A Trace-based Framework for Comparing String Matchers*

Dan Amlund Thomsen,[1] 20040943

Master's Thesis, Computer Science

January 2012

Advisor: Olivier Danvy [†]

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

*Revisions based on feedback from Peter Sestoft, http://www.itu.dk/~sestoft/

[1]E-mail: dan@danamlund.dk

[†]http://users-cs.au.dk/danvy/

Abstract

Many different string matchers exist, but not so much exists on how these string matchers differ from each other. We present a trace-based framework that reveals these differences through various methods of comparing string matchers. These methods help us investigate, understand, and build string matchers.

We introduce three methods of comparing string matchers: (1) identifying a single string matcher by finding a known string matcher that is trace-equivalent with the single string matcher; (2) grouping a set of string matchers into trace-equivalent groups; and (3, our original contribution) building an evolutionary tree over string matchers. We present our results from applying these methods on traced versions of known string matchers, string matchers from published papers, and matchers generated using our framework by combining string-matching concepts in various ways.

Furthermore, we describe the work leading up to our framework. We define string-matching algorithms such as Knuth, Morris and Pratt's. We describe string-matching concepts and the specializing of string matchers as introduced by Futamura. We describe the composition of string matchers by combining string-matching concepts like Queinnec and Geffroy's string-matching framework. Finally, we describe how to compare string matchers by their traces on a set of input, as introduced in Rohde's string-matching framework.

Finally, from our experiences with working on string matching, we introduce a new approach to the understanding of string-matching algorithms. The traditional approach consists of understanding string-matching algorithms one by one. In contrast, our new approach focuses on understanding string-matching concepts and how to combine these basic concepts to compose string matchers. The new approach uses our methods to understand the similarities and differences of composed, specialized, and known string matchers.

Resumé

Vi præsenterer nye metoder til at undersøge, forstå og bygge string matchere med. Disse metoder stammer fra ideen at sammenligne string matchere ved at betragte deres trace. Vi præsenterer tre metoder: (1) en der identificerer matchere ved at finde hvilke andre matchere de er trace-equivalent med; (2) en der fordeler string matchere ind i grupper af hvilke matchere der er trace-equivalente; og (3, vores originale ide) en der giver et overblik over string matchere og viser hvordan de er relateret til hinanden ved at bygge evolutionære træer.

Dette speciale beskriver de ideer og koncepter der har inspireret det: Vi beskriver string-matching algoritmer med henblik på string-matching koncepter. For eksempel beskriver vi Knuth, Morris og Pratt's algoritme og dens brug af positiv og negativ information. Derefter beskriver vi specialisering af string matchere, dvs. hvordan man kan transformere string matchere, for eksempel fra en naive til en Knuth-Morris-Pratt matcher. Fra specialisering kom string-matching frameworks som lader os kombinere string-matching koncepter og derved opbygge string matchere. String-matching frameworks giver indblik i hvilken effekt forskellige koncepter har på string matchere bygget med dem. Vi beskriver hvordan Rohde's string-matching framework introducerede en ny metode til at sammenligne string matchere på: at sammenligne matchere ved at sammenligne deres traces på en mængde inputs.

Derefter beskriver vi vores trace-based framework. Vores framework bygger videre fra Rohde's. Vi beskriver hvordan man, i vores framework, definere og kombinere string-matching koncepter til at opbygge forskellige string matchere. Vi beskriver hvordan vi har optaget traces af string matchere implementeret i C og Scheme. Derefter forklarer vi hvordan vores metoder til at sammenligne string matcher på virker. Vi beskriver hvordan vi har implementeret metoderne, hvilke input vi bruger og hvordan man aflæser metodernes resultater.

Herefter beskriver vi vores resultater fra vores undersøgelse af string matchere med vores metoder. Vi viser en tabel med grupper af trace-equivalente string matchere, og beskriver hvad denne tabel fortæller os om sammenhængen mellem matchere opbygget ud fra koncepter og matchere der implementerer kendte algoritmer. Vi viser vores evolutionære træer og beskriver hvordan træerne grupperer string matchere baseret på hvilke kon-

cepter matcherne er opbygget af. Derefter bruger vi vores metode til at identificere string matchere med, til at vi verificere at matchere fra litteraturen implementere de algoritmer de påstår.

Til sidst konkludere vi vores opdagelser og præsenterer perspektiver omkring vores arbejde med string matchere. Ud fra vores perspektiver opstår ny fremgangsmåde til forståelse af string matchere der bygger på koncepter. Den traditionelle fremgangsmåde består i at forstå string-matching algoritmer en af gangen, hvorimod vores fremgangsmåde bygger på forståelse af koncepter og hvordan man kan kombinere disse til at bygge string matchere. Denne nye fremgangsmåde er gjort mulig igennem vores metoder, da de lader os forstå sammenhænge mellem kombinerede, specialiserede og kendte string matchere.

Acknowledgments

This thesis would not have been possible without the help and support from friends and family.

I would like to thank my thesis supervisor, Olivier Danvy. Through his courses and our work together on this dissertation, he has magnified my interest in programming languages, technical writing, and the English language.

I am also grateful to my colleagues Mustafa Sariyar, Martin Sergio Hedevang Fæster and Kim Munk Petersen for keeping me focused on working; to my family for helping me unfocus and relax too; to everyone at the department of computer science at Aarhus University for a friendly environment where questions and discussions are always welcome; and a special thanks to the sysadmins for keeping everything running smoothly.

Peter Sestoft served as external evaluator for this thesis. His comments prompted me to clarify the definition of trace equivalence and to include an overview over the results obtained by applying the trace-based framework to string matchers documented in the literature.

Dan Amlund Thomsen,
Aarhus, Tuesday 31st January, 2012.

Contents

List of Figures

Introduction

Our thesis is that trace-based frameworks make it possible to compare string matchers, and that this comparison reveals new methods of investigating, understanding and building string-matching algorithms. We defend this thesis by introducing a trace-based framework for comparing string matchers and by describing how we have used it to investigate string matchers. This dissertation describes the work leading up to our thesis, our trace-based framework and its results in five chapters.

1. We define topics related to our thesis. We summarize the background leading up to it and present our contributions supporting it.

2. We then introduce the topics related to our thesis. We describe relevant string-matching algorithms, and the string-matching concepts used by these algorithms. We describe what specialization of string matchers is and how this technique has been used to investigate matchers. Finally, we describe string-matching frameworks from the literature and how they have been used to compose string matchers.

3. We then introduce our trace-based framework. We present our implementation and describe how it composes string matchers by combining string-matching concepts. We also describe our frameworks three methods of comparing string matchers: Identifying a single matcher by comparing its traces on known matchers over a set of inputs; grouping trace-equivalent string matchers in a table; and building an evolutionary tree over string matchers.

4. We then present and examine the results of using our trace-based framework to investigate string matchers. We have used our framework to build an evolutionary tree over string matchers, which reveals how the matchers are related to each other. We have also used our framework string matchers from the literature, both specialized and fully-defined ones.

5. Finally, we conclude and present perspectives about our work. In this chapter we present a new approach to understanding string-matching algorithms.

The trace-based framework, figures presented and figures referenced in this dissertation are available online.[2]

[2] http://www.danamlund.dk/masters_thesis

Chapter 1

Background and Motivation

This chapter presents the work leading up to our thesis. We discuss string matchers: the naive and the sophisticated algorithms known from the literature. We review the use of partial evaluation to specialize a naive matcher into a known one. We describe a general method of defining string matchers, as well as the difficulty of comparing them. We discuss an automatic method that uses tracing to observationally distinguish whether two string matchers are different. Finally, we present the contributions of this thesis.

1.1 String-matching Algorithms

In essence, *string-matching algorithms* find the first occurrence of a string (the pattern) in another string (the text). We call an implementation of a string-matching algorithm a *string matcher*. The *naive string-matching algorithm* checks whether the pattern is a prefix of one of the successive suffixes of the text. Historically, the first well-known string-matching algorithm is due to Knuth, Morris and Pratt (*KMP*) [15]. Compared to the naive string-matching algorithm, the KMP algorithm uses information about earlier comparisons to avoid checking whether the pattern is a prefix of *each* of the successive suffixes.

The KMP algorithm compares characters from left to right as opposed to the Boyer and Moore's algorithm (*BM*) [5] which compares characters from right to left. Another difference between the KMP and the BM algorithms is their complexity: the average runtime for KMP is linear whereas for BM it is sub-linear.

String-matching algorithms can be compared using different methods such as looking at concepts used (left-to-right vs. right-to-left), complexity (average running time) and, for the purpose of this thesis, the *traces* of string matchers implementing the two string-matching algorithms:

Definition 1 (Trace [1, Definition 5]) *The* trace *of a string matcher is the sequence of comparisons performed when searching for a given pattern*

in a given text. Tracing *a string matcher means adding the ability to record traces of that matcher.*

Definition 2 (Trace equivalence [1, Theorem 1]) *String matchers are* trace equivalent *when they have the same trace for all input.*

Trace equivalence is a theoretical definition, but we compare traces in a practical setting. This means we can never fully show two matchers to be trace equivalent since we only compare traces over a finite set of inputs. We can, however, show two matchers to not be trace equivalent by finding one input resulting in different traces for two matchers.[1]

In order to compare string matchers by their traces we assume that two matchers are likely trace equivalent when they have the same traces over a large set of inputs. Our results show that this assumption is valid by presenting meaningful comparisons of string matchers.

1.2 Specialization of String Matchers

In essence, *specialization* is the process of processing and hopefully optimizing a program with respect to a part of its input. Running the specialized program on the rest of the input yields the same result as running the original program on the complete input. Specializing a string matcher with respect to a given pattern produces a specialized matcher taking only a text as input, and searching for the given pattern in any given text.

A canonical measure of specialization techniques is the *KMP test*. The test originally determined whether a technique could specialize a naive string matcher into a KMP matcher. But the KMP test has also been used to show the general case of specializing any quadratic-time naive matcher into a KMP-like matcher running in linear time. The KMP test was introduced by Futamura to show the power of generalized partial evaluation [11].

The KMP test has inspired further work on specialization of string matchers. Some have passed it by using the simpler polyvariant partial evaluation method to specialize a binding-time separated naive string matcher into a KMP matcher, as shown by Consel and Danvy [7]. Others have passed it using the new specialization method called positive supercompilation, which can specialize a naive string matcher into a MP matcher, as shown by Sørensen, Glück and Jones [24].

1.3 Abstraction of String Matchers

Following the work on specializing string matchers to ones implementing KMP, BM and others, Queinnec [18], Amtoft et al. [3, 4] and Rohde [21] have

[1] "Program testing can be used very efficiently to show the presence of bugs, but never to show their absence" - Dijkstra [10]

each designed string-matching frameworks for generating different string matchers. These matchers are generated by instantiating concepts such as traversal order, what text comparisons to cache, etc. The generated string matchers are then specialized into linear-time matchers, some of which implement independently known algorithms from the literature. These frameworks give insight into which concepts define known algorithms and which define new string-matching algorithms.

1.4 Correctness

The difficulty with the string-matching frameworks is to compare the specialized matchers with known string matchers. This comparison is difficult because the source code of the specialized matchers is different from the known matchers as a result of the specialization process. One such comparison between a KMP and a specialized naive string matcher was shown to be equivalent by Ager, Danvy and Rohde [1] using a formal proof. But this proof is complex and not a practical method of checking equality between large numbers of string matchers.

1.5 Propagation of Information

Rohde introduced a method of automatically comparing string matchers [21]. The method compares the traces of string matchers by applying the matchers to a set of inputs. It is inspired by the formal proof of Ager et al., and it automates the negative of that proof. The method is *counter-example driven*: if the traces disagree on any of the inputs, the matchers are not trace-equivalent. Since this method is automated, it solves the problem of comparing large numbers of string matchers.

Rohde described his *trace-based framework* and presented results from experimenting with his implementation. He, however, did not release it as an easy-to-use utility.

1.6 Contribution

In this thesis we complete Rohde's work. We present an implementation of a *trace-based framework* for comparing string matchers by using traces and for specifying *string-matching strategies*. A string-matching strategy is a set of concepts that together define a string matcher.

We have implemented our framework in Scheme, closely following Rohde's description of his framework. Our implementation splits and renames concepts and functions to make specifying string-matching strategies more explicit.

We have access to a large database of string matchers from Charras and Lecroq's Handbook of Exact String Matching Algorithms [6]. This *handbook* of string matchers has led us to a second contribution: three new methods of comparing string matchers.

1.6.1 Identifying a single string matcher

Our first method identifies a single string matcher. This method identifies which algorithm a given string matcher implements. The method takes a single matcher and compares it against a set of other matchers using a set of inputs. This method tells us on what inputs the given matcher is different from the other matchers, and which other matchers are equivalent with the given matcher. We use this method to verify that matchers from the literature implement the string-matching algorithms they claim.

1.6.2 Generating a table separating string matchers

Our second method generates a table separating string matchers. The table tell us which matchers are equivalent and can be used to identify a matcher by looking up the matchers traces in the table. The table consist of groups of trace-equivalent matchers, and which patterns, texts and corresponding traces separate each group from the other matchers in the table. To generate a table separating string matchers, we apply a given set of matchers with a large set of inputs. We then find the matchers that are trace-equivalent on all inputs, and a small set of inputs and corresponding traces that separate groups of trace-equivalent matchers from each other.

1.6.3 Building an evolutionary tree over string matchers

Our third method builds an evolutionary tree over string matchers. The tree shows how string matchers are related to each other. It presents an overview over string matchers grouped according to which string-matching concepts compose them. Furthermore, the tree gives insights into a matchers behavior without requiring a second matcher that is trace equivalent with the first. We determine how similar two string matchers are by using different trace comparison methods. We generate evolutionary trees by counting the number of shared traces between all pairs of matchers for a large set of inputs. We then apply the tool *QuickTree* (made by Howe, Bateman and Durdin [13]) to generate our tree using the neighbor joining method, which was introduced by Saitou and Nei [22].

Chapter 2

Related Work

This chapter presents the previous work that has inspired this thesis.

It consists of three parts: the first part contains descriptions of string-matching algorithms and the often-used concepts these algorithms use; the second part presents previous string-matching frameworks; and the last part provide an explanation of partial evaluation and how to specialize a naive $O(n^2)$ string-matching algorithm into a linear one for a static pattern.

