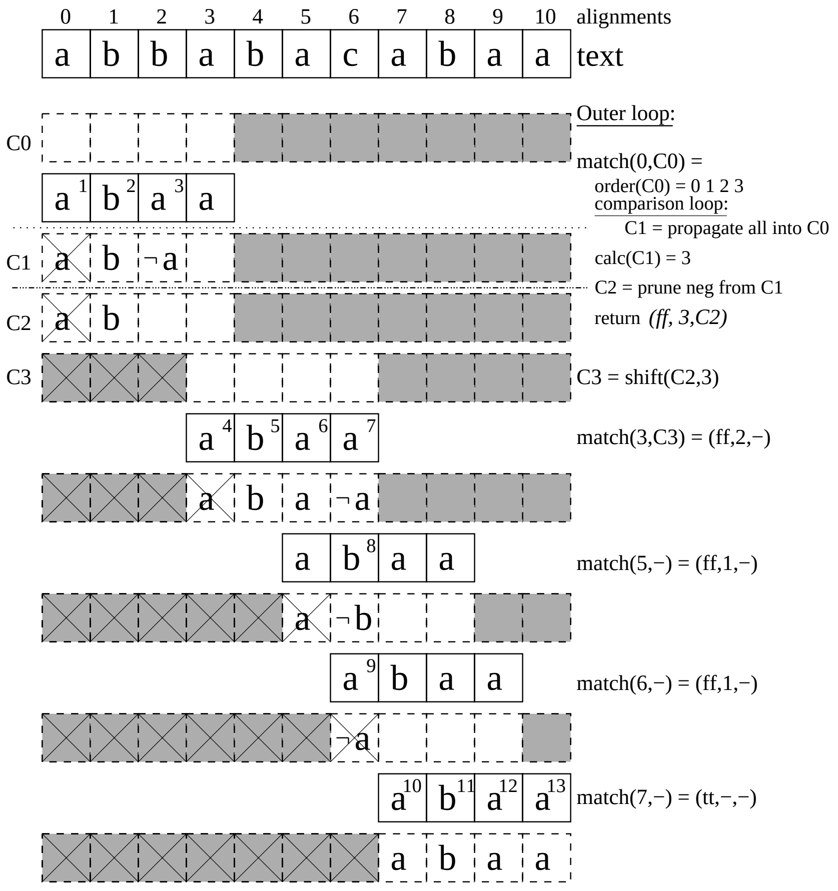


Figure 2: The Knuth-Morris-Pratt algorithm

We illustrate how the Knuth-Morris-Pratt algorithm propagates information by searching for the pattern string `abaa` in the text string `abbabacabaa`. The figure shows the contents of the cache during matching (shown as dashed boxes with a white 'window'). A letter in the cache, like "a" , represents propagated positive information about the text; a negated letter, like "¬a", represents negative information. An "X" represents that the algorithm knows that the text cannot contain the pattern at this position – this information is implicitly propagated by the primitives.

The first round of comparisons, `match(0,C0)`, is shown in detail; the cache marked with `C1`, which is used by calc(·), is the essential one, since it contains the exact information exploited. The other rounds are abbreviated with only the essential caches included. The final cache is purely cosmetic.

$$\mathcal{M}[\![\textsc{Parallel}(s_1,s_2)]\!](k,\textsc{Aligned}(t_1,t_2)) = \begin{cases} \text{let } (b_1,j_1,t_1') = \mathcal{M}[\![s_1]\!](k,t_1) \\ \quad\;\; (b_2,j_2,t_2') = \mathcal{M}[\![s_2]\!](k,t_2) \\ \text{in } \; (b_1 \wedge b_2, \max(j_1,j_2), \textsc{Aligned}(t_1',t_2')) \end{cases}$$

Also, for $B \in \{\textsc{Backtracking},\textsc{Sequential},\textsc{Parallel}\}$,

$$\begin{aligned} \mathcal{T}[\![\mathrm{B}(s_1,s_2)]\!](m) &= \textsc{Aligned}(\mathcal{T}[\![s_1]\!](m),\mathcal{T}[\![s_2]\!](m)) \\ \mathcal{S}[\![\mathrm{B}(s_1,s_2)]\!](j,\textsc{Aligned}(t_1,t_2)) &= \textsc{Aligned}(\mathcal{S}[\![s_1]\!](j,t_1),\mathcal{S}[\![s_2]\!](j,t_2)) \end{aligned}$$

As an example of a Backtracking matcher, consider Horspool's simplification [28]

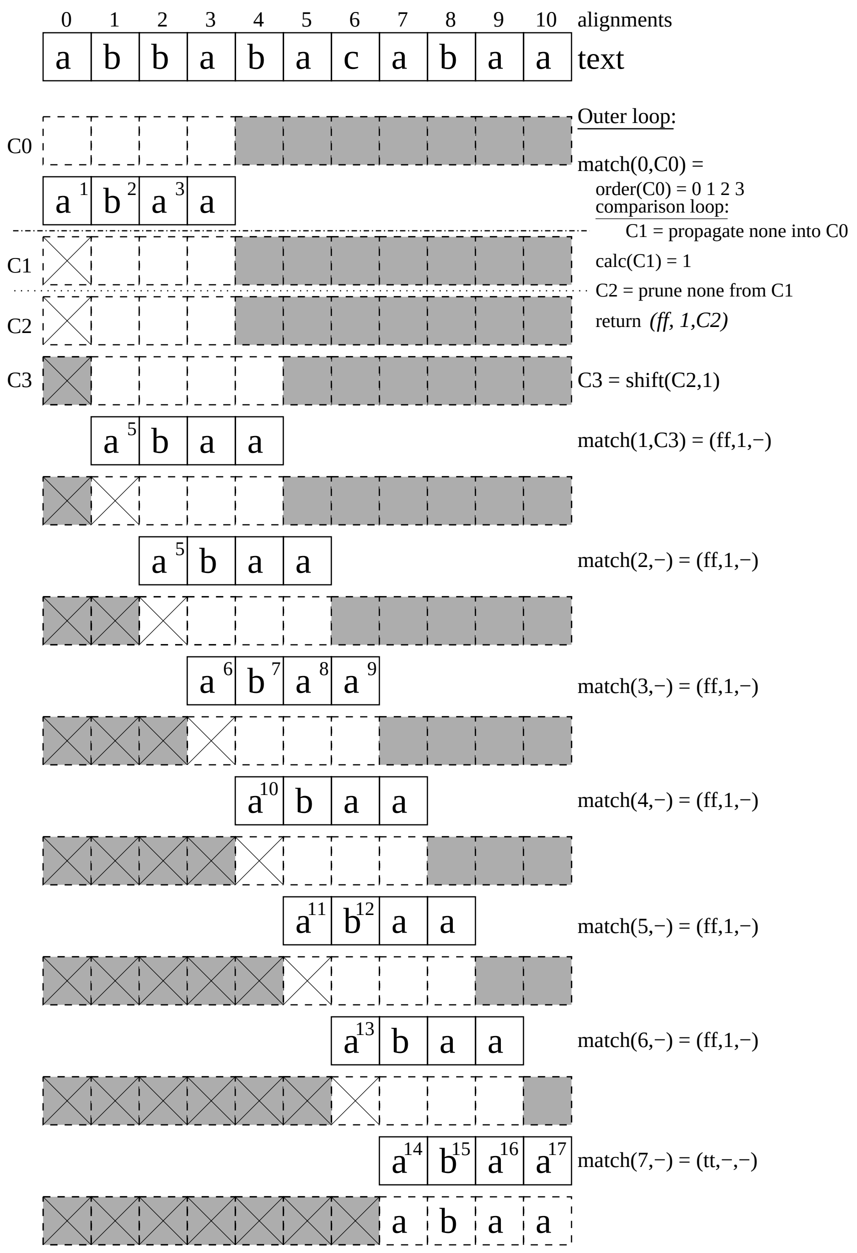


Figure 3: The Brute-Force algorithm

We illustrate how the Brute-Force algorithm propagates information by searching for the pattern string `abaa` in the text string `abbabacabaa`. Here, nothing is propagated and the returned offset is always the trivially safe offset 1.

of the Boyer-Moore algorithm

$$\text{Horspool} \quad \begin{array}{l} \textsc{Backtracking} \\ \text{— } \textsc{Basic}(\textit{last-left-to-right},\ \textit{char},\ +\textit{none},\ -\textit{none}) \\ \text{— } \textsc{Basic}(\textit{last-only},\ \textit{table},\ +\textit{all},\ -\textit{all}) \end{array}$$

whose information propagation is illustrated in Figure 3.1.2; the latter Basic partial matcher specifies the *bad-character-shift* heuristic.