2.1 String-matching Algorithms

This section describes the concepts of our string-matching strategies. We introduce *positive* and *negative* information as well as which *traversal order* is used to compare the pattern against text suffixes. Along with each string-matching concept, we introduce a corresponding string-matching algorithm.

```
k: 01234567890
t: abbabacabaa, tl=10
p:        abaa, pl=4
i:        0123
output: 7
```

This figure shows a string-matching problem with the pattern `p=abaa` and the text `t=abbabacabaa`. The output is where in `t` the first occurrence of `p` fits. Since `t[k] = p[i]` for $k = 7, 8, 9, 10$ and $i = 0, 1, 2, 3$ respectively, the output is 7. For patterns that do not occur in the text, the output is -1.

Figure 2.1: Notation of string-matching figures

2.1.1 Naive algorithm

The *naive algorithm* compares characters *from left to right*. On mismatches the pattern is moved one character to the right and comparison starts again

by comparing the first character of the pattern with the second character of the text. Moving the pattern in this fashion is called the *sliding window technique*. Moving the pattern one character to the right is also called *shifting the sliding window by one*.

```
k: 01234567890
t: abbabacabaa
?: ==!
p: abaa
?:  !
p:  abaa
?:   !
p:   abaa
?:    ===!
p:    abaa
...
?:        ====
p:        abaa
```

This figure shows how the naive algorithm checks whether `abaa` occurs in `abbabacabaa`

Figure 2.2: Naive algorithm example

2.1.2 Positive information

Positive information is knowledge about something that we know is true. For example, for the code "if $x = 1$ then C_1 else C_2", when we are in C_1 we know that x denotes 1, This knowledge is a positive piece of information. Positive information is used by the *Morris-Pratt string-matching algorithm*.

In Figure 2.3, the first mismatch is $\texttt{p}[2] \neq \texttt{t}[2]$. At this point, we know that $\texttt{p}[0] = \texttt{t}[0] = \texttt{a}$ and $\texttt{p}[1] = \texttt{t}[1] = \texttt{b}$. In the naive matcher, we would proceed to check $\texttt{p}[0] \stackrel{?}{=} \texttt{t}[1]$, but the positive information tells us that $\texttt{p}[0] = \texttt{a}$ and $\texttt{b} = \texttt{t}[1]$. Since $\texttt{a} \neq \texttt{b}$, we can shift the pattern two characters instead of one.

2.1.3 Negative information

Negative information is knowledge about something that we know is false. For example, for the code "if $x = 1$ then C_1 else C_2", when we are in C_2 we know that x does not denote 1. This knowledge is a negative piece of information. Negative information is used by the KMP algorithm. KMP use positive information and one entry of negative information (i.e., one mismatch of the form $\texttt{p}[\texttt{i}] \neq \texttt{t}[\texttt{k}]$).

Figure 2.4 shows comparisons being made by the KMP algorithm.

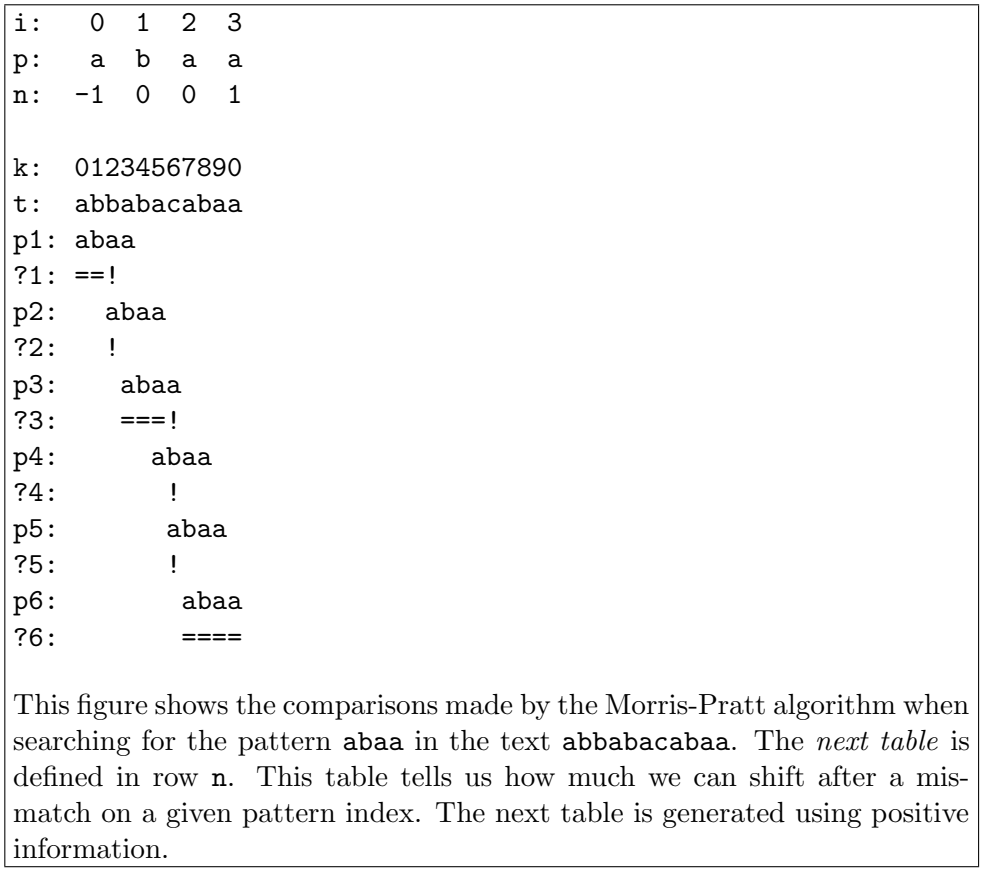


Figure 2.3: Morris-Pratt algorithm example

That we only use one entry of negative information is demonstrated in Steps 2 and 4 of the figure. After Step 2 we know that $\texttt{a} = \texttt{p}[3] \neq \texttt{t}[6] = \texttt{c}$ but in Step 4 we still compare $\texttt{a} = \texttt{p}[0] \stackrel{?}{=} \texttt{t}[6] = \texttt{c}$. This comparison is needed because our one entry of negative information was overwritten in Step 3 by $\texttt{b} = \texttt{p}[1] \neq \texttt{t}[6] = \texttt{c}$.

2.1.4 Bad character shift heuristic

The *bad character shift heuristic* uses characters in the text to determine where it is possible to align the pattern with the text. For a given character in the text, we can shift the pattern until the text character is opposite the same character in the pattern. To illustrate this concept, let us look at *Sunday's Quick Search* string-matching algorithm [25].

Figure 2.5 shows how the Quick Search algorithm searches for the pattern `abaa` in the text `abbabacabaa`.

```
i:   0  1  2  3
p:   a  b  a  a
f:  -1  0 -1  1

k:  01234567890
t:  abbabacabaa
p1: abaa
?1: ==!
p2:    abaa
?2:    ===!
p3:      abaa
?3:       !
p4:       abaa
?4:       !
p5:        abaa
?5:        ====
```

Comparisons made by the KMP algorithm when finding the pattern `abaa` in the text `abbabacabaa`. This algorithm uses positive information and one entry of negative information (i.e., one mismatch of the form $\texttt{p[i]} \neq \texttt{t[k]}$). `f` is the failure table that serves the same purpose as the next table from Morris-Pratt. The failure table uses one entry of negative information to improve the next table. After Step 1, we have the negative information $\texttt{a} = \texttt{p[2]} \neq \texttt{t[2]} = \texttt{b}$. We can skip the next step which would check $\texttt{p[0]} \stackrel{?}{=} \texttt{t[2]}$ since we already know that $\texttt{a} = \texttt{p[0]} \neq \texttt{t[2]} = \texttt{b}$. The skipped step is the lines `p2 and ?2` in Figure 2.3.

Figure 2.4: Knuth-Morris-Pratt algorithm example

2.1.5 Traversal order

Traversal order refers to the order in which the characters of the pattern is compared to a text suffix. To illustrate this concept, let us present the Boyer-Moore algorithm. This algorithm uses positive information, one entry of negative information, the bad character shift heuristics, and it compares the pattern to a text suffix from right to left.

Figure 2.6 shows how the Boyer-Moore algorithm looks for the pattern `abaa` in the text `abbabacabaa`. The number of shifts is the maximum of either using the failure-table or the bad character shift heuristics table. In Step 1, the failure table has the maximum value. In Step 3, however, the `bcs` has the maximum value because the character `c` is not part of the pattern.

```
      a  b  "all other characters"
bcs:  1  3  5

k:  01234567890
t:  abbabacabaa
p1: abaa
?1: ==!           bcs[t[4] = b] = 3
p2:    abaa
?2:    ===!       bcs[t[7] = a] = 1
p3:     abaa
?3:     !         bcs[t[8] = b] = 3
p5:        abaa
?5:        ====
```

Comparisons made by the Quick Search algorithm when finding the pattern `abaa` in the text `abbabacabaa`. This algorithm uses the bad character shift heuristic in the from of the table `bcs`. In Step 1, we determine that the current pattern alignment does not match and we decide how much to shift the pattern. If we shift by 1, the last character of the pattern `p`[3] will be matched with `t`[4] but `t`[4] = `b` ≠ `a` = `p`[3]. Shifting by 2 would result in `t`[4] = `b` ≠ `a` = `p`[2], but shifting by 3 gives us the comparison `t`[4] = `b` = `b` = `p`[1]. If the character in `t[4]` had not been in the pattern, we would know all comparisons between the pattern and `t[4]` would be mismatches, and we could skip to text index 4 by shifting 5 times.

Figure 2.5: Quick search algorithm example

2.1.6 Summary and conclusions

This section has defined string-matching algorithms and string-matching concepts.

We presented the following string-matching algorithms: naive, Morris-Pratt, Knuth-Morris-Pratt, quick search and Boyer-Moore. Together with these algorithms we introduced the string-matching concepts: positive information, negative information, the bad character heuristics table, and traversal order.

In the following section we introduce a different method of looking at string-matching algorithms: specialization. Specialization can transform a naive matcher into a new matcher implementing a different algorithm by keeping track of string-matching concepts.

```
     a  b  "all other characters"
bcs: 1  2  4

i:   0  1  2  3
p:   a  b  a  a
gs:  6  5  2  2

k:  01234567890
t:  abbabacabaa
p1: abaa
?1:   !=          max(gs[2], bcs[b]+(pl-1-i)) = max(2, 1) = 2
p2:  abaa
?2:     !         max(gs[3], bcs[b]+(pl-1-i)) = max(2, 2) = 2
p3:    abaa
?3:       !       max(gs[3], bcs[c]+(pl-1-i)) = max(2, 4) = 4
p4:        abaa
?4:        ====
```

Comparisons made by the Boyer-Moore algorithm when finding the pattern `abaa` in the text `abbabacabaa`. This algorithm uses the bad character shift heuristic. `bcs` is the bad character shift heuristics table and `gs` is the good-suffix table (which plays the same role as the failure table do in Knuth-Morris-Pratt). The equations in the right column show the failure table (called good-suffix) and the bad character shift heuristics table. `(pl-1-i)` is the length to the last index in pattern.

Figure 2.6: Boyer-Moore algorithm example

2.2 Specialization of String Matchers

In this section we describe specialization. We describe the general definition and how specialization can be applied to string matchers. We then describe how the KMP test is passed by two specialization techniques: polyvariant partial evaluation and positive supercompilation.

2.2.1 Specializing programs

We specialize a given program representation ($\ulcorner t_1 \times t_2 \to t_3 \urcorner$) with respect to an input (of type t_1) into a new program representation ($\ulcorner t_2 \to t_3 \urcorner$). Applying the new program with a single input (of type t_2) is equivalent to applying the original program with the two inputs of types t_1 and t_2.

$$\text{Specialization: } \ulcorner\ulcorner t_1 \times t_2 \to t_3 \urcorner \times t_1 \to \ulcorner t_2 \to t_3 \urcorner\urcorner$$

2.2.2 Specializing string matchers

A string matcher takes as input a pattern and a text and gives as output an index of where the first occurrence of the pattern occurs in the text. M is the representation of a string matcher:

$$M : \ulcorner pattern \times text \to \mathbb{Z} \urcorner$$

Applying a specializer S to SM with respect to a pattern likewise gives us a new program representation. This new program only takes a text as input and outputs an integer of the first point in the text we see the pattern we specialized SM with:

$$\text{run } S\langle M, \langle pattern, \_\rangle\rangle : \ulcorner text \to \mathbb{Z} \urcorner$$

For appropriate S and M, the specialized string matcher can be improved to run in linear time.

2.2.3 The KMP test

The KMP test is a canonical test for specialization techniques. It checks whether a quadratic runtime string matcher can be specialized with respect to its pattern into a linear runtime matcher. The KMP test was proposed by Futamura to show the power of generalized partial evaluation [11], and it was named by Sørensen et al. [24].

The KMP test has inspired examination into the generality of string matchers. This examination involves finding what it takes to specialize the naive string matcher into matchers known from the literature.

2.2.4 Specialization with polyvariant partial evaluation

Specializing a naive string matcher into a KMP matcher using polyvariant partial evaluation, which is simpler than generalized partial evaluation, was first achieved by Consel and Danvy [7]. The simpler specialization requires a *binding-time separated naive string matcher*. A binding-time separated naive string matcher is a naive string matcher that has been non-trivially modified in order to help the specialization technique. These modifications separate usage of the pattern (which is static), from usage of the text (which is dynamic). The idea for the modifications came from realizing that after a mismatch, the pattern is matched against a shifted version of itself up until the mismatch, as illustrated in Figure 2.7. This insight resulted in a binding-time separated naive string matcher that was specialized into a matcher having the same concepts as a KMP matcher. Since the specialized matcher had the same running time as KMP, the same order of comparing

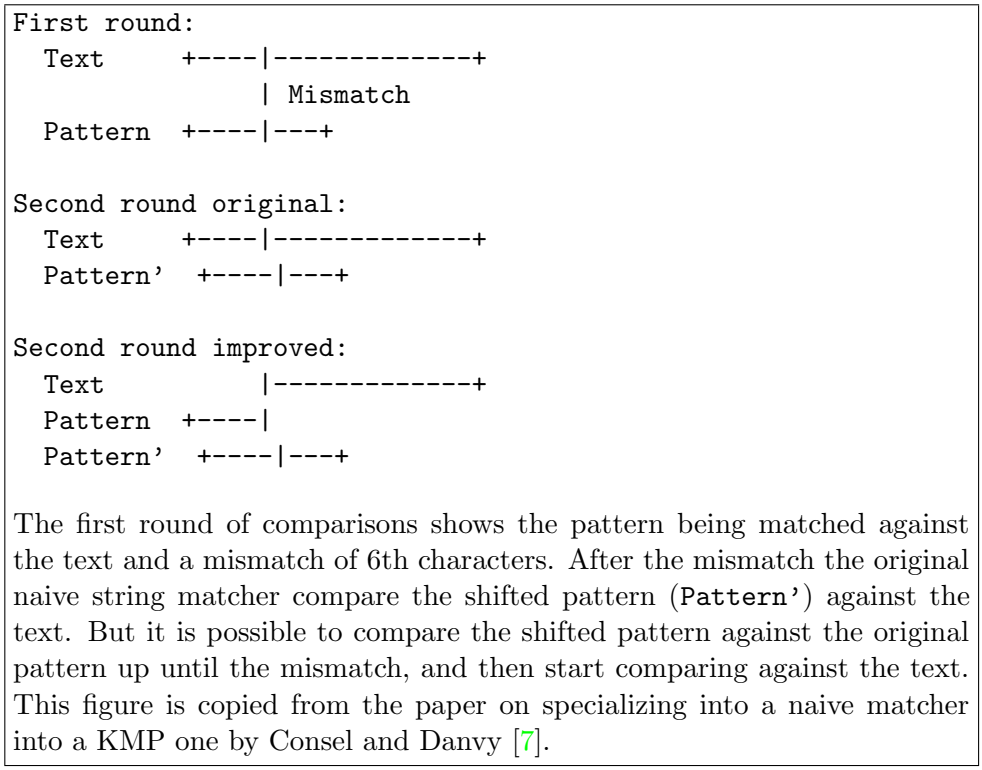


Figure 2.7: KMP specialization insight

characters, and the same failure table expressed in algorithmic form (for one pattern), it probably implemented KMP.[1]

Furthermore, specializing a binding-time separated naive string matcher into a Boyer-Moore one using the polyvariant partial evaluation technique was later achieved by Danvy and Rohde [9]. The naive matcher is modified to contain the two concepts of BM: the good-suffix table and the bad character shift heuristic. The good-suffix table is similar to the KMP's failure table, but working from right to left. A naive matcher modified to use the good-suffix table was found by Ager, Danvy and Rohde [2]. The bad character shift heuristic concept was implemented using a function consisting of a case-statement over each distinct character in the input. Combining these two modifications resulted in a binding-time separated naive string matcher that specializes into a matcher that have the same properties as a BM matcher. Since the specialized program was a string matcher, used the same order for comparing characters and has the same running time as BM, it probably implemented BM.

Checking equivalence between a specialized and a known string matcher is difficult. The previously mentioned results compared matchers by the

[1] "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."

duck test mentioned in Footnote 1. But to truly show equivalence, we need a formal proof that two string matchers perform the same text and pattern comparisons on all inputs. A formal proof that a specialized naive string matcher is equivalent with Knuth-Morris-Pratt has been presented by Ager, Danvy and Rohde [1]. The proof consists of formalizing the languages used by the specialized and the Knuth-Morris-Pratt algorithm, defining a comparison semantic, and showing that the two matchers perform the same comparisons. The proof, however, not an easy method of showing equivalence between the large number of specialized and known string matchers.

2.2.5 Specialization with positive supercompilation

Positive supercompilation is a transformation method that is more powerful than polyvariant partial evaluation but simpler than Futamura's generalized partial evaluation technique. It was introduced by Sørensen, Glück and Jones [24].

The positive supercompiler transforms programs written in a minimal first-order functional language. The transformation follows two phases: Positive driving and folding. Positive driving unfolds function calls for all encountered values of each of the arguments. Positive driving only saves positive information because arguments are only defined as being equal to values. Negative driving would save negative information, which would mean defining arguments as being *not* equal to some values. The folding phase combines identical functions generated in the driving phase. Without the folding phase, the driving algorithm would hardly ever terminate.

Using positive supercompilation, an unmodified naive string matcher can be specialized into a KMP-like string matcher. Figure 2.8 shows a matcher that has been specialized with respect to the pattern `AAB`. We can see that Lines 21-22 are redundant. We have this redundancy because negative information is not accounted for in positive supercompilation. The paper proves that specialized string matchers always run in linear time, meaning that positive supercompilation passes the KMP test. However, KMP uses one entry of negative information and since supercompilation does not take negative information into account, the specialized matcher cannot be KMP. Later in this thesis, in Section 4.4.4, we show that the matcher in Figure 2.8 likely implements the Morris-Pratt algorithm, and if we remove the redundant lines, the specialized matcher likely implements the KMP algorithm.

The paper presenting positive supercompilation does not focus on verifying that the specialized string matchers exactly implement a specific KMP-like algorithm. But the authors are well aware that there are differences between algorithms since the specialized matcher in this thesis' Figure 2.8 is identified as "Almost KMP specialized matcher" in the paper and another figure without the redundancy lines is identified as "KMP specialized matcher".

```
1  loop_AAB ss = case ss of
2                     []     -> False
3                 (s' : ss') -> if A = s'
4                                 then loop_AB ss'
5                                 else loop_AAB ss'
6
7  loop_AB ss  = case ss of
8                     []     -> False
9                 (s' : ss') -> if A = s'
10                                then loop_B ss'
11                                else if A = s'
12                                     then loop_AB ss'
13                                     else loop_AAB ss'
14
15 loop_B ss   = case ss of
16                    []     -> False
17                (s' : ss') -> if B = s'
18                                then True
19                                else if A = s'
20                                     then loop_B ss'
21                                     else if A = s'
22                                          then loop_B ss'
23                                          else loop_AAB ss'
```

This code is Figure 11 from the paper on positive supercompilation [24, p. 826].

Figure 2.8: Naive string matcher specialized by a positive supercompiler

2.2.6 Summary and conclusions

Several different specialization techniques, all simpler than Futamura's, have been used to pass the KMP test. Consel and Danvy's specialized matcher exactly implements the KMP algorithm, but it starts from a binding-time separated naive string matcher. Sørensen et al.'s specialization starts from a completely naive string matcher, but the specialized matcher implements the simpler MP algorithm, a distinction that can easily be found using our framework.

In the following section we introduce frameworks that can combine string-matching concepts to compose different binding-time separated string matchers.

2.3 String-matching Frameworks

String-matching frameworks generate quadratic-time string matchers using different concepts such as traversal orders from left to right or from right to left. The generated matchers of some of these frameworks are designed in a way that makes specializing them result in linear runtime string matchers from the literature such as the Knuth-Morris-Pratt or the Boyer-Moore matchers.

In this section we describe the mechanisms and contributions of 3 string-

matching frameworks. We present the frameworks in the chronological order of their publication: firstly the framework by Queinnec and Geffroy [18], secondly the framework by Amtoft, Consel, Danvy and Malmkjær [4] and lastly the framework by Rohde [21].

2.3.1 Queinnec and Geffroy

Queinnec and Geffroy's string-matching framework defines a language that uses, and searches, in S-expressions. We explain the language, how pattern matching is used as a *cache* and then how the framework creates KMP-like and BM-like string matchers. The main contribution is designing a framework that caches and reuse positive and negative information already checked.

The language searches through S-expressions. How to search and what to search for, is determined by a program defined in the language of the framework. S-expressions define tree-structures, but, for this thesis, we focus on strings. To apply terminology from string matchers, the text is here defined as a flat list of symbols (`(cons 'B (cons 'A (cons 'B (cons 'A (cons 'R '()))))))`). The pattern and the string matcher have been combined in a program defined in the language of the framework. This language, when limited to strings, is essentially a regular-expression matcher. Figure 2.9 explains a program defined in the framework.

```
(tree skip (cons (any) (hole skip))
           (cons 'B
            (cons 'A
             (cons 'B
              (cons 'A
               (cons 'R (any)))))))
```

A program defined in the language from Queinnec's framework. The program searches for the string `BABAR` from left to right. The command `(tree v a b)` either searches according to `b` or it searches according to `a` where `(hole v)` in `a` matches `(tree v a b)`. The command `(tree v a b)` plays the same role as the Kleene star does in regular expressions. This program is the same as the regular expression "`.*BABAR.*`".

Figure 2.9: Queinnec and Geffroy's framework example

The framework fills a cache with positive and negative information. The cache is represented by a language similar to the language defining the string matchers. If a text character is determined to be `A`, this is represented by `'A`. If a text character is determined to *not* be `A`, it is represented by `(not 'A)`. Some other commands are: `(and)` to combine representations, `(any)` to signify no information about a text character and `(cons)` to build

lists that show which positive and negative information is related to which text character. On a mismatch the cache is used to determine which text characters can be safely skipped in the next round of comparisons.

The framework is used to build a KMP-like and a BM-like string matcher. The KMP-like matcher generated is shown in Figure 2.9. The specialized string matcher is a set of mutually recursive functions. To generate the BM-like string matcher with an traversal order of right-to-left they introduce the `(xcons)` command. `(xcons v w)` is the same as `(cons w v)`, this means the right argument (`w`) is evaluated before the left (`v`). The generated string matchers have been shown to not be the KMP and the BM algorithms by Rohde [21, p. 30].

2.3.2 Amtoft et al.

Amtoft et al.'s string-matching framework uses a binding-time separated naive matcher modified to use a cache. Changing properties of the naive string matcher results in different string matchers. The framework is used to generate a KMP-like and a BM-like string matcher. The main contribution is showing that one string matcher can be specialized to the KMP and BM matchers by changing the order in which the pattern is compared to the text with.

String matchers are instantiated by changing properties of binding-time separated native string matcher. The properties that can be changed are traversal order and what positive and negative information to remove from the cache (called "pruning"). The ability to prune information from the cache is original to Amtoft's PhD thesis [3].

The cache is a list of either: a single character or another list of characters. A single character means a text character is known and is that single character — a positive information. A list of characters means a text character is known to not be any of the characters in that list — a negative information.

The framework generates several KMP-like and BM-like string matchers. The framework can generate a Morris-Pratt matcher by only saving positive information and it can generate a KMP-like matcher by saving both positive and negative information and pruning negative information. The generated matcher is not fully the KMP algorithm as it prunes the cache on matches but not on mismatches [21, p. 28]. The framework can generate several BM-like matchers. But the several generated BM-like string matchers are not trace-equivalent with matchers known to implement their corresponding BM-like algorithms. The generated matchers do not implement the BM-like because the framework does not implement table lookup to model the bad character shift heuristic table [9].

2.3.3 Rohde

Rohde's string-matching framework builds on Amtoft et al.'s framework. The string matcher in Rohde's has been expanded with more control over pruning and a table lookup mechanic was added. Table lookup emulates the bad character shift heuristic table. Furthermore, different instantiations of the expanded matcher can be combined to simulate using more than one cache. The main contribution is introducing traces of string matchers.

Generated string matchers caches information about the text after each comparison using two methods: *char* and *table*. The *char* method, which was used in Amtoft et al.'s string-matching framework, caches positive information for matches and negative information for mismatches. The *table* method caches positive information on both matches and mismatches.

Another improvement in this framework over that of Amtoft et al. is the ability to combine different string-matching strategies. This concept is used to define, among others, the Horspool algorithm [12]. The Horspool algorithm first compares the last character of the pattern and on a mismatches does a bad character shift heuristic table lookup. But on a match the algorithm compares the pattern with the text like the naive algorithm. The Horspool algorithm can be seen as a combination of a naive strategy that checks the last pattern character first and a strategy that only checks the last character using the *table* comparison method. These strategies are combined using the BACKTRACKING matcher as demonstrated and explained in Figure 2.10.

BACKTRACKING
— BASIC(*last-left-to-right*, *char*, *+none*, *-none*)
— BASIC(*last-only*, *table*, *+all*, *-all*)

This figure show the Horspool algorithm defined using Rohde's framework. The first BASIC function defines a string matcher comparing the last character of the pattern first, followed by the rest from left to right, as well as not adding any information to a cache. The second BASIC defines a matcher that only compares the last character of the pattern and it builds a bad character shift heuristic table. The BACKTRACKING matcher matches using its first argument up until a mismatch is found. On a mismatch, the number of shifts is determined as the maximum of both the matchers given as arguments.

Figure 2.10: Rohde's framework example

Rohde has used his string-matching framework to generate several known string-matching algorithms. The main difference between this framework and the others is the ability to compare the traces of the matchers. Applying

this method of comparison to the previous frameworks and other work have shown that some string matchers are not equivalent to their corresponding known algorithms.

2.3.4 Summary and conclusions

This section has presented the string-matching frameworks: Queinnec and Geffroy's, Amtoft et al.'s and Rohde's.

These frameworks have examined which string-matching algorithms we get when composing matchers using different string-matching concepts. Understanding which string-matching algorithm a composed string matcher implements, was greatly improved by Rohde's framework and its method of comparing string matchers by their traces.

2.4 Summary and conclusions

This chapter has described the background leading up to our thesis. We described string-matching algorithms and concepts. We described specialization and presented several examples of how it has been used to show how concepts and matchers are related. Lastly, we described string-matching frameworks, how they have been used to show connections between concepts and matchers, and how Rohde's framework introduced the method of comparing matchers by their traces.

In the following chapter we describe our own framework. Our framework builds on Rohde's, with more explicit naming when generating composed string matchers and more methods of comparing string matchers.

Chapter 3

Framework

This chapter describes our trace-based framework, and how we use it to generate and to compare string matchers.

Generation of string matchers is similar to that of Rohde's framework [21]. Compared to Rohde's framework, ours has a different cache and divides the basic string matcher up into several matchers. The framework is implemented in the Scheme programming language[14].

Comparing string matchers involve comparing the traces of matchers on a set of inputs. The string matchers are those generated from the framework and also matchers implemented in other languages. The set of inputs are patterns and texts of string permutations over a given alphabet and up to a given length. The result is a small number of inputs that differentiates as many of the string matchers as possible with the given set of inputs.

3.1 Composing String Matchers

This section describes how our framework was implemented. We give an overview over data structures and how the functions fit together to generate various string matchers.

3.1.1 String matcher

We look at the type of string matchers that check if the given pattern is a prefix of successive text suffixes. If the pattern is a prefix of the suffix the algorithm stop, but if the pattern is not a prefix of the suffix, the algorithm continue checking against a new text suffix. Checking whether the pattern is a prefix of a given suffix is called the *matching phase*.