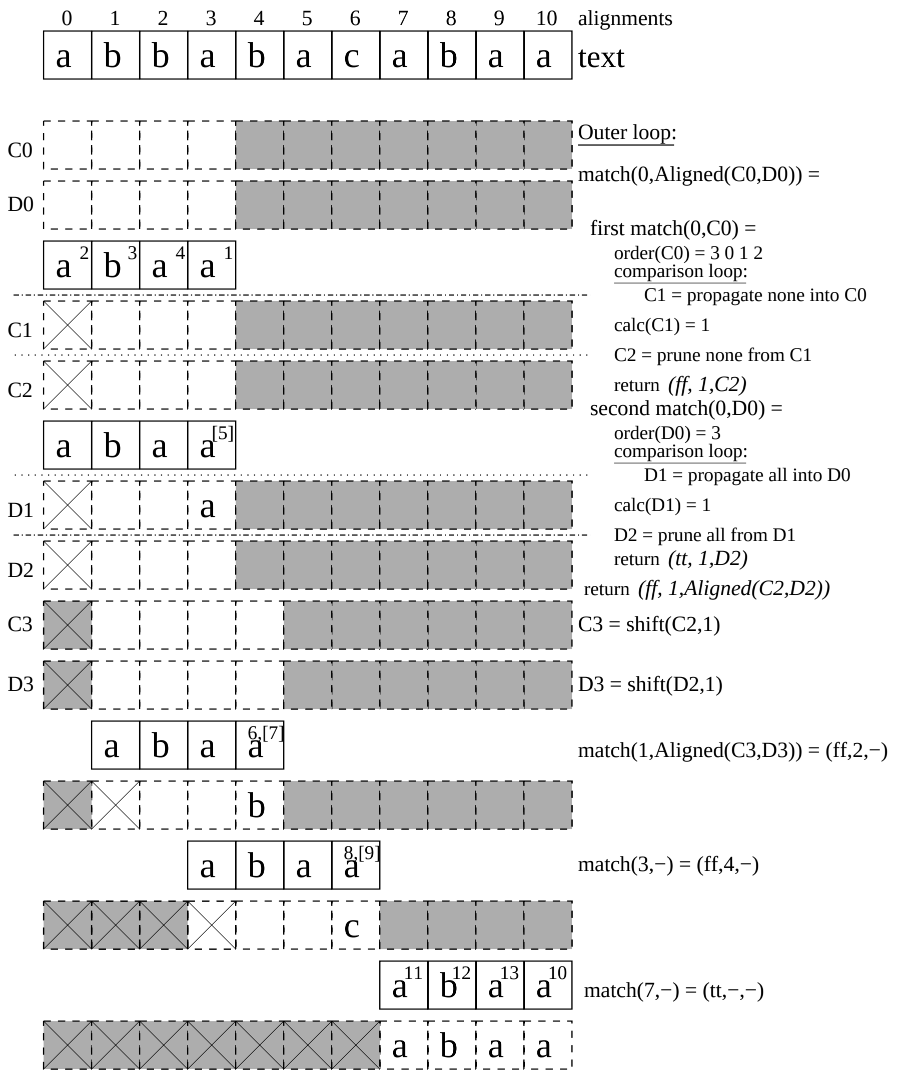


Figure 4: Horspool's algorithm

We illustrate how Horspool's algorithm propagates information by searching for the pattern string `abaa` in the text string `abbabacabaa`. In the abbreviated rounds, the content of the always empty cache is omitted. The bracketed comparisons indicate *table* comparisons, which here demonstrate the behavior of the *bad-character-shift* heuristic. Note in particular comparison 9, which is followed by a situation that allows the *bad-character-shift* heuristic to return an optimal safe shift.

So far, all caches have been aligned. However, several algorithms with advanced backtracking require misaligned caches to properly account for their information propagation.

The ALTERNATE combinator is a refinement of BACKTRACKING and allows two partial matchers to be used alternately, where the second matcher is shifted before used:

$$\begin{array}{l}
\mathcal{M}[\![\textsc{Alternate}(s_1,s_2)]\!](k,\textsc{Aligned}(t_1{,}t_2)) = \\
\quad \text{case } \mathcal{M}[\![s_1]\!](k,t_1) \\
\quad \text{of} \begin{cases} (\mathtt{tt},j_1,t_1') \rightarrow (\mathtt{tt},j_1,\textsc{Aligned}(t_1'{,}t_2)) \\ (\mathtt{ff},j_1,t_1') \rightarrow \begin{cases} \text{let } (b_2,j_2,t_2') = \mathcal{M}[\![s_2]\!](k+j_1,\mathcal{S}[\![s_2]\!](j_1,t_2)) \\ \text{in case } b_2 \text{ of} \begin{cases} \mathtt{tt} \rightarrow (\mathtt{ff},j_1,\textsc{Non-aligned}(t_1'{,}t_2'{,}j_1)) \\ \mathtt{ff} \rightarrow (\mathtt{ff},j_1+j_2,\textsc{Non-aligned}(t_1'{,}t_2'{,}j_1)) \end{cases} \end{cases} \end{cases}
\end{array}$$

$$\begin{array}{l}
\mathcal{T}[\![\textsc{Alternate}(s_1,s_2)]\!](m) = \textsc{Aligned}(\mathcal{T}[\![s_1]\!](m){,}\mathcal{T}[\![s_2]\!](m)) \\
\mathcal{S}[\![\textsc{Alternate}(s_1,s_2)]\!](j,t) = \\
\quad \text{case } t \\
\quad \text{of} \begin{cases} \textsc{Aligned}(t_1{,}t_2) \rightarrow \textsc{Aligned}(\mathcal{S}[\![s_1]\!](j,t_1){,}\mathcal{S}[\![s_2]\!](j,t_2)) \\ \textsc{Non-aligned}(t_1{,}t_2{,}\mathrm{a}) \rightarrow \textsc{Aligned}(\mathcal{S}[\![s_1]\!](j,t_1){,}\mathcal{S}[\![s_2]\!](j\text{-}a,t_2)) \end{cases}
\end{array}$$

An example of an ALTERNATE matcher is Sunday's Quick Search algorithm

	ALTERNATE
Quick Search	— BASIC(*left-to-right*, *char*, $+none$, $-none$)
	— BASIC(*last-only*, *table*, $+all$, $-all$)

whose information propagation is illustrated in Figure 3.1.2.

The SKEW combinator is somewhat particular to the Boyer-Moore algorithm, which uses it as an optimization. It is a refinement of BACKTRACKING, where the second partial matcher is right-aligned to the *mismatch position* of the first. For simplicity, we require that the first partial matcher is BASIC and that the second prunes everything. First, we modify the auxiliary 'loop' function used by $\mathcal{M}[\![\textsc{Basic}(\cdot,\cdot,\cdot,\cdot)]\!]$ to return the mismatch position as well:

$$\begin{array}{l}
\text{loop}'(c,i^+,i^-,k,l,c') = \\
\quad \text{case } l \text{ of} \begin{cases} j :: l' \rightarrow \begin{cases} \text{let } (b,i) = \mathcal{C}[\![c]\!](pattern[j],text[k+j]) \\ \qquad c'' \;\; = \mathcal{I}^+[\![i^+]\!](c',i,j) \\ \text{in case } b \text{ of} \begin{cases} \mathtt{tt} \rightarrow \text{loop}'(c,i^+,i^-,k,l',c'') \\ \mathtt{ff} \rightarrow (\mathtt{ff},\text{calc}(c''),\textsc{Single}(\mathcal{I}^-[\![i^-]\!](c'')),j) \end{cases} \end{cases} \\ [\,] \quad \rightarrow (\mathtt{tt},\text{calc}(c'),\textsc{Single}(\mathcal{I}^-[\![i^-]\!](c')),-1) \end{cases}
\end{array}$$

We then define

$$\begin{array}{l}
\mathcal{M}[\![\textsc{Skew}(\textsc{Basic}(o,\ c,\ i^+,\ i^-){,}s_2)]\!](k,\textsc{Aligned}(\textsc{Single}(c'){,}t_2)) = \\
\quad \text{case } (\text{loop}'(c,i^+,i^-,k,\mathcal{O}[\![o]\!](m),c')) \\
\quad \text{of} \begin{cases} (\mathtt{tt},j_1,t_1',p) \rightarrow (\mathtt{tt},j_1,\textsc{Aligned}(t_1'{,}t_2)) \\ (\mathtt{ff},j_1,t_1',p) \rightarrow \begin{cases} \text{let } (b_2,j_2,t_2') = \mathcal{M}[\![s_2]\!](k-m+p+1,t_2) \\ \text{in } (\mathtt{ff},\max(j_1,j_2-m+p+1),\textsc{Aligned}(t_1'{,}t_2')) \end{cases} \end{cases} \\
\mathcal{T}[\![\textsc{Skew}(s_1,s_2)]\!](m) = \textsc{Aligned}(\mathcal{T}[\![s_1]\!](m){,}\mathcal{T}[\![s_2]\!](m)) \\
\mathcal{S}[\![\textsc{Skew}(s_1,s_2)]\!](j,\textsc{Aligned}(t_1{,}t_2)) = \textsc{Aligned}(\mathcal{S}[\![s_1]\!](j,t_1){,}\mathcal{S}[\![s_2]\!](j,t_2))
\end{array}$$

The prime example of a SKEW matcher is of course the Boyer-Moore algorithm, whose specification is

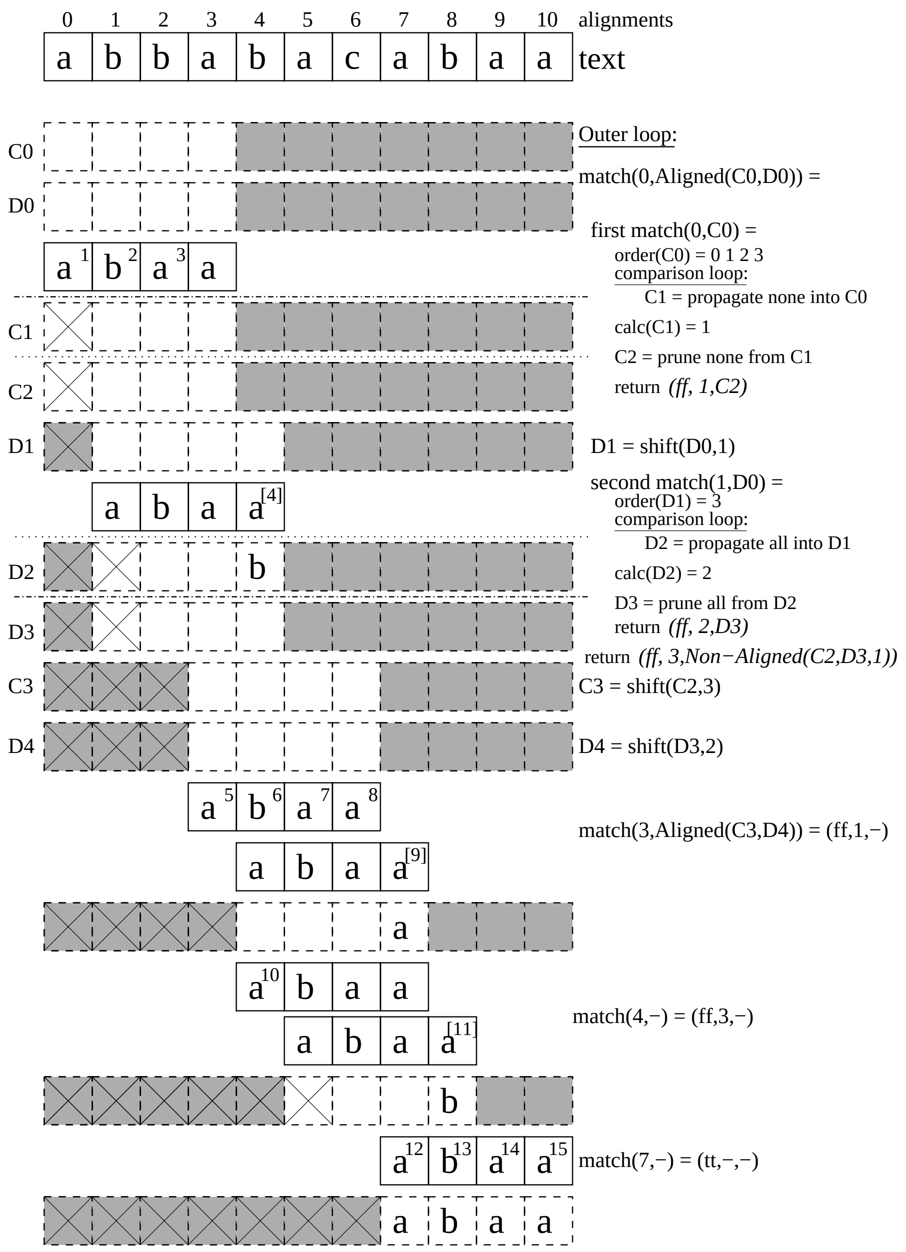


Figure 5: The Quick Search algorithm

We illustrate how the Quick Search algorithm propagates information by searching for the pattern string `abaa` in the text string `abbabacabaa`. Note how the ALTERNATE combinator is used to access the character just past the pattern.

	SKEW
Boyer-Moore	— BASIC(*right-to-left*, *char*, +*all*, −*all*)
	— BASIC(*last-only*, *table*, +*all*, −*all*)

and whose information propagation is illustrated in Figure 3.1.2.

Finally, we will find it useful to have a trace-wise neutral partial matcher FAIL,

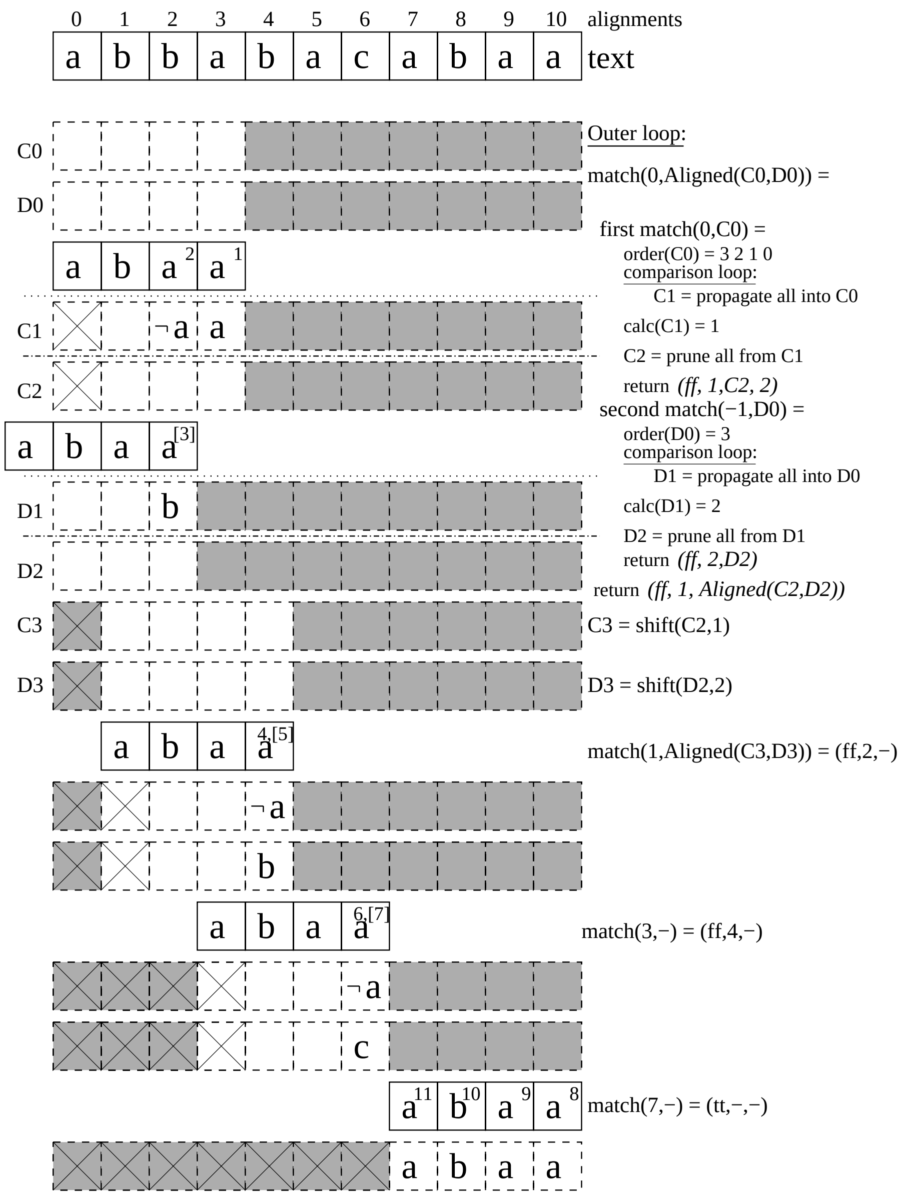


Figure 6: The Boyer-Moore algorithm

We illustrate how the Boyer-Moore algorithm propagates information by searching for the pattern string `abaa` in the text string `abbabacabaa`. As is clear from the first round of comparisons, the SKEW combinator opens the door for potentially unsound situations. Note also the use of non-trivial disjoint information.

which just fails immediately:

$$\mathcal{M}[\![\textsc{Fail}]\!](k, \textsc{None}) = (\text{ff}, 1, \textsc{None})$$

$$\mathcal{T}[\![\textsc{Fail}]\!](m) = \textsc{None} \quad \mathcal{S}[\![\textsc{Fail}]\!](j, \textsc{None}) = \textsc{None}$$

Still, the behavior of many advanced matchers depends on properties of the pattern or external knowledge. For instance, the Maximal Shift uses an order that depends on the pattern and the Optimal Mismatch uses an order that depends on the character frequencies in typical English text [43]. Specifying such matchers would require the framework to be extended with suitable combinators.