Our framework generates string matchers by building *match* functions. A *match* function check one matching phase and returns. A *match* function takes as input a pattern, a text, an index of a text suffix, and a cache. As output, a *match* function returns whether a match was found, the number

of times we can shift the pattern safely, a cache, a trace, and the pattern index that caused the mismatch:

$$\begin{aligned}\text{match (types)} &: \Sigma^* \times \Sigma^* \times \mathbb{N} \times \textit{cache} \to \mathbb{B} \times \mathbb{N} \times \textit{cache} \times \mathbb{N}\ \textit{list} \times \mathbb{Z} \\ \text{match (contexts)} &: \text{pattern} \times \text{text} \times \text{text-suffix} \times \textit{cache} \to \\ &\qquad \text{found-match} \times \text{shifts} \times \textit{cache} \times \text{trace} \times \text{mismatch-index}\end{aligned}$$

3.1.2 Cache

Our cache holds positive and negative information about text characters. Text characters are referenced by their index, starting from 0. Positive information is represented by a single character from the alphabet Σ and negative information is a list of characters from the alphabet. A map from text indices to either a single character or a list of characters is called a *flat* cache:

$$\begin{aligned}\text{flat-cache (types)}&: \mathbb{N} \mapsto (\Sigma + \Sigma\ \textit{list}) \\ \text{flat-cache (contexts)}&: \textit{text-index} \mapsto (\textit{is-character} + \textit{is-not-characters})\end{aligned}$$

We also need to know from which matching phase each piece of information was gathered. We cache which information came from which matching phase by using a list of flat caches in reverse chronological order, meaning the information from the latest matching is the first flat cache in the list. This gives us our full cache:

$$\text{cache (types)}: (\mathbb{N} \mapsto (\Sigma + \Sigma\ \textit{list}))\ \textit{list}$$

In contrast, Rohde's framework defines a cache as a list of entries of positive or negative information where the indices of the entries determine which text character the information relates to.

The Scheme code defining the functions used to access the cache is available in Figure 3.1.

3.1.3 Basic matchers

The basic matchers take inputs that defines a string-matching strategy and generates a *match* function. The basic matchers determine four concepts: traversal order, how much positive and negative information to cache, whether to emulate a bad character shift heuristic table or not, and whether the cache is used during matching.

```
(define pos? char?)

(define (neg? e)
  (or (null? e) (pair? e)))

(define (cache-hit? cache j pat-char)
  "Whether the given cache has information about a text index j and a
character pat-char."
  (let ((e (cache-ref cache j)))
    (or (pos? e)
        (and (neg? e) (member pat-char e)))))

(define (cache-mismatch? cache j pat-char)
  "Whether the cache can tell us if the text index j is different
from the character pat-char."
  (let ((e (cache-ref cache j)))
    (if (pos? e)
        (not (equal? e pat-char))
        (member pat-char e))))
```

The list of functions used to access the cache. The missing function `cache-ref` searches the cache for the best information about a given text index. The best information is an entry of positive information, the next-best would be a list of negative information from all previous matching phases.

Figure 3.1: Our frameworks cache implementation

Traversal order

A function determines the traversal order the indices of the pattern is being checked in. We call this function an *orderer*. It takes as input the length of the pattern and returns a list of pattern indices:

$$\text{orderer (types)}: \mathbb{N} \to \mathbb{N}\ \textit{list}$$
$$\text{orderer (contexts)}: \textit{pattern-length} \to \textit{pattern-indices}$$

A table of the traversal orders being defined in our framework is displayed in Figure 3.2.

Traversal orders are defined as functions instead of just lists, but are otherwise the same as in Rohde's framework. The traversal orders table is essentially the same as in the paper on Rohde's framework [21].

The Scheme code defining the orderer functions is displayed in Figure 3.1.

How much positive and negative information to cache

A function determines much positive and negative information to cache. We call this function a *pruner*. It takes as input a cache and returns a potentially changed cache:

Traversal order	List of pattern indices
left-to-right(m)	$0, 1, \cdots, m-1$
right-to-left(m)	$m-1, m, \cdots, 0$
last-left-to-right(m)	$m-1, 0, 1, \cdots, m-2$
last-only(m)	$m-1$
second-only(m)	1
left-to-right-skip-second(m)	$0, 2, 3, \cdots m-1$
third-to-right-first(m)	$2, 3, \cdots m-1, 0$
last-first-middle-rest(m)	$m-1, 0, \frac{m}{2}, 1, 2, \cdots, \frac{m}{2}-1, \frac{m}{2}+1, \cdots, m-2$

m is the length of the pattern.

Figure 3.2: The traversal orders functions in our trace-based frameworks

$$\text{pruner (types): } \textit{cache} \rightarrow \textit{cache}$$
$$\text{pruner (contexts): } \textit{input-cache} \rightarrow \textit{pruned-cache}$$

The cache can be pruned by either removing information of a certain type, or it can be pruned by removing information of a certain *age*. The *age of information* is how many matching phases ago a bit of information was cached. The types of information in the cache is either positive or negative. A table of the pruner functions being used in this thesis is displayed in Figure 3.4.

In contrast, pruning the cache in Rohde's framework is defined using two functions: one determining what information to gather and another determining what information to prune. To prune all information older than 1, we define the two functions to gather and prune the same information.

The Scheme code defining the cache pruning functions is displayed in Figure 3.5.

Emulating a bad character shift heuristic table

Whether to simulate a bad character shift heuristic table is determined by which basic matcher we use. The matcher Table emulates a table, but Basic does not. The matchers take as input an *orderer* function and a *pruner* function and return a *match* function.

$$\textsc{Basic}, \textsc{Table} : \textit{orderer} \times \textit{pruner} \rightarrow \textit{match}$$

Emulating the bad character shift heuristic table is done by always saving positive information. So in case of a mismatch, we cache positive information using the text character (as opposed to saving negative information about the pattern character).

In contrast, Rohde's framework emulates a table in a comparator function. The comparator function compares a text and a pattern index as well as returning an entry of positive or negative information.

Use of the cache during matching

Using the cache during matching means that a comparison is only made if the result cannot be inferred from the cache. Whether the cache is used during matching is determined by which basic matcher we use. When the cache is not used during matching, it is only used to determine how many times we can shift the pattern safely.

The default matchers use the cache to potentially skip over comparisons. To not use the cache during matching we have the matchers: Basic-Shifts and Table-Shifts:

$$\textsc{Basic-Shifts}, \textsc{Table-Shifts} : \mathit{orderer} \times \mathit{pruner} \rightarrow \mathit{match}$$

In contrast, Rohde's framework model *not using the cache during matching* by pruning the same as is gathered. This Modeling works because Rohde's framework calculates the shifts before pruning the cache [21, p. 12]

The definition of the basic matcher

The Scheme code generating the main match function is displayed in Figure 3.6. Let us explain the important parts of the code.

Lines 9-10 check if the matching has finished successfully. If there are no more indices to check against, all previous indices must have matched.

Lines 11-14 check if the text and pattern indices are valid. Valid meaning within the range of the text and pattern strings.

Lines 20-21 determine whether to add the current text index to the trace. This depends on whether the cache contains enough information to determine the result of the comparison of the current text and pattern indices without having to access the text.

Lines 23-25 is the result of a match between the pattern and text characters. The result consists of updating the cache with positive information and continuing with the next pattern index.

Lines 26-29 is the result of a mismatch between the pattern and text characters. This result stops the algorithm and updates the cache with negative information. However, if we are emulating a bad character shift heuristic, we instead add positive information to the cache.

Line 30 starts the main loop. The loop is started using the traversal order function and an empty list is added to the cache to represent a new matching phase.

Lines 32-35 process the output. The number of shifts is calculated from the new cache and the trace is reversed to be in the correct order.

3.1.4 Composite matchers

Composite matchers take as input two *match* functions and returns a new combined *match* function. In this section we describe our five composite matchers and a special *match* that always fails. The composite matchers of our framework are the same as those from Rohde's framework[21, p. 13-20].

Since composite matchers uses two matchers, they also need two caches. In Rohde's framework these two caches could be aligned or non-aligned, depending on if the first element of the cache referred to the same index of the text or not, respectively. The caches in our framework do not need this distinction since they store which text indices refer to which elements.

The *match* functions given as input to composite matchers can be either *partial* or *complete*. A *partial match* function do not check all pattern indices, which means a partial *match* function *cannot* check if a pattern exists in a text. A *complete match* function, however, do check all pattern indices and is able to determine if a pattern exists in a text.

The trace of a composite matcher is the sequence from both *match* functions. In the full matcher we remove duplicate accesses in the same matching phase, from the trace.

Backtracking

The backtracking composite matcher works by using the first *match* function to check if a match exists, and the second *match* function to potentially shift the pattern more than a single character.

The idea of the backtracking composite matcher is presented using pseudo-code in Figure 3.7. The full Scheme code is available in Figure 3.8. We will not include the full Scheme code for the other composite matchers, since the code is hard to read, and mostly the same as the code for the backtracking composite matcher.

Using the backtracking composite matcher we can make a matcher implementing the Horspool algorithm. Defining a Horspool matcher in our framework is shown in Figure 3.9. The functions used are:

`make-matcher` This function converts a *match* function into an actual string matcher taking a pattern and a text as input.

`match-backtracking` This function check for a string using the first *match* function, and uses the second *match* function to possible shift by more than one.

`match-basic` This function is a naive string matcher that checks the last character of the patten first. We check the last character first to ensure the trace is correct. If we did not check the last character first, the second *match* function of the backtracking matcher would access the last text character later and its trace would be different from other matchers.

`match-table-shifts` This function calculate how much we can shift the pattern depending on the text character opposite the last pattern index.

`(prune-older-than 1)` This function ensures that the second *match* function only takes into account the latest last character, as opposed to all characters that have previously been the last in the pattern.

Alternate

The alternate composite matcher works similarly to the backtracking matcher, with the main difference of having the second *match* function start on a shifted text index. By shifted text index we mean that, if the first *match* function was run on text index k resulted in the pattern being shifted twice, the second *match* function would be run on text index $k + 2$.

The idea of the alternate composite matcher is presented using pseudo-code in Figure 3.10.

Using the alternate composite matcher we can make a matcher implementing Sunday's Quick Search. Defining a Quick Search matcher in our framework is shown in Figure 3.11.

The difference between the Quick Search and the Horspool matcher is which character is used to calculate shifts by looking up in the bad character heuristics table; Horspool uses the character opposite the last pattern, Quick Search uses the character after that one. In our framework, this difference is captured by using the alternate composite matcher.

Skew

The skew composite matcher also works similarly to the backtracking matcher, with the difference of having the second *match* function start on a text index, such that the last pattern character is where the first *match* function found a mismatch. For example, if the first *match* function found a mismatch between the pattern and the text on text index 2. The second *match*

function is run on text index 2 − *pattern-length*. This offset means the last pattern character in the second *match* will be opposite text index 3.

The idea of the skew composite matcher is presented using pseudo-code in Figure 3.12.

Using the skew composite matcher we can make a matcher implementing the Boyer-Moore algorithm. Defining a Boyer-Moore matcher in our framework is shown in Figure 3.13.

Sequential

The sequential composite matcher runs the first *match* function, then if that succeeds, runs the second *match* function and returns the highest number of shifts of the two *match* functions. The sequential composite matcher succeeds if both of the *match* functions succeed. Compared to the previous composite matchers, this matcher uses the second *match* function to verify the pattern is in the text, rather than just finding a higher number of shifts.

The idea of the sequential composite matcher is presented using pseudo-code in Figure 3.14.

Using the sequential composite matcher we can make a matcher implementing the not-so-naive algorithm [6]. Defining a not-so-naive matcher in our framework is shown in Figure 3.15.

Parallel

The parallel composite matcher runs both the first and second *match* functions and if they both succeed, then the composite matcher succeeds. The number of shifts is the highest of the two *match* functions.

The idea of the parallel composite matcher is presented using pseudo-code in Figure 3.16.

Using the parallel composite matcher we can make a matcher implementing the *Smith algorithm* [23]. The *Smith algorithm* shifts based on the highest amount of the two bad character heuristics tables from the Horspool and Quick Search algorithms.

Defining the Smith matcher in our framework is shown in Figure 3.17. We use the function `match-fail` here. This function is a *match* function that always fails, does not trace anything and shifts by one. This function is only used to shift the pattern by one. This function is used in the Quick Search part of the Smith matcher to move the pattern which enables Quick Search to access the character it needs to look up in the bad character heuristics table.

3.1.5 Summary and conclusions

This section has described how our trace-based framework composes string matchers from string-matching concepts.

We introduced how to define the string-matching concepts, and how to combine different matchers to generate complex matchers.

In the following section we describe how to record the traces of string matchers implemented in Scheme and C. To verify that our generated matchers implement the string-matching algorithms we claim, we need to compare against matchers that truly implement the algorithms. This verification requires comparing against these matchers, which means we need to record their traces.

```
(define (range-up-aux from to lst)
  (letrec ((walk (lambda (i)
                   (if (>= i to)
                       lst
                       (cons i (walk (1+ i)))))))
    (walk from)))
(define (range-up from to)
  "Returns the list of integers [from, from+1, ..., to-1]."
  (range-up-aux from to '()))

(define (range-down from to)
  "Returns the list of integers [to-1, to-2, ..., from]."
  (letrec ((walk (lambda (i)
                   (if (< i from)
                       '()
                       (cons i (walk (- i 1)))))))
    (walk (- to 1))))

(define (order-left-to-right pat-length)
  (range-up 0 pat-length))

(define (order-right-to-left pat-length)
  (range-down 0 pat-length))

(define (order-last-left-to-right pat-length)
  (cons (- pat-length 1)
        (range 0 (- pat-length 1))))

(define (order-last-only pat-length)
  (list (- pat-length 1)))

(define (order-second-only pat-length)
  (list (if (= pat-length 1) 0 1)))

(define (order-left-to-right-skip-second pat-length)
  (cons 0 (range-up 2 pat-length)))

(define (order-third-to-right-first pat-length)
  (range-up-aux 2 pat-length (list 0)))

(define (order-last-first-middle-rest pat-length)
  (case pat-length
    ((1) (list 0))
    ((2) (list 1 0))
    (else
     (let ((first 0)
           (middle (floor (/ pat-length 2)))
           (last (- pat-length 1)))
       (cons last
             (cons first
                   (cons middle
                         (range-up-aux (1+ first) middle
                                       (range-up (1+ middle) last)))))))))
```

The definitions of the traversal order functions. Each takes as input the length of the pattern and outputs a list of pattern indices. It is assumed that `pat-length` > 0.

Figure 3.3: The traversal order implementation in our trace-based framework

Pruner	What is pruned
none	Nothing
all	Everything
pos	All positive information
neg	All negative information
older-than(x)	Everything older than x
pos-older-than(x)	All positive information older than x
neg-older-than(x)	All negative information older than x

The functions *older-than*, *pos-older-than* and *neg-older-than* take as input a natural number and returns a pruner function.