3.2 A taxonomy of information propagation

The new framework provides us with concise specifications of string matchers, allowing us to group and identify string matchers according to how information is propagated (cf. other taxonomies [30, 46]). A benefit of using an explicit representation of text information is that often it is easy to estimate the *size* of the corresponding (efficient) tabulated/specialized matchers, namely the number of exploited cache tree instances. An upper such bound is called a "cache bound" in the following.

3.2.1 The Basic matchers

The simplest matcher is the Brute-Force algorithm, which propagates no information (see also Figure 3.1.1). Its specification is

Brute-Force	Basic(*left-to-right*, *char*, $+$*none*, $-$*none*)

To exhibit the behavioral differences between the various matchers, we will search for the pattern `abaa` in the text `abbabacabaa` (non-progress is underlined; bracketed numbers indicate *table* comparisons). Notice the often fine connection between the cache bound and actual number of caches in the previous Figures.

Name	Trace	Cache bound
Brute-Force	$0\ 1\ 2\ \underline{1\ 2}\ 3\ 4\ 5\ 6\ \underline{4\ 5\ 6\ 6}\ 7\ 8\ 9\ 10$	1

The Knuth-Morris-Pratt group The Morris-Pratt and Knuth-Morris-Pratt algorithms gain their efficiency from maintaining positive information. This use has been called the "KMP method", and such matchers differ in the exploitation of negative information (i.e., the *unsuccessful* comparison outcomes):

(noname#1)	Basic(*left-to-right*, *char*, $+$*all*, $-$*none*)
Knuth-Morris-Pratt	Basic(*left-to-right*, *char*, $+$*all*, $-$*neg*)
Morris-Pratt	Basic(*left-to-right*, *char*, $+$*pos*, $-$*none*)

These differences show up in the trace:

Name	Trace	Cache bound
(noname#1)	$0\ 1\ 2\ 3\ 4\ 5\ 6\ \underline{6}\ 7\ 8\ 9\ 10$	$\min(\lvert pattern\rvert^2, \lvert pattern\rvert \cdot \lvert\Sigma\rvert)$
Knuth-Morris-Pratt	$0\ 1\ 2\ 3\ 4\ 5\ 6\ \underline{6\ 6}\ 7\ 8\ 9\ 10$	$\lvert pattern\rvert$
Morris-Pratt	$0\ 1\ 2\ \underline{2}\ 3\ 4\ 5\ 6\ \underline{6\ 6}\ 7\ 8\ 9\ 10$	$\lvert pattern\rvert$

Compared to the Knuth-Morris-Pratt example in Figure 3.1.1, the Morris-Pratt algorithm must perform an extra comparison before comparison 4, because it does not remember that the 3rd text character is not an "a". Similarly, the (noname#1) algorithm will skip comparison 9, because it will continue to remember the outcome of comparison 7. Note that these three matchers are the ones from the introduction.

The unnamed (noname#1) matcher has previously been considered in partial evaluation circles [2, 7, 24]. It stores extra information that can only be exploited on *repeated* mismatches, and it is conjectured that it only uses twice the space of the Knuth-Morris-Pratt algorithm [2]. However, using only that much space requires it to use a more complicated data structure, namely an array of lists. For comparison with the original Knuth-Morris-Pratt algorithm, Figure 7 shows the (noname#1) matcher phrased as a traditional algorithm.

The Boyer-Moore/*good-suffix* heuristic group Boyer and Moore's *good-suffix* heuristic is in fact a linear-time string matcher on its own [9, 32]. It is also the most popular Boyer-Moore variant in the program generation community [7, 16, 26]. The linearity relies here on the use of one character of negative information (cf. the original, quadratic version [9, page 771]):

Good Suffix	Basic(*right-to-left*, *char*, $+$*all*, $-$*all*)
Original GS	Basic(*right-to-left*, *char*, $+$*pos*, $-$*all*)

Their traces on the example are:

Name	Trace	Cache bound
Good Suffix	3 2 4 6 8 10 9 8 7	$\lvert \mathit{pattern} \rvert$
Original GS	$3\ 2\ 4\ 5\ \underline{4}\ 6\ 7\ \underline{6}\ 8\ 9\ \underline{8}\ 10\ \underline{9\ 8\ 7}$	$\lvert \mathit{pattern} \rvert$

The Automaton group Finally, we consider the Automaton and other matchers that rely solely on *table* comparisons. Since no negative information is ever generated, such matchers probe each text character at most once as long as positive information is not pruned. The deterministic finite automaton is the prime example, but *right-to-left* "automata" have also been considered as variants of the Boyer-Moore algorithm [9, 32, 35]:

Automaton	Basic(*left-to-right*, *table*, $+$*all*, $-$*none*)
Optimal BM	Basic(*right-to-left*, *table*, $+$*all*, $-$*none*)
Partsch-Stomp	Basic(*right-to-left*, *table*, $+$*all*, $-$*all*)

These matchers trade time for space, although an accurate bound for the Optimal BM algorithm is not known:

Name	Trace	Cache bound
Automaton	[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]	$\lvert\Sigma\rvert \cdot \lvert \mathit{pattern} \rvert$
Optimal BM	[3] [2] [4] [6] [10] [9] [8] [7]	$\lvert\Sigma\rvert \cdot 2^{\lvert \mathit{pattern} \rvert}$
Partsch-Stomp	[3] [2] [4] [6] [10] [9] [8] [7]	$\lvert\Sigma\rvert \cdot \lvert \mathit{pattern} \rvert$

The running example does not distinguish the Optimal BM algorithm from the Partsch-Stomp algorithm, but they are easily distinguished by the pattern `aa` and the text `baa`, say:

Name	Trace
Optimal BM	[1] [0] [2]
Partsch-Stomp	[1] [0] [2] [1]

```
datatype entry = {index, next}

i = -1; j = 0;
t[0] = new entry(index = -1);

while(j < |pattern|-1){
  head = t[i];
  while(i >= 0 && pattern[i]!= pattern[j]){
    i = head->index;
    head = head->next;
  }
  i=i+1; j=j+1;

  head = t[i];
  chain = t[j] = new entry(index = i);

  k = i;
  while(k!=-1){
    if(pattern[k]==pattern[j])
      chain->index = head->index;
    else {
      chain->next = new entry(index = head->index);
      chain = chain->next;
    }
    k = head->index;
    head = head->next;
  }
}

j = k = 0;
while(j < |pattern| && k < |text|){
  head = t[j];
  while(j >= 0 && text[k]!= pattern[j]){
    j = head->index;
    head = head->next;
  }
  k=k+1; j=j+1;
}

if(j==|pattern|) return k-j; else return -1;
```

Figure 7: Pseudo-code for the (noname#1) matcher

3.2.2 Composite matchers

The composite matchers form a potentially vast group, which turns out to be focused on advanced methods of backtracking.

The Boyer-Moore group Boyer and Moore pioneered a sophisticated yet very practical matcher that not only was linear in the worst case, but actually sub-linear in the best case. It consists of a *bad-character-shift* and a *good-suffix* heuristic, whose proper cooperation is captured by the SKEW combinator:

	SKEW
Boyer-Moore	— BASIC(*right-to-left*, *char*, $+$*all*, $-$*all*)
	— BASIC(*last-only*, *table*, $+$*all*, $-$*all*)
	SKEW
Original BM	— BASIC(*right-to-left*, *char*, $+$*pos*, $-$*all*)
	— BASIC(*last-only*, *table*, $+$*all*, $-$*all*)

The heuristics are often considered on their own. For very small alphabets the *good-suffix* heuristic seems to dominate the performance of the Boyer-Moore algorithm, while in practice, for larger alphabets such as ASCII, Horspool showed that the *bad-character-shift* heuristic dominates the performance [28]. Note that for the example the effective alphabet size is only three, and so it is not surprising that the Boyer-Moore algorithm and the Good Suffix algorithm here behave similarly.

Name	Trace	Cache bound
Boyer-Moore	3 2 [2] 4 [4] 6 [6] 10 9 8 7	$\|\Sigma\| + \|pattern\|$
Original BM	3 2 [2] 4 [4] 6 [6] 10 9 8 7	$\|\Sigma\| + \|pattern\|$

Again, the running example does not distinguish the two versions; consider instead the pattern `cababa` and the text `xxxxaababacababa` [9]:

Name	Trace
Boyer-Moore	5 4 [4] 9 8 7 6 5 4 [4] 15 14 13 12 11 10
Original BM	5 4 [4] 7 6 5 4 [4] 9 8 7 6 5 4 [4] 15 14 13 12 11 10

The Boyer-Moore/*bad-character-shift* heuristic group The pure *bad-character-shift* heuristic is illustrated by Horspool's algorithm (see Figure 3.1.2). Only very recently have this heuristic been obtained by partial evaluation, using non-trivial binding-time improvements [15]. This heuristic is captured by the use of a *table* comparison at the last position. The group comprises three parallel blood lines depending on which character positions are used [28, 40, 43]:

Horspool:	Uses the last character position of the pattern.
Sunday:	Uses the character position just past the pattern.
Smith:	Uses the best of the above.