Figure 3.4: The pruning functions in our trace-based frameworks

```
1  (define (prune-none cache)
2    cache)
3
4  (define (prune-all cache)
5    '())
6
7  (define (prune-pos cache)
8    (cache-filter (lambda (e) (not (pos? e))) cache))
9
10 (define (prune-neg cache)
11   (cache-filter (lambda (e) (not (neg? e))) cache))
12
13 (define (prune-pos-all-but i)
14   (lambda (cache)
15     (append (list-head' cache i)
16             (prune-pos (list-tail' cache i)))))
17
18 (define (prune-neg-all-but i)
19   (lambda (cache)
20     (append (list-head' cache i)
21             (prune-neg (list-tail' cache i)))))
22
23 (define (prune-all-but i)
24   (lambda (cache)
25     (list-head' cache i)))
```

The definitions of the cache pruning functions. The function `list-tail'` when applied to `list` and `k` returns a sub-list of `list` by omitting the first `k` elements. If `k` is greater than the length of `list` the empty list is returned. The function `list-head'`, when applied to `list` and `k` returns a sub-list of `list` containing the first `k` elements. If `k` is greater than the length of `list` the whole list is returned.

Figure 3.5: The pruning implementation of our trace-based framework

```
(define (match-basic-or-table orderer pruner is-table)
  "match-basic-or-table orderer pruner is-table
orderer * pruner * boolean ->
  (pattern * text * txt-offset * cache ->
    match? * shifts * cache * trace * mismatch-i)"
  (lambda (pattern text txt-offset cache)
    (letrec ((walk (lambda (pat-indices cache trace)
                     (cond
                      ((null? pat-indices)
                       (list #t cache trace -1))
                      ((or (>= (car pat-indices) (string-length pattern))
                           (>= (+ (car pat-indices) txt-offset)
                               (string-length text)))
                       (list #f cache trace -1))
                      (else
                       (let* ((i (car pat-indices))
                              (pat-char (string-ref pattern i))
                              (j (+ txt-offset i))
                              (txt-char (string-ref text j))
                              (trace' (if (cache-hit? cache j pat-char)
                                          trace (cons j trace))))
                         (if (equal? pat-char txt-char)
                             (walk (cdr pat-indices)
                                   (cache-upd-pos cache j pat-char)
                                   trace')
                             (list #f (if is-table
                                          (cache-upd-pos cache j txt-char)
                                          (cache-upd-neg cache j pat-char))
                                   trace' i))))))))
      (xapply (walk (orderer (string-length pattern)) (cons '() cache) '())
              (lambda (match? cache trace mismatch-i)
                (let ((cache' (pruner cache)))
                  (list match?
                        (cache-shifts pattern txt-offset cache')
                        cache' (reverse trace) mismatch-i)))))))
```

The definitions of the main matching function. The function `(xapply fun args)` is `apply` with reversed arguments. The function `(cache-shifts pattern txt-offset cache)` returns the number it is possible to shift a pattern which starts at a given text index by using a given cache. The function works by checking if each pattern index would cause a mismatch based on information in the cache. As long as there are mismatches, we can shift the pattern one step and check each pattern index again.

Figure 3.6: Our frameworks matching phase implementation

Input: integer *txt-offset*
Output: a boolean *match?* and an integer *shifts*
  $(match?1, shifts1) = match1(txt\text{-}offset)$
  if *match?1* **then**
    return $(\textbf{true}, shifts1)$
  else
    $(match?2, shifts2) = match2(txt\text{-}offset)$
    return $(\textbf{false}, \max(shifts1, shifts2))$
  end if

Pseudo-code describing the idea behind the backtracking composite matcher. Compared to the real implementation, this pseudo-code ignores handling caches, traces and on which index a mismatch occurred. We have left out this information as it is not read or modified in this composite matcher. Description of what the variables represent:

match1*, *match2 The first and last *match* function, respectively.

txt-offset Check if the pattern matches starting from this text index.

match?1*, *match?2 Whether the pattern was in the text according to the first and last *match* functions, respectively.

shifts1*, *shifts2 How much it is possible to shift the pattern for the next matching phase according to the first and second *match* functions, respectively.

Figure 3.7: Backtracking composite matcher pseudo-code

```
1  (define (match-backtracking match1 match2)
2    (lambda (pattern text txt-offset caches)
3      (let ((cache1 (if (pair? caches) (car caches) '()))
4            (cache2 (if (pair? caches) (cadr caches) '())))
5        (xapply (match1 pattern text txt-offset cache1)
6                (lambda (match?1 shifts1 cache1' trace1 mismatch-i1)
7                  (if match?1
8                      (list match?1
9                            shifts1
10                           (list cache1' cache2)
11                           trace1
12                           mismatch-i1)
13                     (xapply (match2 pattern text txt-offset cache2)
14                             (lambda (match?2 shifts2 cache2' trace2 mismatch-i2)
15                               (list match?1
16                                     (max shifts1 shifts2)
17                                     (list cache1' cache2')
18                                     (append trace1 trace2)
19                                     mismatch-i1)))))))))
```

This is our implementation of the backtracking composite matcher. This is essentially the same as the simple pseudo-code from Figure 3.7 that also shows how we handle caches, traces and on which index a mismatch occurred.
The big difference compared to the idea, is that this is a curried function. We take as input two `match` functions, then we return a *match* function to handle matching phases.
The extra inputs are a pattern, a text and a pair of caches used by the two *match* functions. The output needs a potentially updated version of the pair of the two caches (compare Lines 10 and 17).
The extra output are the caches, the trace and which pattern index had a character different from the text if a mismatch occurred. We only use the trace and pattern mismatch index from the first *match* function, since the second *match* function could be a partial matcher.

Figure 3.8: Our frameworks backtracking composite matcher

```
1 (define horspool
2   (make-matcher
3    (match-backtracking
4     (match-basic order-last-left-to-right prune-all)
5     (match-table-shifts order-last-only (prune-older-than 1)))))
```

This is how to define a Horspool matcher using our framework. The first *match* function is a complete matcher. The second *match* function is partial and only accesses characters that have already been accessed by the first *match*. The second *match* function emulates the bad character heuristics table to potentially increase shifting.
Note that, in each matching phase, the last character is accessed in both *match* functions. But we do not want to trace the same index more than once in one matching phase. We remove all duplicates index within a single matching phase, in the *make-matcher* function.

Figure 3.9: Horspool matcher defined in our framework

Input: integer *txt-offset*
Output: a boolean *match?* and an integer *shifts*
  (*match?1*, *shifts1*) = *match1*(*txt-offset*)
  if *match?1* **then**
    return (**true**, *shifts1*)
  else
    (*match?2*, *shifts2*) = *match2*(*txt-offset* + *shifts1*)
    if *match?2* **then**
      return (**false**, *shifts1*)
    else
      return (**false**, *shifts1* + *shifts2*)
    end if
  end if

Pseudo-code describing the idea behind the alternate composite matcher. Compared to the real implementation, this pseudo-code ignores handling caches and on which index a mismatch occurred. We have left out this information as it is not read or modified in this composite matcher
Description of what the variables represent:

match1*, *match2 The first and last *match* function, respectively.

txt-offset Check if the pattern matches starting from this text index.

match?1*, *match?2 Whether the pattern was in the text according to the first and last *match* functions, respectively.

shifts1*, *shifts2 How much it is possible to shift the pattern for the next matching phase according to the first and second *match* functions, respectively.

Figure 3.10: Alternate composite matcher pseudo-code

```
1 (define quick-search
2   (make-matcher
3    (match-alternate
4     (match-basic order-left-to-right prune-all)
5     (match-table-shifts order-last-only (prune-older-than 1)))))
```

This is how to define a Quick Search matcher using our framework. The first *match* function is a complete matcher and the second is partial matcher.

Figure 3.11: Sunday's Quick Search matcher defined in our framework

Input: integer *txt-offset*
Output: a boolean *match?*, an integer *shifts* and an integer *mismatch-i*
  $(match?1, shifts1, mismatch\text{-}i1) = match1(txt\text{-}offset)$
  if *match?1* **then**
    return $(\textbf{true}, shifts1)$
  else
    $(match?2, shifts2, mismatch\text{-}i2) =$
      $match2(txt\text{-}offset + mismatch\text{-}i1 + 1 - pattern\text{-}length)$
    return $(\textbf{false}, \max(shifts1, shifts2 + mismatch\text{-}i1 + 1 - pattern\text{-}length))$
  end if

Pseudo-code describing the idea behind the skew composite matcher. Compared to the real implementation, this pseudo-code ignores handling caches and traces. We have left out this information as it is not read or modified in this composite matcher.
Description of what the variables represent:

match1*, *match2 The first and last *match* function, respectively.

txt-offset Check if the pattern matches starting from this text index.

match?1*, *match?2 Whether the pattern was in the text according to the first and last *match* functions, respectively.

shifts1*, *shifts2 How much it is possible to shift the pattern for the next matching phase according to the first and second *match* functions, respectively.

mismatch-i1*, *mismatch-i2 The pattern index where a mismatch between the pattern and text was found.

pattern-length The length of the current pattern.

Figure 3.12: Skew composite matcher pseudo-code

```
1 (define boyer-moore
2   (make-matcher
3    (match-skew
4     (match-basic-shifts order-right-to-left (prune-older-than 1))
5     (match-table-shifts order-last-only (prune-older-than 1)))))
```

This is how to define a Boyer-Moore matcher using our framework. The first *match* function is a complete matcher and the second is a partial matcher.

Figure 3.13: Boyer-Moore matcher defined in our framework

Input: integer *txt-offset*
Output: a boolean *match?* and an integer *shifts*
  (*match?1*, *shifts1*) = *match1*(*txt-offset*)
  if *match?1* **then**
    (*match?2*, *shifts2*) = *match2*(*txt-offset*)
    return (*match?2*, max(*shifts1*, *shifts2*))
  else
    return (**false**, *shifts1*)
  end if

Pseudo-code describing the idea behind the sequential composite matcher. Compared to the real implementation, this pseudo-code ignores handling caches, traces and on which index a mismatch occurred. We have left out this information as it is not read or modified in this composite matcher. Description of what the variables represent:

match1*, *match2 The first and last *match* function, respectively.

txt-offset Check if the pattern matches starting from this text index.

match?1*, *match?2 Whether the pattern was in the text according to the first and last *match* functions, respectively.

shifts1*, *shifts2 How much it is possible to shift the pattern for the next matching phase according to the first and second *match* functions, respectively.

Figure 3.14: Sequential composite matcher pseudo-code

```
1 (define not-so-naive
2   (make-matcher
3    (match-sequential
4     (match-basic-shifts order-second-only (prune-older-than 1))
5     (match-basic order-third-to-right-first prune-all))))
```

This is how to define a not-so-naive matcher using our framework. Both the *match* functions are partial, but together they check all pattern characters, which means the resulting composite matcher is complete. The whole pattern is checked by the combination of both the first and second *match* functions. The first *match* checks the second character and the second checks the remaining characters.

Figure 3.15: Not-so-naive matcher defined in our framework

Input: integer *txt-offset*
Output: a boolean *match?* and an integer *shifts*
$(match?1, shifts1) = match1(txt\text{-}offset)$
$(match?2, shifts2) = match2(txt\text{-}offset)$
return $(match?1 \wedge match?2, \max(shifts1, shifts2))$

Pseudo-code describing the idea behind the parallel composite matcher. Compared to the real implementation, this pseudo-code ignores handling caches, traces and on which index a mismatch occurred. We have left out this information as it is not read or modified in this composite matcher. Description of what the variables represent:

match1*, *match2 The first and last *match* function, respectively.

txt-offset Check if the pattern matches starting from this text index.

match?1*, *match?2 Whether the pattern was in the text according to the first and last *match* functions, respectively.

shifts1*, *shifts2 How much it is possible to shift the pattern for the next matching phase according to the first and second *match* functions, respectively.

Figure 3.16: Parallel composite matcher pseudo-code

```
1 (define smith
2   (make-matcher
3    (match-backtracking
4     (match-basic order-left-to-right prune-all)
5     (match-parallel
6      (match-table-shifts order-last-only (prune-older-than 1))
7      (match-alternate
8       (match-fail)
9       (match-table-shifts order-last-only (prune-older-than 1)))))))
```

This is how to define a Smith matcher using our framework. The first *match* function of the backtracking composite matcher is the only complete matcher. We use the parallel composite matcher to find the highest number of shifts of using either Horspool or Quick Search. Getting the number of shifts of Quick Search requires the use of the alternate composite matcher since we need the character after the current last pattern character.

Figure 3.17: Smith matcher defined in our framework

3.2 Tracing string matchers

In this section we describe how we trace string matchers and our assumptions regarding matchers. Our assumptions ensure that the matchers are traced in an uniform manner, which in turn ensure that matchers implementing the same string-matching algorithm have the same traces.

We describe how to trace string matchers by first tracing a matcher implemented in Scheme, and which is straightforward to trace. We then talk about tracing more complicated string matchers implemented in the C programming language. These matchers are described together with our assumptions: how they can be ensured and why they are needed.

It is important to not change the behavior of string matchers in the process of adding tracing to it. We prevent changes to behavior by changing as few things as possible, and keeping close attention to the changes we do make.

3.2.1 String matcher in Scheme

In this section, we trace a Scheme implementation of the specialized string matcher from Figure 2.8.

The traced string matcher and its description is available in Figure 3.18.

3.2.2 String matchers in C

The string matchers implemented in C access text characters by two means: array lookup and using the `memcmp` function. We chose to trace array lookups instead of comparisons (as we did in the Scheme matcher) because some of the matchers store a text character in a temporary variable.

We added tracing to array lookups by replacing the lookup (e.g. `y[j]`) with a call to a function (`trace_get(y,j)`). The function `trace_get` records that the j'th text character have been accessed and returns that character.

We added tracing to the `memcmp` function by similarly replacing it with a new function. The `memcmp` function compares some elements of two arrays. For example `memcmp(x, y + j, m - 1)` performs these comparisons: $x[0] = y[j], x[1] = y[j+1], \cdots, x[m-1] = y[j+m-1]$ up until a mismatch occurs. To add tracing to this example we replace the `memcmp` call with `trace_memcmp(x, y + j, m - 1, j)`. The extra fourth argument j tells us where we start searching from in the text, we need this index to calculate which text index is being accessed.

But tracing every text character access is not a reliable approach, due to differences in programmers, languages and programming environments. We try to make the trace reliable by making assumptions about the behavior, inputs and tracing of matchers.