Since all matchers rely solely on the *bad-character-shift* heuristic to skip positions, the matchers within each blood line differ only in the order in which they perform character comparisons. Although Horspool, Sunday, and Smith's algorithms are really *any-order*, we here use the orders of the particular versions they each presented.

Note that being any-order implies that each algorithm is dual to (a variant of) itself. The Raita algorithm [37] is an instance of the Horspool algorithm. We have:

	BACKTRACKING
Horspool	— BASIC(*last-left-to-right*, *char*, +*none*, −*none*)
	— BASIC(*last-only*, *table*, +*all*, −*all*)
	BACKTRACKING
Raita	— BASIC(*last-first-middle-rest*, *char*, +*none*, −*none*)
	— BASIC(*last-only*, *table*, +*all*, −*all*)
	ALTERNATE
Quick Search	— BASIC(*left-to-right*, *char*, +*none*, −*none*)
	— BASIC(*last-only*, *table*, +*all*, −*all*)
	BACKTRACKING
	— BASIC(*left-to-right*, *char*, +*none*, −*none*)
	— PARALLEL
Smith	— BASIC(*last-only*, *table*, +*all*, −*all*)
	— ALTERNATE
	— FAIL
	— BASIC(*last-only*, *table*, +*all*, −*all*)

Since the *bad-character-shift* heuristic is very sensitive to individual characters in the text, the matchers do not compare as well as the Knuth-Morris-Pratt group.

Name	Trace	Cache bound
Horspool	3 0 1 2 [3] 4 [4] 6 [6] 10 7 8 9	$\lvert\Sigma\rvert$
Raita	3 0 2 [3] 4 [4] 6 [6] 10 7 9 8	$\lvert\Sigma\rvert$
Quick-search	0 1 2 [4] 3 4 5 6 [7] 4 [8] 7 8 9 10	$\lvert\Sigma\rvert$
Smith	0 1 2 [3] [4] 3 4 5 6 [6] [7] 7 8 9 10	$2 \cdot \lvert\Sigma\rvert$

The Not-So-Naive matcher As a footnote, it is interesting to notice the Not-So-Naive matcher, which is a constant-space Brute-Force-refinement that can even be slightly sub-linear [10]. It simply remembers the outcome of the second comparison, which in turn allows it to sometimes skip a position. Its specification is

	SEQUENTIAL
Not-So-Naive	— BASIC(*second-only, char, +all, −all)*
	— BASIC(*left-to-right-skip-second, char, +none, −none)*

and its trace on the example is only `1 2 3 4 5 6` $\underline{6}$ `7 8 9 10` $\underline{7}$. This matcher induces a flora of constant-space variants that remember any finite number of comparison outcomes; one might be considered a "Poor man's Good Suffix":

	SEQUENTIAL
(noname#2)	— BASIC(*last-only, char, +all, −all)*
	— BASIC(*right-to-left-skip-last, char, +none, −none)*

Its trace on the example is `3 2 4 6 8 10 9 8 7`.

3.3 Summary

In this section, we have developed a framework (that is, a specification language and interpretation) for characteristic information propagation. Verified using the measure of Section 2, the framework provides concise specifications for a wide variety of behaviors. It builds on existing frameworks and as such enjoys the same benefits [6, 7, 36] (see also Section 4.2). The technical novelties of our framework are: (1) the flexible concept of a partial matcher; (2) the use of multiple (and non-aligned) caches; and (3) to distinguish between *char* and *table* comparisons.

By measuring *real* string matchers [10], we further obtain concise specifications of these, many of which have never been obtained before. A key point is that we can move freely between the "information propagation"-oriented specification and its behaviorally-equivalent efficient implementation. This connection has allowed us to present an information-propagation-based taxonomy of string matchers. Notably, we have obtained the practically efficient Boyer-Moore/*bad-character-shift* heuristic in a systematic and reusable way.

Measuring real string matchers also allows us to introduce sound canonical naming of specifications and behaviors, and furthermore gives us plenty of data points for controlling exploration.

4 An analysis of information propagation

In the program-generation community, the Knuth-Morris-Pratt algorithm in particular has been reconstructed many times, since Futamura in 1987 proposed it as a challenging problem for partial evaluators [1,5,6,8,12,16,17,19,21,22,25,27,33,36,39, 41, 44]. From an information-propagation perspective, often either the Morris-Pratt algorithm, the (noname#1) algorithm, or the Automaton has been reconstructed instead, but as a catalyst for advancements in partial evaluation such imprecision is harmless. The imprecision does, however, prohibit direct comparison of methods and approaches.

Only lately did Ager, Danvy, and the author prove formally what exactly gives rise to the Knuth-Morris-Pratt algorithm [2].

4.1 Advanced program transformations

We now analyze characteristic information propagation of advanced program transformations. Foremost, we phrase the KMP-test in terms of information propagation (behavior) rather than efficiency (time complexity):

$$\text{KMP}(T) \Leftrightarrow \exists M.\, \|M\| = \|\text{Brute-Force}\| \wedge \|T(M)\| = \|\text{Knuth-Morris-Pratt}\|$$

Similarly, we have a Morris-Pratt-test and a (noname#1)-test. The use of *characteristic* information is here convenient, since we thereby factor out representation issues (such as propagating $p \neq nil$ vs. $m \neq 0$) and how information is propagated (substitution vs. environments vs. predicates).

We restrict the analysis to transformations that pass the original KMP-test.

4.1.1 Positive supercompilation

Positive supercompilation is a variant of supercompilation that propagates positive information only and uses *folding* without generalization [41]. The benefits of the simplification are that the propagation of information can be defined as term substitution due to the absence of negative information and that it is still powerful enough to pass the original KMP-test.

As for information propagation, positive supercompilation passes the Morris-Pratt-test.

4.1.2 S-Graph supercompilation

Another variant of supercompilation, due to Glück and Klimov, propagates both positive and negative information (so-called restrictions) using environments [22]. For passing the original KMP-test, a simple folding strategy similar to the above suffices, where only identical configurations are folded.

As for information propagation, S-Graph supercompilation passes the (noname#1)-test.

4.1.3 Turchin's supercompilation

Supercompilation, as described by Turchin [45], is a certain type of program transformation that works by first observing, analyzing, and modeling the *execution* of a source program and then returning an equivalent, optimized program. Phrased in the functional language Refal, it has been reported to pass the original KMP-test [23]. An implementation of both Refal and the supercompiler scp4 are available [38].

In this context, a problematic feature of Refal is its very general pattern matching capabilities (a comparison must be defined as pattern matching in Refal), because the pattern matching algorithm is non-deterministic and does not readily give rise to a well-defined trace. Under the assumption that pattern matching is non-redundant (see Section 4.3.2) and by experimenting with the implementation, we discovered that scp4 passes either the Morris-Pratt-test or the (noname#1)-test, depending on how comparisons are encoded in the source program.

In Refal, strings are sequences of symbols and the pattern matching encoding of character comparisons really encodes two "standard" comparisons: "is the text string non-empty?" and, if so, "is the first symbol equal to the appropriate one in the pattern?". Apparently, negative information is not propagated (in the sense of exploited) if the two "else" branches of the encoded character comparison are merged, that is, if $(t = A :: t' \wedge A \neq X)$ is in disjunction with the non-characteristic $(t = nil)$.

4.1.4 Generalized partial computation

Generalized Partial Computation is a program transformation *method* that utilizes partial evaluation, theorem proving, and the propagation of predicates [19]. As such, GPC is capable of generating the whole Knuth-Morris-Pratt-family, if the right fold/unfold choices are made.

Here, its (semi-)automatic instances are more interesting [18, 20]. We consider a tail-recursive Brute-Force matcher [20, page 73]; by hand, we have followed the semi-automatic GPC instance on the "tricky" pattern `abaa` and obtained the (noname#1) matcher. Hence, as for information propagation, GPC passes the (noname#1)-test.

4.2 String-matching frameworks

Existing string-matching frameworks have hitherto been difficult to distinguish from each other, because they have all generated both Knuth-Morris-Pratt-like and Boyer-Moore-like matchers and all verified the duality folklore. Accounting for propagated information, however, changes the situation.

4.2.1 The "duality" folklore revisited

We first re-investigate the "duality" folklore, accounting for information propagation. We list valid dualities containing every variant:

left-to-right	Specification	*right-to-left*
Brute-Force	Basic(-, *char*, +*none*, −*none*)	(noname#3)
(noname#1)	Basic(-, *char*, +*all*, −*none*)	(noname#4)
Knuth-Morris-Pratt	Basic(-, *char*, +*all*, −*neg*)	(noname#5)
Morris-Pratt	Basic(-, *char*, +*pos*, −*none*)	(noname#6)
(noname#7)	Basic(-, *char*, +*all*, −*all*)	Good Suffix
(noname#8)	Basic(-, *char*, +*pos*, −*all*)	Original GS
Automaton	Basic(-, *table*, +*all*, −*none*)	Optimal BM
(noname#9)	Basic(-, *table*, +*all*, −*all*)	Partsch-Stomp
(noname#10)	Skew — Basic(-, *char*, +*all*, −*all*) — Basic(*last-only*, *table*, +*all*, −*all*)	Boyer-Moore
(noname#11)	Skew — Basic(-, *char*, +*pos*, −*all*) — Basic(*last-only*, *table*, +*all*, −*all*)	Original BM

Note that we still need to exhibit a counter-example to show that certain matchers are *not* dual (since specifications are not necessarily unique); a trivial comparison of the traces of the running example reveals that there is no precise duality connecting the Knuth-Morris-Pratt group with either of the Boyer-Moore variants. We must thus reject the duality conjecture when accounting for information propagation.

However, some of the underlying intuition does hold:

1. We do have a precise duality between the Automaton and the Optimal BM algorithm; this duality is thus the only known precise duality between algorithms from the literature. These matchers are however knowledge-wise simpler than the Knuth-Morris-Pratt and Boyer-Moore matchers.

2. We observe that if we enforce that *right-to-left* matchers always prune everything, the following weak dualities emerge:

left-to-right	Specification	*right-to-left*; −*all*
(noname#1)	Basic(-, *char*, +*all*, (−*none*))	Good Suffix
Knuth-Morris-Pratt	Basic(-, *char*, +*all*, (−*neg*))	Good Suffix
Morris-Pratt	Basic(-, *char*, +*pos*, (−*none*))	Original GS
Automaton	Basic(-, *table*, +*all*, (−*none*))	Partsch-Stomp

The Knuth-Morris-Pratt group is weakly dual to the Boyer-Moore/*good-suffix* heuristic group (where the repetition of the Good Suffix is an artifact of our choice of pruning primitives).

However, the implied promise of the conjecture, namely that by duality one could get two efficient matchers for the price of one, seems doubtful.

4.2.2 Amtoft et al.'s framework

Our closest related framework is by Amtoft, Consel, Danvy, and Malmkjær [7, 14]. The framework explicitly represents text knowledge in a single cache, which may contain both positive and negative information; such knowledge is acquired by character comparisons. Upon a mismatch, negative information is always acquired and kept until a match occurs; then entries of the cache may be cleared according to one of several strategies (e.g., clear all negative entries). However, clearing the cache also after each mismatch is mentioned as a possible improvement. Their work also includes a correctness proof (based on equational reasoning) ensuring that they only generate *correct* string matchers.

The main aim of their work is to show that Knuth-Morris-Pratt-like and Boyer-Moore-like matchers can be obtained from a common matcher by varying the traversal order. They mention the following *left-to-right* matchers

Morris-Pratt	BASIC(*left-to-right*, *char*, +*pos*, −*none*)
Knuth-Morris-Pratt	BASIC(*left-to-right*, *char*, +*all*, −*neg*)
(noname#1)	BASIC(*left-to-right*, *char*, +*all*, −*none*)

and the following *right-to-left* matchers

	BACKTRACKING
Horspool	— BASIC(*last-left-to-right*, *char*, +*none*, −*none*)
	— BASIC(*last-only*, *table*, +*all*, −*all*)
	SKEW
Boyer-Moore	— BASIC(*right-to-left*, *char*, +*all*, −*all*)
	— BASIC(*last-only*, *table*, +*all*, −*all*)
Partsch-Stomp	BASIC(*right-to-left*, *table*, +*all*, −*all*)
Optimal BM	BASIC(*right-to-left*, *table*, +*all*, −*none*)

They informally conjecture that the behavior of these matchers appears to be obtained, with the exception of the Boyer-Moore, which "exploits the two tables in a rather unsystematic way" [7, page 347].

Under our definition of behavior, the conjecture does not fully hold. This is mainly because: (1) always acquiring and keeping negative information between matches turns out to be too inflexible to obtain any matcher from the literature, since no such matcher keeps more than one character of negative information; and (2) our definition insists that table lookups must not be "simulated" by character comparisons in the same sense that the recognition process of the Automaton is "simulated" by the Morris-Pratt. They thus obtain groups of Knuth-Morris-Pratt-like and Good Suffix-like matchers with additional uses of negative information (not all expressible in our framework as is). The precise duality present in their work is

(noname#1)	BASIC(-, *char*, $+$*all*, $-$*none*)	(noname#4)

along with similar but inexpressible dualities. Their improved framework, in contrast, would be equivalent to the BASIC(-, *char*, -, -) subset of ours (modulo the formulation of pruning primitives).

4.2.3 Amtoft's framework

Amtoft's PhD thesis [6, Chapter 7] contains an earlier framework. In the setting of multi-level transition systems, string matchers are specified as transition rules that explicitly represent text knowledge in a single cache. The cache may contain both positive and negative information; such knowledge is acquired by character comparisons (i.e., rules where the triggering conditions are paired such that each state has only two possible successor states dependent on a single character comparison). The partial evaluator is an abstract machine that has the ability to generate rules where each state may have any number of successor states.

The main aim is also here to show that that Knuth-Morris-Pratt and Boyer-Moore algorithms can be obtained from a common parameterized matcher and to formalize that the Knuth-Morris-Pratt algorithm is dual to the Optimal BM algorithm. The various matchers are obtained through different traversal orders (called search strategies), which may at any point contain an operation that clears the cache. The following matchers are mentioned:

Knuth-Morris-Pratt	BASIC(*left-to-right*, *char*, $+$*all*, $-$*neg*)
	BACKTRACKING
Horspool	— BASIC(*last-left-to-right*, *char*, $+$*none*, $-$*none*)
	— BASIC(*last-only*, *table*, $+$*all*, $-$*all*)
	SKEW
Boyer-Moore	— BASIC(*right-to-left*, *char*, $+$*all*, $-$*all*)
	— BASIC(*last-only*, *table*, $+$*all*, $-$*all*)
Partsch-Stomp	BASIC(*right-to-left*, *table*, $+$*all*, $-$*all*)
Optimal BM	BASIC(*right-to-left*, *table*, $+$*all*, $-$*none*)

– where the latter four are called the "naive", "original", "standard", and "optimal" Boyer-Moore algorithm, respectively. Amtoft does not attempt to obtain the "naive" and "original" versions of the Boyer-Moore algorithm; they are considered to "exploit their information in a rather unsystematic way" [6, page 175]. He conjectures that the remaining three matchers are obtained and that the aim is accomplished.

Under our definition of behavior, the conjecture does not quite hold. This is mainly because: (1) the Automaton from Aho, Hopcroft, and Ullman's classic book [4, Algorithm 9.3] is not distinguished from the Knuth-Morris-Pratt algorithm [6, page 161]; and (2) the trace depends on whether an implementation of the framework evaluates the conditions of inapplicable rules or not.

Assuming that inapplicable conditions are never evaluated, the following matchers appear to be obtained:

Automaton	BASIC(*left-to-right*, *table*, $+$*all*, $-$*none*)
Partsch-Stomp	BASIC(*right-to-left*, *table*, $+$*all*, $-$*all*)
Optimal BM	BASIC(*right-to-left*, *table*, $+$*all*, $-$*none*)

The precise duality found is accordingly

Automaton	BASIC(-, *table*, $+$*all*, $-$*none*)	Optimal BM

Amtoft thus appears to be the first to obtain the Automaton group and the first to find the only known duality between algorithms from the literature.

4.2.4 Queinnec and Geffroy's framework

The first published framework is due to Queinnec and Geffroy [36]. The framework explicitly represents text knowledge in a single cache (called a description), which may contain both positive and negative information; such knowledge is acquired by character comparisons and never forgotten (clearing the cache is however mentioned as future work). Beyond more advanced examples using various combinators, they mention two matchers:

Knuth-Morris-Pratt	BASIC(*left-to-right*, *char*, $+$*all*, $-$*neg*)
	SKEW
Boyer-Moore	— BASIC(*right-to-left*, *char*, $+$*all*, $-$*all*)
	— BASIC(*last-only*, *table*, $+$*all*, $-$*all*)

They conjecture that these matchers are instances of the framework, dependent only on the traversal order.

Under our definition of behavior, the conjecture does not hold. The reasons are the same as for Amtoft et al.'s framework. Additionally, the Boyer-Moore's use of disjoint information seems to be difficult – if not impossible – to capture with a single cache. The matchers obtained and the precise duality thus seems to be

(noname#1)	BASIC(-, *char*, $+$*all*, $-$*none*)	(noname#4)

As a footnote, our analysis also explains why – in contrast to the Boyer-Moore algorithm – Queinnec and Geffroy for the (noname#4) matcher report that "the generated code tends to be unpredictably voluminous". The reason being that the space usage for the (noname#4) varies greatly, whereas the usual implementation of the Boyer-Moore algorithm always uses $|\Sigma| + |pattern|$ memory cells.

4.3 Assessment

Due to the standardization of the Knuth-Morris-Pratt example, our analysis sheds new light on existing work. Notably, it allows us to accurately quantify differences in information propagation of transformations and frameworks that were previously considered to be equivalent. The analysis has two corollaries.