Traced Scheme matcher:

```
(define (soerensen-al-JFP96-fig-11 pat txt)
  (define trace '())
  (define (traced-equal? char s')
    (set! trace (cons s' trace))
    (equal? char (string-ref txt s')))
  (define txt-length (string-length txt))
  (define (loop-aab ss)
    (if (>= ss txt-length)
        #f
        (let ((s' ss)
              (ss' (1+ ss)))
          (if (traced-equal? #\a s')
              (loop-ab ss')
              (loop-aab ss')))))
  (define (loop-ab ss)
    (if (>= ss txt-length)
        #f
        (let ((s' ss)
              (ss' (1+ ss)))
          (if (traced-equal? #\a s')
              (loop-b ss')
              (if (traced-equal? #\a s')
                  (loop-ab ss')
                  (loop-aab ss')))))))
  (define (loop-b ss)
    (if (>= ss txt-length)
        #f
        (let ((s' ss)
              (ss' (1+ ss)))
          (if (traced-equal? #\b s')
              #t
              (if (traced-equal? #\a s')
                  (loop-b ss')
                  (if (traced-equal? #\a s')
                      (loop-ab ss')
                      (loop-aab ss')))))))
  (if (string= pat "aab")
      (begin (loop-aab 0)
             (reverse trace))
      '()))
```

This is a traced implementation, in Scheme, of the matcher from Figure 2.8. To avoid changing the behavior of the matcher, we have focused on staying as close as possible to the code in Figure 2.8. In order to have access to the text index, we work with a string as opposed to a list of characters. We have managed to encapsulate most of the work in the `traced-equal?` function. This function compares the character given as the first argument against the character in the string at the index given as the second argument, and the function saves the index as part of the trace.
Also note that this is a specialized matcher that returns the empty trace when given a pattern other than `aab`.

Figure 3.18: Tracing Sørensen et al.'s string matcher in Scheme

Our assumptions are rules requiring specific behavior, inputs and tracing of string matchers. These assumptions are needed because they ensure that the traces of the string matchers are comparable. Our assumptions work by ensuring string matchers have the same behavior in the parts we are not interested in differentiating, and that tracing is recorded in an uniform manner.

3.2.3 Matching after the first occurrence

We assume string matchers stop matching after finding the first occurrence of the given pattern in the given text.

Without this assumptions, matchers could continue matching after the first occurrence, which would mean more text indices would be recorded, and the traces of the matchers would not be comparable.

We ensure this assumption by stopping the string matchers if we find the pattern in the text.

3.2.4 Duplicate accesses in the same matching phase

We assume a text index is only accessed once per matching phase.

Recording when every text character is accessed can be too much. Some of the string matchers access the same text character twice without needing to. By "needing to" we mean we could store the text character in a temporary variable without changing the asymptotic time or memory usage of the matcher. These duplicate accesses often occur in matchers designed by humans, since an array lookup does not affect the behavior or the performance of the matcher, and having duplicate accesses can make the code easier to understand. However, duplicate access rarely occur without cause in matchers designed by automation such as specialization.

We ensure this assumption by checking for, and removing, duplicate accesses during the execution of a block of code. We do this using the two functions `start_pruning_duplicates()` and `stop_pruning_duplicates()`. These functions surround a block of code defining the matching phase. The functions remove any duplicate accesses made by the code block.

This method is demonstrated in the tracing of the Raita matcher, which is displayed in Figure 3.19 [19].

However, determining what a matching phase consist of, is not always straightforward. An example of this is the handbooks Morris-Pratt matcher (see Figure 3.21) and our frameworks Morris-Pratt matcher. The handbooks Morris-Pratt matcher have an inner loop in its matching phase to determine how much to shift, which can correctly access the same character twice. The Morris-Pratt matcher in our framework does not have this inner loop, and instead defines the behavior of the inner loop as separate matching phases.

Standard Raita matcher:

```
1  j = 0;
2  while (j <= n - m) {
3    c = y[j + m - 1];
4    if (lastCh == c
5        && firstCh == y[j]
6        && middleCh == y[j + m/2]
7        && memcmp(secondCh, y + j + 1, m - 2) == 0)
8      OUTPUT(j);
9    j += bmBc[c];
10 }
```

Traced Raita matcher:

```
1  j = 0;
2  set_txt_length(n);
3  while (j <= n - m) {
4    start_pruning_duplicates();
5    c = y[j + m - 1];
6    if (lastCh == c
7        && firstCh == y[j]
8        && middleCh == y[j + m/2]
9        && memcmp(secondCh, y + j + 1, m - 2) == 0)
10     OUTPUT(j);
11   j += bmBc[c];
12   stop_pruning_duplicates();
13 }
```

`x, y` pattern and text as a byte arrays

`m, n` the lengths of the pattern and text

`j` text index

`OUTPUT(j)` Indicate the pattern was found in the text at index j

`firstCh, middleCh, LastCh` the first, middle and last characters of the pattern

The main loop of a Raita matcher implemented in C, before and after adding tracing. There is also an explanation of the variables and functions. We need to prune tracing of duplicate indices, because the middle character is compared twice (once against `middleCh`, and once in the memcmp() function).

Figure 3.19: Tracing a Raita matcher in C

Because of this assumption, we trace the Smith matcher differently in our framework compared to Rohde's. This difference comes from a special case in Smith matcher: the last character can be accessed twice if we mismatch on the comparison involving the last pattern character. This case means that, in the trace of Rohde's Smith matcher, there is an extra index in the trace compared to our frameworks Smith matcher [21, p. 24].

3.2.5 Recording out-of-bounds text indices

We assume no out-of-bounds text indices are recorded. A text index is out-of-bounds if it does not refer to a character in the text.

This assumption ensures that string matchers made using different languages and in different environments are comparable. For example, In C, strings are terminated with a null byte. Fetching the terminating null byte from the text array can simplify the algorithm. The null byte, however, is not in a valid range of the text and should therefore not be traced.

We ensure this assumption by only tracing indices in the range $[0, n-1]$, where n is the length of the text. To be able to discard indices greater than the length of the text we call the function `set_txt_length(n)` which discards records indices greater than $n-1$.

This concept is demonstrated in the tracing of the not-so-naive matcher in Figure 3.20.

3.2.6 Pattern and text lengths

We assume our matchers are not applied with patterns or texts that consist of less than 3 characters.

This assumption is needed because some string matchers have undefined behavior for very short patterns and texts. These matchers can refer to specific indices (not-so-naive uses the second index) or the matchers divide the pattern up into several parts (the Raita algorithm uses the last, first and middle index). These matchers have undefined behavior when the pattern is very short; not-so-naive and the Raita algorithm assumes pattern lengths of at least 2 and 3, respectively.

We ensure this assumption by only using pattern and text inputs with sizes of at least 3. The number 3 was chosen because that is the maximum number of specific indices referred to in the descriptions of the algorithms we looked at. The maximum of 3 "specific indices referred to" is from the Raita algorithm.

To demonstrate what may happen if this assumption is not satisfied, let us say we applied a pattern of length 1 to the not-so-naive matcher displayed in Figure 3.20. When the second character of the pattern is read we get a null-character because this is implemented in C. A null-character is different from all characters in the text. The null-character causes the if-statement in line 3 to fail, which causes the implementation to skip to checking another text suffix. The if-statement only succeeds for the last suffix of the text. Therefore, when the pattern has length 1, the pattern is only checked against the last character of the text.

Standard not-so-naive matcher:

```
1  j = 0;
2  while (j <= n - m) {
3    if (x[1] != trace_get(y, j + 1))
4      j += k;
5    else {
6      if (memcmp(x + 2, y + j + 2, m - 2) == 0
7          && x[0] == y[j])
8        OUTPUT(j);
9      j += ell;
10   }
11 }
```

Traced not-so-naive matcher:

```
1  j = 0;
2  set_txt_length(n);
3  while (j <= n - m) {
4    start_pruning_duplicates();
5    if (x[1] != trace_get(y, j + 1))
6      j += k;
7    else {
8      if (trace_memcmp(x + 2, y + j + 2, m - 2, j + 2) == 0
9          && x[0] == trace_get(y, j))
10       OUTPUT(j);
11     j += ell;
12   }
13   stop_pruning_duplicates();
14 }
```

`x`, `y` pattern and text as a byte arrays

`m`, `n` the lengths of the pattern and text

`j` text index

`OUTPUT(j)` Indicate the pattern was found in the text at index `j`

`k`, `l` How much to shift when the second character doesn't and does match, respectively

The main loop of a not-so-naive matcher implemented in C, before and after adding tracing. There is also an explanation of the variables and functions. We need to define a text length to prune indices out-of-bounds of the text. If the text has a size less than 3, the `memcmp` will check against an index out-of-bound.

Figure 3.20: Tracing a not-so-naive matcher in C

3.2.7 Existence of at least one match

We assume our matchers are only applied with inputs where the pattern occurs in the text at least once.

This assumption is needed because some of our string matchers do not

stop matching as soon as the matcher know a match is no longer possible. Like with duplicate accesses, whether the matcher stops as soon as a match is not possible, or the matcher access a few more characters does not affect the asymptotic running time or memory usage of the matcher.

We ensure this assumption by only using inputs where there is always at least one occurrence of the pattern in the text. We do this by appending the pattern to the end of the text.

To demonstrate what may happen if this assumption is not satisfied, let us look at the non-traced Morris-Pratt C matcher in Figure 3.21. The variables `i` and `j` denote the next text and pattern indices to be compared. In the second while-statement on Line 3 a mismatch can occur while comparing against the end of the text. This mismatch sets the pattern index `i` without checking if the matcher should stop. Decrementing the pattern index has the same effect as comparing against a later text suffix. The matcher should stop here because the current pattern suffix cannot fit in the current text suffix.

3.2.8 Further assumptions

The assumptions we have listed are enough to ensure uniform tracing for the string matchers we have examined. But applying our framework to new programming environments or string-matching algorithms could require further assumptions.

For example, we did not need an assumption about evaluation order. If we apply a function with two text characters and each text character access has a side-effect of recording its index, different evaluation orders will result in different traces. Assuming the evaluation order is left-to-right would give us uniform tracing.

3.2.9 Summary and conclusions

This section has described how we add tracing to string matchers. To avoid the unreliable nature of traces, we have made assumptions about the behavior, inputs and tracing of matchers.

We assume that string matchers stop matching after the first found occurrence of the pattern the text, remove duplicate access in the same matching phase, do not record out-of-bounds text indices, use pattern and text lengths of at least 3 and that there must always exists at least one occurrence.

Our assumptions ensure that the string matchers are similar enough to be comparable. By making our matchers comparable, we also assume that our string matchers do not have interesting behavior outside of our assumptions. For example, we assume string matchers do not have specific

Morris-Pratt matcher:

```
1  i = j = 0;
2  while (j < n) {
3    while (i > -1 && x[i] != y[j])
4      i = mpNext[i];
5    i++;
6    j++;
7    if (i >= m) {
8      OUTPUT(j - i);
9      i = mpNext[i];
10   }
11 }
```

Traced Morris-Pratt matcher:

```
1  i = j = 0;
2  set_txt_length(n);
3  while (j < n) {
4    while (i > -1 && x[i] != trace_get(y,j))
5      i = mpNext[i];
6    i++;
7    j++;
8    if (i >= m) {
9      OUTPUT(j - i);
10     i = mpNext[i];
11   }
12 }
```

`x, y` pattern and text as a byte arrays

`m, n` the lengths of the pattern and text

`i, j` pattern and text index

`OUTPUT(j)` Indicate the pattern was found in the text at index j

`mpNext[i]` How much to shift when the `i`'th pattern character does not match a text character

The main loop of a Morris-Pratt matcher implemented in C, before and after adding tracing. There is also an explanation of the variables and functions. We have not added duplicates pruning here like we did in Figures 3.19 and 3.20. We do not prune duplicates here because it is part of the algorithm that the inner while-loop can access the same index more than once.

Figure 3.21: Tracing a Morris-Pratt matcher in C

behavior defined when no match is found or when given very short inputs. None of the algorithms we have looked at has any such behavior.

In conclusion, we have presented assumptions that ensure that the traces are comparable for the string matchers we have examined. Different string matchers, however, may reveal new forms of unreliability that require further

assumptions.

In the following section we describe several methods of comparing our generated string matchers and other matchers we have traced.

3.3 Comparing String Matchers

This section describe how we compare string matchers. We describe the type of input we test with. We also describe our three methods of comparing matchers: *identifying a single matcher*, generating a *table of separate string matchers* and building an *evolutionary tree*.

3.3.1 Test inputs

We test with permutations of strings up to a given length and an alphabet of a given size. The problem with checking all permutations (compared to fewer random strings of greater lengths) is that matchers can cache a specific amount of positive or negative information. A matcher storing a large number of entries of negative information is hard to distinguish from a matcher storing all negative information.

We chose our default test inputs as string permutations of lengths 3 and 4 over the alphabet a and b for the patterns and lengths 1 to 5 over the alphabet a, b and c for the texts. The extra symbol in the texts alphabet is to highlight the best-case scenario of a lookup in a bad character heuristics table. We have tried lengths up to 6 for both patterns and texts without separating our string matchers further.

3.3.2 Comparing two matchers

We compare two matchers using a set of inputs by comparing the resulting traces from applying the matchers with each input. The matchers are not trace equivalent if one of the inputs result in different traces. The matchers are trace equivalent (according to the inputs) if all of the inputs result in the same trace.

If the matchers are not trace equivalent, the input and trace that separate the two matchers, are from the first input resulting in different traces.

3.3.3 Identifying a single string matcher

Our first method identifies a given string matcher. This method compares the traces of the given matcher against the traces of a set of other matchers. The method returns the set of other matchers that were trace equivalent with the given one for all inputs.

This method of identifying string matchers takes as input a given matcher to identify, a set of matchers to compare against, and a set of inputs. The

method works by comparing the trace of the given matchers and the set of other matchers when applied to each input. Any other matchers whose trace do not match the given matchers are removed from further consideration. The algorithm stops when all inputs have been tested or when the set of other matchers is empty. If the set of other matchers is empty, we know the given matcher is different from all the matchers we compared against, and there is no need to apply more input. If all inputs have been tested, the given matcher is equivalent with the remaining set of matchers to compare against.

We handle identifying a given specialized matcher by ignoring the input the given matcher is not defined on. We handle comparing against specialized matchers by keeping track of the matchers that have not been defined on the same inputs as the given matcher. Using specialized matchers, however, can cause some problems. An example of this are the Morris-Pratt and Knuth-Morris-Pratt algorithms, The pattern `ab` results in the same next and failure table, and these tables are the only difference between the MP and KMP algorithms. The equivalent next and failure table means we cannot verify whether matchers specialized with respect to the pattern *ab* are trace-equivalent with KMP or MP matchers.

An example of this approach is displayed in Figure 3.22. We can see that from the figure that our given specialized KMP is different from a Morris-Pratt matcher on the pattern *aab* and the text *abaab*. The Figure also tells us that we did not share any input with the matcher *Specialized-MP-aba*, that was specialized with respect to the pattern *aba*. The conclusion is that our given specialized KMP is trace equivalent with a Knuth-Morris-Pratt matcher, and a KMP matcher with two negative entries. Knuth-Morris-Pratt matchers, however, are not trace equivalent with the KMP-2neg matcher as seen in Figure 3.23. The reason it concludes that our given matcher is trace equivalent with both KMP and KMP-2neg is because we the pattern *aab* cannot differentiate these two.

3.3.4 Generating a table separating string matchers

The *table separating string matchers* contains sets of trace-equivalent string matchers and the set of inputs and traces that separate each set from all other matchers in the table.

An example of a table separating string matchers is displayed in Figure 3.23. This table consist of three different string matchers, with inputs and corresponding traces that separate the three matchers.

To identify a given matcher using this table we would first apply our matcher with the pattern *aaa* and the text *abaa* to get a corresponding trace. If our trace is $0, 1, 1, 2, 3, 4$ then our matcher is trace-equivalent with MP (meaning with the matchers in the group containing the MP matcher in the table), and if our trace is $0, 1, 2, 3, 4$ our matcher is likely either KMP

```
Comparing Specialized-KMP-aab against 4 other matchers...

Specialized-KMP-aab is different from Morris-Pratt
   pattern 'aab' and text 'abaab'
   Specialized-KMP-aab trace: (0 1 2 3 4)
   Morris-Pratt trace: (0 1 1 2 3 4)

No more inputs

These matchers do not share any inputs with Specialized-KMP-aab:
   Specialized-MP-aba

The matcher Specialized-KMP-aab is trace equivalent with:
   Knuth-Morris-Pratt
   KMP-2neg
```

The result of trying to identifying a specialized KMP matcher against a KMP, a MP, a specialized MP matcher, and the KMP-2neg matcher which is a KMP matcher saving two entries of negative information.
Our inputs consisted of the pattern *aab*, and all texts up to a length of 5 with the alphabet *a*, *b* and *c*.

Figure 3.22: Output from identifying a specialized KMP string matcher

or KMP-2neg depending on another trace. The other trace we need is from applying our matcher with the pattern *abaa* and the text *abacabaa*. Like before, we look up this trace in the table to check if our given matcher is likely KMP or KMP-2neg.

Specialized string matchers are not used in the generation of tables separating matchers. Specialized matchers are only defined on one pattern, which forces us to use inputs on that pattern, and this would result in using more inputs and traces to separate the matchers in the table.

When we have identified a given matcher using this table, we do not know if our matcher is actually equivalent with the identified matcher. In fact, we only know that our matcher is different from all matchers in the table excluding the ones in the same groups as the matcher we were equivalent with.

Comparing a set of string matchers is a much harder problem than comparing two. In order to generate a table separating matchers in a reasonable time, we trim our data while applying input to the set of matchers. We use a greedy approach to find the sets of inputs and their corresponding traces separating the trace equivalent groups of matchers. We trim our data by occasionally calculating a temporary table from our temporary data, and then removing all inputs and traces not included in this temporary table.

pattern	text	trace	matcher
aaa	abaaa	0, 1, 1, 2, 3, 4	Morris-Pratt
aaa	abaaa	0, 1, 2, 3, 4	Knuth-Morris-Pratt
abaa	abacabaa	0, 1, 2, 3, 3, 3, 4, 5, 6, 7	
aaa	abaaa	0, 1, 2, 3, 4	KMP-2neg
abaa	abacabaa	0, 1, 2, 3, 3, 4, 5, 6, 7	

A table separating string matchers over the three matchers: Morris-Pratt, Knuth-Morris-Pratt and KMP-2neg which is a KMP matcher that stores two entries of negative information (opposed to one).
The table uses inputs consisting of patterns up to a length of 4 with the alphabet a and b, and texts up to a length of 5 with the alphabet a, b and c.

Figure 3.23: Table separating KMP-like string matchers

Greedy method of separating groups of trace-equivalent matchers

Here we describe our greedy method of separating groups of trace-equivalent string matchers, which results in a table separating matchers.

We have gathered inputs and traces data from our matchers. This data is a set over the tuple consisting of a pattern and text, together with sets of traces and the matchers that had that trace when applied to the pattern and text. This data separates two matchers if there exists a pattern and text that, when applied to our matchers, result in different traces.

Our greedy algorithm works like this:

1. Define all matchers as being equivalent.
2. Find the input and trace which separate the most matchers. Split the matchers into new groups according to the found input and trace.
3. Apply Step 2 on each new group of matchers as long as there exists an input and trace that can separate some matchers in at least one group.

To get the final table we parse and print the output from the greedy algorithm.

3.3.5 Building an evolutionary tree over string matchers

Evolutionary trees reveal the structure of string matchers, matchers that share concepts are closer to each other in the tree. This section describes how we determine similarity between matchers using two trace comparison methods, how these similarities are used to build evolutionary trees and the different effects our two trace comparison methods have on the built trees.

Determining similarity between matchers

The first part of building an evolutionary tree is determining how similar matchers are. We determine similarity by comparing the traces of our matchers when applied with a set of inputs. We compare the traces using two methods: the *naive* method which decide whether traces are equivalent or different, and the *pairwise-alignment* method which decide similarity based on how similar the traces are. Because of the tool we use to generate evolutionary trees, we do not compare the similarity between matchers, but rather the difference between matchers.

We determine the difference between the traces from a single input by applying one of our two comparison methods with each pair of matchers and their traces, when the traces are different. The comparison methods return a value determining how different the traces are. The sum of the values determining difference between whenever two matchers have different traces determines how different these two matchers are.

The naive trace comparison method

The *naive* trace comparison method decide whether the two traces compared are equivalent or not. This method always returns a value of 1 since the trace comparison methods are applied on matchers every time the matchers give a different trace. The *naive* method calculates the difference between two matchers as the number of inputs which caused the matchers to have different traces.

The pairwise alignment trace comparison method

The *pairwise-alignment* trace comparison method uses the Needleman-Wunsch algorithm [16] to determine how different two traces are. The Needleman-Wunsch algorithm determines how best to transform one string into another by a sequence of steps consisting of either inserting, removing or replacing one character.

We define the best method of transforming one string into another by assigning costs to the steps we use to transform strings. The *gap cost* is the cost of inserting or removing a character into the string, and the *difference cost* is the cost of replacing a character. The best method of transforming one string into another is the sequence of steps that have the minimum sum of costs of the steps. The sum of costs of the steps is the cost of transforming one string into another, and it is a value determining how different two strings are from each other.

The Needleman-Wunsch algorithm uses a dynamic approach by defining a solution in terms of solutions to shorter inputs. The algorithm builds a matrix with the best score for prefixes of the two strings the algorithm

compares. The idea behind the algorithm is displayed in Figure 3.24 using pseudo-code.

Function $cost(i, j)$:
if $T_{i,j}$ **then**
  return $T_{i,j}$
end if
$v1, v2, v3, v4 \Leftarrow$ *undefined*
if $i > 0 \wedge j > 0$ **then**
  if $A_i = B_j$ **then**
    $v1 \Leftarrow cost(i-1, j-1)$
  else
    $v1 \Leftarrow cost(i-1, j-1) + diffcost$
  end if
end if
if $i > 0 \wedge j \geq 0$ **then**
  $v2 \Leftarrow cost(i-1, j) + gapcost$
end if
if $i \geq 0 \wedge j > 0$ **then**
  $v3 \Leftarrow cost(i, j-1) + gapcost$
end if
if $i = 0 \wedge j = 0$ **then**
  $v4 \Leftarrow 0$
end if
$T_{i,j} \Leftarrow \max(v1, v2, v3, v4)$
return $T_{i,j}$

- The variables A and B are the strings we are comparing.
- The variables i and j are the prefixes of A and B, respectively.
- The variable T is the matrix storing the costs of all prefixes of A and B.
- The variables *diffcost* and *gapcost* denote the difference cost and the gap cost, respectively.
- The output of $cost(i, j)$ is a value representing the difference between the i'th prefix of A and the j'th prefix of B.

Figure 3.24: Pseudo-code of the Needle-Wunsch algorithm

Building evolutionary trees

An *evolutionary tree* over string matchers is a tree where the leaves are matchers. The distance between leaves determine how related the matchers in the leaves are, the shorter the distance the more related. We build an *evolutionary tree* from a *distance matrix* using the *neighbor joining* method, which was first described by Saitou and Nei [22].

A *distance matrix* represents how different a set of string matchers are from each other. A distance matrix D over n matchers is $n \times n$ and the element $D_{i,j}$ has a value representing how different the i'th matcher and the j'th matcher are from each other. An example of a distance matrix is displayed in Figure 3.25.

	Naive	MP	KMP	KMP-2neg
Naive	0	8544	16660	16632
MP	8544	0	8116	8088
KMP	16660	8116	0	28
KMP-2neg	16632	8088	28	0

This is a distance matrix defining the differences between the naive, MP, KMP and KMP-2neg matchers. The table was generated using the pairwise-alignment trace comparison method with a gap cost of 2 and a difference cost of 5.

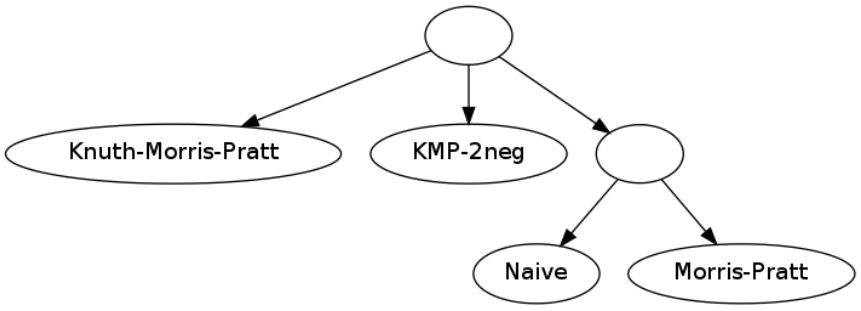
Figure 3.25: Distance matrix example

Using the distance matrix we make an evolutionary tree using the neighbor joining method. The neighbor joining method combines string matchers in the same subtree that are related to each other and different from all other matchers. The method is described below:

1. Define clusters for each string matcher.
2. Assign values to each cluster representing how different that cluster is from each of the other clusters.
3. Initialize a unrooted tree with a single node having all the string matchers as leaves.
4. Find the two clusters that are most related to each other and most different from all other clusters.
5. Combine these two clusters and redefine the values representing difference between this new cluster and the other clusters.
6. In the tree, combine the leaves or subtrees represented by the two clusters from Step 4. This combination creates a new node, which is connected to the initial node in the tree.

7. Go to Step 4 until there are only 2 clusters left.

An example of an evolutionary tree is displayed in Figure 3.26. This tree have split the matchers into two pairs: the naive and the MP matchers, and the KMP and KMP-2neg matchers. This split tells us that matchers in each pair are more similar to each other than they are to the opposite pair.



This is the evolutionary tree generated using the QuickTree tool based on the distance matrix in Figure 3.25. The tree was generated using the neighbor joining method.

Figure 3.26: Distance matrix example

We build our evolutionary trees using the *QuickTree* tool. QuickTree is a fast implementation of the neighbor joining method, and it was made by Howe, Bateman and Durbin [13].

3.3.6 Summary and conclusions

This section has introduced our methods of comparing string matchers. We have introduced a method of identifying a single matcher, a method showing us groups of trace-equivalent matchers, and a method of generating evolutionary trees that show us how matchers are related.

3.4 Summary and conclusions

This chapter has described our trace-based framework, how it generates string matchers, and its different methods of comparing string matchers. We also described how to trace string matchers, which are needed to have a set of known matchers to compare with.

In the following chapter, we present our results from using the framework to compare a set of string matchers. We describe what these results have revealed about the string matchers.

Chapter 4

Results

This chapter displays and describes our experiments on string matchers using our trace-based framework. The chapter has four sections: the first section describes our selection of which matchers we use in the experiments; the second section presents a table separating our chosen matchers; the third presents an evolutionary tree of the matchers; and the final section identifies matchers from papers we have cited, and describe these findings.

4.1 Selecting string matchers to experiment on

This section describes which string matchers we chose to experimented on, and why.

Our *chosen string matchers* is a mix of matchers consisting of:

- Implementations of algorithms from the literature defined using our framework. These implementations have names with the prefix *fw_*.
- Some interesting permutations of string matchers. A permutation matcher is a matcher we composed by combining permutations of concepts.
- The matchers described in the papers we have cited.
- The matchers from the handbook [6] which are trace equivalent with the previously mentioned matchers.

The permutations of string matchers are generated using our framework as compositions of the following string-matching concepts:

- whether to use a bad character heuristics table (*tbl*) or not (*no-tbl*);
- whether to avoid comparisons whose outcome is already known from the cache (*skip*) or not (*no-skip*);

- whether to check the pattern from left to right (*l2r*) or from right to left (*r2l*);
- how many matching phases worth of positive information to save: none (*0pos*), 1 (*1pos*), 2 (*2pos*), all of them (*pos*); and
- how many matching phases worth of negative information to save: none (*0neg*), 1 (*1neg*), 2 (*2neg*), all of them (*neg*).

An example of a generated matcher is *fw_no-tbl_skip_l2r_pos_1neg*. This matcher does not use a bad character heuristics table (*no-tbl*), it does skip comparisons if the cache tells us it is safe (*skip*), it compares the pattern from left to right (*l2r*), it saves all positive information (*pos*) and it saves negative information from up to 1 matching phase ago (*1neg*). This permutation matcher is trace equivalent with KMP.

Compositions of the string-matching concepts are not distinct, many of the string matcher permutations are trace equivalent. This is demonstrated in our table separating all of our matchers, which is displayed in Appendix A.

We have chosen string matcher permutations that are trace equivalent with our other chosen matchers. The names of permutation matchers define which concepts combine to make them, and in turn which concepts combine to make any matchers that are trace equivalent with the permutation matchers.

We have included the handbook matchers which are trace equivalent with our other matchers. The handbook matchers give us confidence that the other chosen matchers are correct.

4.2 Table separating our chosen string matchers

This section displays a table separating our chosen string matchers. We point out and describe interesting aspects of the table.

The table is displayed in Figure 4.1. Below are descriptions of some of the groups of trace-equivalent string matchers:

Naive The first group contains the naive matcher. We can see here that the naive matchers of our framework and the handbook are trace equivalent on the inputs tested. We have also included a permutation matcher to describe the concepts that compose a naive matcher. The naive permutation matcher is called *fw_no-tbl_skip_l2r_0pos_0neg*.