4.3.1 Self-application and efficient preprocessing

The preprocessing phase of string matching algorithms is often quite efficient; in contrast, the transformation required to specialize a naive matcher is usually not geared towards efficiency [1].

In the context of GPC, in particular, self-application has been mentioned as a way to eliminate the overhead introduced by the transformation itself (here, mainly theorem proving) [16, 23]. This idea has been conjectured to give rise to linear-time preprocessing, thus obtaining the full Knuth-Morris-Pratt algorithm. However, by the analysis, if GPC really generates the (noname#1) matcher instead, linear-time preprocessing is perhaps impossible without laziness, since the (noname#1) matcher may not have linear size [24]. (If it does have linear size as conjectured [2], then proving this non-trivial conjecture must be done first.) This example suggests that, for self-application at least, tight control with propagated information is likely to be important.

Recently, however, the propagation of information has been restricted by certain special conditional expressions and been used to obtain Boyer-Moore-like and Knuth-Morris-Pratt-like matchers [16], in the traversal-order/search-phase-time-complexity sense. Although such special expressions allow GPC to obtain more string matchers, they are no longer obtained fully automatically by transforming naive matchers.

4.3.2 Influence of underlying formalism

In some cases, the reported results have been influenced by the choice of underlying formalism.

In particular, reported results tend to exclusively use either *char* or *table* comparisons (e.g., unrestricted transition systems can *a priori* only be interpreted as the latter). The treatments of Boyer-Moore-like string matchers have suffered most, since this family of string matchers gain their practical worth from skillfully combining these two types of comparisons.

The advanced pattern-matching capabilities of Refal make it a difficult language to analyse, since concept of "the sequence of character comparisons" – as well as the non-redundancy assumption mentioned in Section 4.1.3 – is essentially implementation-defined. The assumption simply means that we interpret, say,

```
F1 {  'ab' e   = <F2 e> ;
      'a' e    = <F3 e> ;
      e        = <F4 e> ; }
```

as equivalent to

```
F1 {  'a' e    = <F5 e> ;         F5 {  'b' e    = <F2 e> ;
      e        = <F4 e> ; }             e        = <F3 e> ; }
```

Although in this case it seems like a decent assumption, it is far less clear what the behavior of the following valid string matcher, for `abaa`, say, is:

```
F1 {  e1 'abaa' e2 = <sizeof e1> ; }
```

Furthermore, even under the non-redundancy assumption, the reported program from Glück and Turchin's work [23] behaves like a hybrid between the Morris-Pratt and Knuth-Morris-Pratt algorithms (not expressible in our framework as is).

5 Conclusion and perspectives

We have presented an investigation of information propagation in partial evaluation.

The core contribution is a practical formalism-independent measure called *behavior* – namely text character accesses – of the propagation of so-called *characteristic* information of string matchers. In this context, we have accurately measured the amount of non-trivial information propagated by program transformations and revealed the standard Knuth-Morris-Pratt example as several distinct examples with respect to information propagation.

Furthermore, we have developed a framework (that is, a specification language and interpretation) for characteristic-information propagation. The framework provides concise specifications for a wide variety of behaviors. We obtain concise specifications of string matchers from the literature [10], many of which have never been obtained before. Not only does this connection introduce sound canonical naming of specifications and behaviors, but a key point is that we can move freely between the "information propagation"-oriented specification and its behaviorally-equivalent efficient implementation. Notably, we have obtained the Boyer-Moore/*bad-character-shift* heuristic in a systematic and reuseable way.

This work sheds new light on the information-propagation aspect of a variety of existing work. For example can we see that the intricate Boyer-Moore algorithm makes good sense from an information-propagation perspective. Although existing frameworks and transformation techniques in general maintain too much negative information, the situation is not clear cut, as the Good Suffix vs. Original GS matchers show. Still, in the case of self-application, tight control with propagated information is likely to be essential.

This work shows that string matching is still a rich catalyst for experiments in program transformation. A practical application of the present work may be in the development of program transformations, where the *behavior* measure can be used mechanically to keep information propagation under control. Furthermore, the framework can be used for controlled exploration.

The large gap between good theoretical properties (notably the Knuth-Morris-Pratt algorithm) and practical worth (notably the Horspool algorithm) suggests that aiming for the generation of a Boyer-Moore/ *bad-character-shift* variant from some naive matcher would be a promising achievement (i.e., a BM-test). More ambitiously, it seems worthwhile to investigate how well opportunistic applications of bounded static variation – as needed to pass the BM-test [15] – scales to other settings.

Acknowledgments A special thanks to Olivier Danvy for introducing me to the topic; for encouragement; and for countless useful comments and discussions. I am also grateful to Mads Sig Ager, Torben Amtoft, and Neil Jones for insightful comments to preliminary versions of this article.

References

[1] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. Fast Partial Evaluation of Pattern Matching in Strings. *ACM Transactions on Program-*

ming Languages and Systems. Available as technical report BRICS-RS-04-40. To appear.

[2] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. On Obtaining Knuth, Morris, and Pratt's String Matcher by Partial Evaluation. In Chin [11], pages 32–46. Extended version available as the technical report BRICS-RS-02-32.

[3] Alfred V. Aho. Algorithms for finding patterns in strings. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 5, pages 255–300. The MIT Press, 1990.

[4] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[5] Maria Alpuente, Moreno Falaschi, Pascual Juliàn, and German Vidal. Specialization of inductively sequential functional logic programs. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 151–162, Amsterdam, The Netherlands, June 1997. ACM Press.

[6] Torben Amtoft. *Sharing of Computations.* PhD thesis, University of Aarhus, Denmark, 1993. Technical report PB-453.

[7] Torben Amtoft, Charles Consel, Olivier Danvy, and Karoline Malmkjær. The abstraction and instantiation of string-matching programs. In Mogensen et al. [34], pages 332–357.

[8] Anders Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, Computer Science Department, University of Copenhagen, 1990. DIKU Rapport 90/17.

[9] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[10] Christian Charras and Thierry Lecroq. Exact string matching algorithms. `http://www-igm.univ-mlv.fr/~lecroq/string/`, 1997.

[11] Wei-Ngan Chin, editor. *Proceedings of the ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* ACM Press, September 2002.

[12] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, January 1989.

[13] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[14] Charles Consel, Olivier Danvy, and Karoline Malmkjær. The abstraction and instantiation of string-matching programs. Unpublished manuscript, February 1990, and talks given at Stanford University, Indiana University, Kansas State University, Northeastern University, Harvard, Yale University, and INRIA Rocquencourt.

[15] Olivier Danvy and Henning Korsholm Rohde. On obtaining the Boyer-Moore string-matching algorithm by partial evaluation. *Information Processing Letters*. Available as technical report BRICS-RS-05-14. To appear.

[16] Yoshihiko Futamura, Zenjiro Konishi, and Robert Glück. Automatic generation of efficient string matching algorithms by generalized partial computation. In Chin [11], pages 1–8.

[17] Yoshihiko Futamura, Zenjiro Konishi, and Robert Glück. Program transformation system based on generalized partial computation. *New Generation Computing*, 20(1):75–99, 2002.

[18] Yoshihiko Futamura, Zenjiro Konishi, and Robert Glück. WSDFU: Program transformation system based on generalized partial computation. In Mogensen et al. [34], pages 358–378.

[19] Yoshihiko Futamura and Kenroku Nogi. Generalized partial computation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, 1988.

[20] Yoshihiko Futamura, Kenroku Nogi, and Akihiko Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61–79, 1991.

[21] Robert Glück and Jesper Jørgensen. Generating optimizing specializers. In Henri Bal, editor, *Proceedings of the Fifth IEEE International Conference on Computer Languages*, pages 183–194, Toulouse, France, May 1994. IEEE Computer Society Press.

[22] Robert Glück and Andrei Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In Patrick Cousot, Moreno Falaschi, Gilberto Filé, and Antoine Rauzy, editors, *Proceedings of the Third International Workshop on Static Analysis WSA'93*, number 724 in Lecture Notes in Computer Science, pages 112–123, Padova, Italy, September 1993. Springer-Verlag.

[23] Robert Glück and Valentin F. Turchin. Application of metasystem transition to function inversion and transformation. In *Proceedings of the international symposium on symbolic and algebraic computation*, pages 286–287, Tokyo, Japan, August 1990. ACM, ACM Press.

[24] Bernd Grobauer and Julia L. Lawall. Partial evaluation of pattern matching in strings, revisited. *Nordic Journal of Computing*, 8(4):437–462, 2002.

[25] Manuel Hernández and David A. Rosenblueth. Development reuse and the logic program derivation of two string-matching algorithms. In Harald Søndergaard, editor, *Proceedings of the Third International Conference on Principles*

and Practice of Declarative Programming, pages 38–48, Firenze, Italy, September 2001. ACM Press.

[26] Manuel Hernández and David A. Rosenblueth. Disjunctive partial deduction of a right-to-left string-matching algorithm. *Information Processing Letters*, 87:235–241, 2003.

[27] Carsten K. Holst and Carsten K. Gomard. Partial evaluation is fuller laziness. In Hudak and Jones [29], pages 223–233.

[28] R. Nigel Horspool. Practical fast searching in strings. *Software—Practice and Experience*, 10(6):501–506, 1980.

[29] Paul Hudak and Neil D. Jones, editors. *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, New Haven, Connecticut, June 1991. ACM Press.

[30] Andrew Hume and Daniel Sunday. Fast string searching. *Software—Practice and Experience*, 21(11):1221–1248, 1991.

[31] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, London, UK, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/`.

[32] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[33] Laura Lafave and John P. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In Norbert E. Fuchs, editor, *7th International Workshop on Program Synthesis and Transformation*, number 1463 in Lecture Notes in Computer Science, pages 168–188, Leuven, Belgium, July 1997. Springer-Verlag.

[34] Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough, editors. *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, number 2566 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[35] Helmuth Partsch and Frank A. Stomp. A fast pattern matching algorithm derived by transformational and assertional reasoning. *Formal Aspects of Computing*, 2(2):109–122, 1990.

[36] Christian Queinnec and Jean-Marie Geffroy. Partial evaluation applied to pattern matching with intelligent backtrack. In *Proceedings of the Second International Workshop on Static Analysis WSA'92*, volume 81-82 of *Bigre Journal*, pages 109–117, Bordeaux, France, September 1992. IRISA, Rennes, France.

[37] Timo Raita. Tuning the Boyer-Moore-Horspool string searching algorithm. *Software—Practice and Experience*, 22(10):879–884, 1992.

[38] Refal-5 homepage. http://www.refal.net/.

[39] Donald A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In Hudak and Jones [29], pages 62–71.

[40] P. D. Smith. Experiments with a very fast substring search algorithm. *Software—Practice and Experience*, 21(10):1065–1074, 1991.

[41] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[42] Graham A. Stephen. *String Searching Algorithms.* World Scientific, 1994.

[43] Daniel M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, August 1990.

[44] Masato Takeichi and Yoji Akama. Deriving a functional Knuth-Morris-Pratt algorithm. *Journal of Information Processing*, 13(4):522–528, 1990.

[45] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[46] Bruce Watson and G. Zwaan. A taxonomy of keyword pattern matching algorithms. Technical report, Eindhoven University of Technology, The Nederlands, December 1992. Computer Science Note 92/27.

A Exact string matching algorithms

We provide a brief introduction to string matching focusing on the Knuth-Morris-Pratt and Boyer-Moore algorithms and on the properties usually outlined. The presentation follows standard material [3, 10, 42].

Let a string s be a finite sequence of atomic characters $s[i]$, where $i \in \{0, 1, ..., |s|-1\}$ and $|s|$ denotes the length of s. String matching algorithms here solve the following problem: given a pattern string *pattern* and a text string *text*, return the least κ such that for all $i \in \{0, 1, ..., |pattern| - 1\}$, $pattern[i] = text[\kappa + i]$, or -1 if no such κ exists. We use the standard abbreviations $m = |pattern|$ and $n = |text|$. For convenience are program fragments always in the scope of *pattern* and *text*.

The simplest algorithm is the Brute-Force algorithm, which performs an exhaustive search for m:

```
i=k=0;
while(i<m && k<n)       /* κ = k-i */
    if(pattern[i]==text[k]){
         k=k+1; i=i+1;
    } else {
         k=k-i+1; i=0;
    }
if(i==m) return k-i; else return -1;
```

It will serve to illustrate the *sliding window* mechanism [10], which is a rationalization over how most string matchers are structured (exceptions include e.g., position trees [4]).

The main loop (Lines 2-7) is the bounded search for applicable κ (with $\kappa = k - i$), where the invariant is that all $\kappa' < \kappa$ does not satisfy the matching criterion. For each κ, an *attempt* is made to verify the criterion; if successful the search is complete and the current k is the final result (Line 8); if not (Lines 5-6), the search is continued. If the search is exhausted, no κ satisfying the criterion exists (Line 8).

The Brute-Force algorithm runs in $O(mn)$ but uses no auxiliary space.

The Morris-Pratt algorithm improves on the Brute-Force algorithm after an analysis of its inefficiency. After a failed attempt, at i, say, we have already verified that for all $i' \in \{0, .., i\}$, $pattern[i'] = text[k + i']$. This verfication means that the subsequent *backtracking* – namely the comparisons up to accessing $text[k + i]$ again – are essentially comparing the pattern against itself; precalculation leads to an $\Theta(m + n)$ algorithm performing at most $2n - 1$ text character comparisons. It uses a m-sized "failure" table, which is cleverly constructed in $\Theta(m)$ using the same idea:

```
 1   i=0;
 2   j=f[0]=-1;
 3   while(i<m-1){
 4       while(j>-1 && pattern[i]!=pattern[j])
 5           j=f[j];
 6       i=i+1;
 7       j=j+1;
 8       f[i]=j;
 9   }
```

Backtracking is now replaced by a lookup:

```
10   i=k=0;
11   while(i<m && k<n){     /* κ = k-i */
12       while(i>-1 && pattern[i]!=text[k])
13           i=f[i];
14       i=i+1;
15       k=k+1;
16   }
17   if(i==m) return k-i; else return -1;
```

Note that k is now never decreased. The *delay* – which is the maximum number of text character comparisons before k increases – is bounded by m.

The Knuth-Morris-Pratt algorithm [32] improves on the Morris-Pratt algorithm by additionally exploiting the *negative* information after a mismatch, namely that $pattern[i] \neq text[k + i]$. Simply expanding Line 8 above to

```
 8.1   if(pattern[i]==pattern[j])
 8.2         f[i]=f[j];
 8.3   else
 8.4         f[i]=j;
```

incorporates this change. The improved failure table is called the "next" table; the asymptotic time and space usages are unchanged. A property that does distinguish these algorithms is the delay, which improves from m to $log_{\Phi}(m)$, where Φ is the golden ratio ($\Phi = \frac{1+\sqrt{5}}{2}$).

The Boyer-Moore algorithm [9] pioneered the idea of of trying to verify the matching criterion from right to left. In contrast to the above left-to-right algorithms, harnessing this idea allows it to sometimes avoid accessing every text character making it sub-linear at best. It uses two heuristics to gain efficiency: the *good suffix* heuristic (which is analogous to the "next" table) and the *bad-character-shift* heuristic (which aligns the text character that caused the mismatch with its right-most occurrence in the pattern). Note that the latter heuristic relies on the alphabet Σ being known (usually Σ = ASCII).

Efficient precalculation of the good-suffix heuristic is non-trivial [32]:

```
for(i=0; i<m; i=i+1)
     gs[i]=2*m-i-1;
j=m-1;
k=m;
while(j>-1){
    f[j]=k;
    while(k<m && pattern[j]!=pattern[k]){
         gs[k]=min(gs[k],m-j-1);
         k=f[k];
    }
    j=j-1; k=k-1;
}
for(i=0; i<k+1; i=i+1)
     gs[i]=min(gs[i],m+k-i);
j=f[k];
while(k<m){
     while(k<=j){
          gs[k]=min(gs[k],j-k+m);
          k=k+1;
     }
     j=f[j];
}
```

Assuming the alphabet is finite, precalculation of the bad-character-shift heuristic is usually done in $\Theta(|\Sigma|)$ time and space:

```
for(c∈ Σ)
     bcs[c]=m;
for(i=0;i<m-1;i=i+1)
     bcs[pattern[i]]=m-i-1;
```

The main loop is simple:

```
k=m-1;
while(k<n){
    i=m-1;
30        while(i>-1 && pattern[i]==text[k]){
31            k=k-1; i=i-1;
31        }
33        if(i==-1)
34            return k-i;
35        else
36            k=k+max(gs[i],bcs[text[i]]-m+i+1);
37    }
38    return -1;
```

After precalculations, the Boyer-Moore algorithm runs in $O(m+n)$ and is considered to be the one of the fastest algorithms in practice. It performs at most $6n$ text character comparisons [32].

The Boyer-Moore algorithm has inspired many variants: Horspool's simplification [28] is perhaps the best known – it consists essentially just of omitting the good-suffix heuristic. The resulting algorithm now runs in $O(mn)$, but performs comparably to the original in practice (see Stephen's overview [42, Section 2.1.4]). Horspool's algorithm is Lines 23-38 with Line 36 replaced by

```
36'           k=k+m-i+bcs[text[m-1]];
```

Note that `k` is now adjusted – not the `bcs[·]` value.

Similar variants based on the bad-character-shift heuristic exist, e.g., the Quick Search algorithm [43] (which instead uses the text character at index m for a modified bad-character-shift heuristic) and Smith's algorithm [40] (which uses the maximum of these two bad-character-shift heuristics).

Recent BRICS Report Series Publications