Morris-Pratt The Morris-Pratt group contains the Morris-Pratt string matchers from our framework and the handbook. We also have the permutation matcher *fw_no-tbl_skip_l2r_pos_0neg* in this group. Compared to the naive permutation matcher, this matcher saves positive information.

Automaton To explain how the Automaton string-matching algorithm works, let us look at the permutation matcher (*fw_tbl_skip_l2r_pos_0neg*) in this group. This permutation matcher uses a bad character heuristic table, which in our framework means negative information is saved as positive information. Because we use a bad character heuristics table we do not save any negative information in this matcher (or any other permutation matchers using a bad character heuristics table). The permutation matcher tells us that the Automaton string-matching algorithm compares the pattern from left to right using a bad character heuristics table.

fw_no-tbl_skip_r2l_0pos_0neg This permutation matcher compares the pattern from right to left, but is otherwise like the naive matcher. The other matcher in this group is described later in Section 4.4.

Partsch-Stomp Like with the Automaton group, this group tells us that the Partsch-Stomp string-matching algorithm compares the pattern from right to left, uses a bad character heuristics table and does not skip comparisons [17].

fw_no-tbl_skip_l2r_pos_neg This permutation matcher compares the pattern from left to right and saves all positive and negative information. This matcher can be described as an *optimal KMP matcher* because it caches all negative information compared to just one entry. In the same group we also have *fw_no-tbl_skip_l2r_pos_2neg*. This matcher is trace equivalent with the optimal KMP matcher in this table and also up to patterns and texts of lengths 7 over bigger alphabets, despite our initial intuition that these two matchers should be different.

The table separating all of our string matchers is displayed in Appendix A.

pattern	text	trace	matcher
aabb	aacbaaabb	0, 1, 2, 1, 2, 2, 3, 4, 5, 6, 5, 6, 7, 8	fw_no-tbl_skip_l2r_0pos_0neg consel-danvy-IPL89-naive-approach cl_naive fw_naive
aabb	aacbaaabb	0, 1, 2, 2, 2, 3, 4, 5, 6, 6, 7, 8	fw_no-tbl_skip_l2r_pos_0neg soerensen-al-JFP96-fig-18-fixed ager-al-TOPLAS06-fig-3 ager-al-TOPLAS06-fig-1 cl_morris_pratt fw_mp
aabb	aacbaaabb	0, 1, 2, 3, 4, 3, 6, 7, 5, 6, 7, 8	cl_smith fw_smith
aabb	aacbaaabb	0, 1, 2, 3, 4, 5, 6, 7, 8	fw_tbl_skip_l2r_pos_0neg cl_automaton fw_automaton
aabb	aacbaaabb	0, 1, 2, 4, 3, 7, 4, 5, 6, 8, 5, 6, 7, 8	cl_quick_search fw_quick-search
aabb	aacbaaabb	1, 2, 2, 4, 5, 5, 6, 6, 7, 8, 5	cl_not_so_naive fw_not-so-naive
aabb	aacbaaabb	3, 0, 1, 2, 4, 6, 8, 5, 6, 7	cl_horspool fw_horspool
aabb	aacbaaabb	3, 0, 2, 4, 6, 8, 5, 7, 6	cl_raita fw_raita
aabb	aacbaaabb	3, 2, 4, 5, 6, 7, 6, 8, 7, 6, 5	fw_no-tbl_skip_r2l_0pos_0neg danvy-rohde-IPL06-sec-2
aabb	aacbaaabb	3, 2, 4, 6, 8, 7, 6, 5	danvy-rohde-IPL06-sec-3 fw_horspool-right-to-left
aabb	aacbaaabb	3, 2, 4, 7, 6, 8, 6, 5	fw_no-tbl_skip_r2l_pos_neg amtoft-al-Jones02-right-to-left
aabb	aacbaaabb	3, 2, 7, 6, 8, 5	fw_tbl_skip_r2l_pos_0neg fw_optimal-bm
aabb	aacbaaabb	3, 2, 7, 6, 8, 7, 6, 5	fw_tbl_no-skip_r2l_1pos_0neg fw_partsch-stomp
aabb aaab	aacbaaabb aaacbaaab	0, 1, 2, 2, 3, 4, 5, 6, 6, 7, 8 0, 1, 2, 3, 3, 3, 4, 5, 6, 7, 8	consel-danvy-IPL89-still-naive-approach
aabb abab	aacbaaabb abcbaabab	3, 2, 6, 8, 7, 6, 5 3, 2, 6, 5, 4, 8, 7, 6, 5	fw_original-bm
aabb abab	aacbaaabb abcbaabab	3, 2, 6, 8, 7, 6, 5 3, 2, 7, 8, 7, 6, 5	danvy-rohde-IPL06-sec-4 cl_boyer_moore fw_boyer-moore
aabb aaab abaa	aacbaaabb aaacbaaab abacaabaa	0, 1, 2, 2, 3, 4, 5, 6, 6, 7, 8 0, 1, 2, 3, 3, 4, 5, 6, 7, 8 0, 1, 2, 3, 3, 3, 4, 5, 5, 6, 7, 8	fw_no-tbl_skip_l2r_pos_1neg consel-danvy-IPL89-further-optimization ager-al-TOPLAS06-fig-4 ager-al-2002-ASIA-PEPM02-fig-6 ager-al-2002-ASIA-PEPM02-fig-3 cl_knuth_morris_pratt fw_kmp
aabb aaab abaa	aacbaaabb aaacbaaab abacaabaa	0, 1, 2, 2, 3, 4, 5, 6, 6, 7, 8 0, 1, 2, 3, 3, 4, 5, 6, 7, 8 0, 1, 2, 3, 3, 4, 5, 5, 6, 7, 8	fw_no-tbl_skip_l2r_pos_neg fw_no-tbl_skip_l2r_pos_2neg amtoft-al-Jones02-left-to-right

A table separating our chosen string matchers. In the table, groups of matchers are surrounded by horizontal lines. The groups of matchers are trace equivalent with each other for the inputs used to generate this table. Furthermore, the inputs and traces for each group separate that group from all other groups.
The table used inputs consisting of patterns up to a length of 4 with the alphabet a and b, and texts up to a length of 5 with the alphabet a, b and c.

Figure 4.1: Table separating our chosen string matchers

4.3 Evolutionary tree of our chosen matchers

This section displays an evolutionary tree of our chosen string matchers. We describe interesting aspects of the tree.

The tree is displayed in Figure 4.2. We have limited the matchers to only include one from each of the trace-equivalent groups. Excluded matchers would be in the same place in the tree as the included matcher from the same trace-equivalent group as the excluded matcher. Find groups of trace-equivalent matchers in the table separating string matchers in Figure 4.1.

The evolutionary tree of all of our string matchers is displayed in Appendix B.1. To find where the excluded matchers belong in the tree, use the table separating all of our matchers in Appendix A.

The evolutionary tree reveals an overview over our string matchers. Let us look at some interesting aspects of the tree:

Three main branches The matchers are separated into three main branches. These branches are the left-to-right matchers, the right-to-left matchers and matchers using different traversal orders.

KMP-like area We have an area of the tree consisting of our KMP-like matchers: the KMP, MP, naive, optimal KMP and IPL89 matchers. This area is dividing up further by whether we cache negative information (KMP, optimal KMP) or not (naive, MP).

consel-danvy-IPL89-still-naive-approach This matcher is not trace-equivalent with any of our other matchers. Its position in the tree, however, tells us it is likely related to the KMP-like matchers.

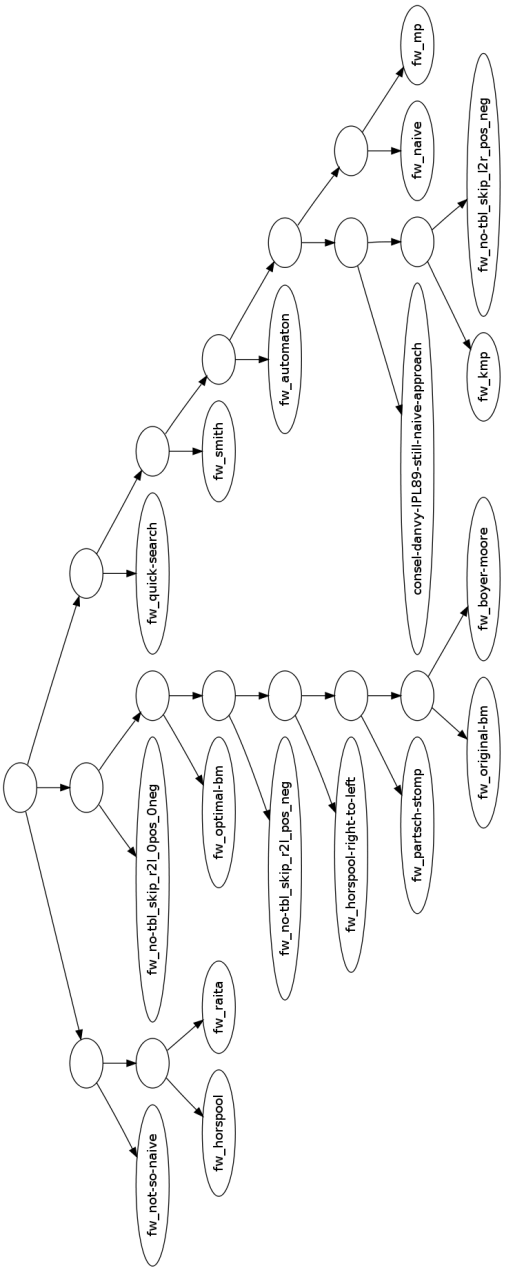
4.3.1 Different comparison methods

This section describes differences in evolutionary trees built using the naive and the pairwise-alignment trace comparison methods.

The naive method has a strong emphasis on separating matchers with different traversal orders. The method has this emphasis because traces will always be different if the first index compared is different. Only when the same traversal order is applied does this method notice different concepts between matchers. An example of a differing concept is whether to save negative information or not. Two matchers that save and do not save negative information will have different traces for some, but not all inputs, depending on the pattern. The three branches in the tree in Figure 4.2 is a result of the emphasis on traversal orders.

The majority of trees built using the pairwise-alignment comparison method are equivalent with tress built using the naive method. We built a tree using a high gap cost and a low difference cost which is displayed

in Appendix B.2. The high gap cost will emphasize differences in trace lengths because pairwise-alignment uses gaps to equalize the lengths of its input strings. In this tree, as opposed to the naive tree, the Horspool and Horspool-right-to-left matchers are close to each other. This closeness is because these two matchers use the same method of calculating shifts, which results in the lengths of the traces being similar despite the different traversal orders of the matchers.



This is an evolutionary tree over a distinct set of our chosen string matchers. Differences between the string matchers was found using the naive comparison method.
The tree was generated by comparing the amount of inputs resulting in the same trace. We used inputs consisting of patterns up to a length of 4 with the alphabet a and b, and texts up to a length of 5 with the alphabet a, b and c.

Figure 4.2: Evolutionary tree of our chosen matchers

4.4 Identifying string matchers from the literature

This section presents our experiments on string matchers from the papers we have cited in this thesis. These source for these matchers were copied directly, or translated to Scheme, from the papers. The experiments were performed using the method for identifying a single string matcher.

Copying the matchers means we can't choose which patterns they were specialized with respect to. Specialized matchers can only be compared on inputs involving one pattern, which means we cannot always uniquely identify which algorithm a specialized matcher implements.

Unless we say otherwise, the matcher was applied with pattern permutations of lengths 3 to 4 over the alphabet a, b and c, and on text permutations of lengths up to 5 over the alphabet a, b, c and d.

The specialized matchers in this section are not included in the table separating string matchers or the evolutionary tree. The table and tree only use matchers defined on all inputs.

The last part of the names of specialized matchers tells us which pattern the matchers were specialized with respect to. When comparing these specialized matchers we set the pattern to the one the matcher was specialized with respect to, and we use text permutations up to a length of 7 and over the alphabet of the pattern, with an extra symbol added.

When we say a given matcher is trace equivalent with, e.g., KMP. We actually mean that the given matcher is trace equivalent with KMP on the inputs we have tested with. The given matcher could be different from KMP on large inputs.

The full output from using our method in identifying these matchers is available online.[1]

4.4.1 Knuth, Morris and Pratt, 1977

The Knuth, Morris and Pratt [15] paper included a specialized KMP matcher.

The specialized KMP matcher is called *knuth-morris-pratt-SIAM77-abcabcacab*. We found that this matcher is trace equivalent with KMP, but also with optimal KMP. From the table in Figure 4.1, however, we know that KMP and optimal KMP are not trace equivalent. This inconsistency comes from not comparing traces for enough input. We can only make inputs with the single pattern *abcabcacab*. And these inputs cannot separate KMP and optimal KMP matchers. This means that for inputs having this pattern, the three matchers KMP, Optimal KMP and *knuth-morris-pratt-SIAM77-abcabcacab* will have the same trace. The paper presents the specialized matcher *knuth-morris-pratt-SIAM77-abcabcacab* as being a KMP string matcher, which means it is not intended to be trace-equivalent with KMP Optimal.

[1] http://www.danamlund.dk/masters_thesis

4.4.2 Consel and Danvy, 1989

This paper by Consel and Danvy [7] describes how to specialize a naive string matcher into a KMP matcher. The paper contains three different matchers: a *naive matcher*, a *KMP matcher* and a *matcher in between* the other two.

The naive string matcher is called *consel-danvy-IPL89-naive-approach*. We found that this matcher is trace equivalent with the naive group.

The in-between string matcher is called *consel-danvy-IPL89-still-naive-approach*. We found that this matcher is trace equivalent with the specialized matcher *soerensen-al-JFP96-fig-4-aab*. That matcher, however, is trace equivalent with KMP, which the in-between matcher is not. The reason the in-between matcher is trace equivalent with this other matcher, is, again, because the other matcher is defined on limited input. From these results we cannot say which algorithm the in-between matcher implements. We also have an in-between matcher specialized with respect to the pattern *ababc*. This specialized matcher is trace-equivalent with the in-between matcher and nothing else.

The KMP string matcher is called *consel-danvy-IPL89-further-optimization*. We found that this matcher is trace equivalent with KMP and with these two specialized matchers (again due to patterns not separating them): *amtoft-al-Jones02-fig-2-aaa* and *soerensen-al-JFP96-fig-4-aab*. This KMP matcher also has a version specialized with respect to the pattern *abcabcacab*. In the paper, this specialized matcher was compared against the specialized matcher from the Knuth, Morris and Pratt paper to argue that *consel-danvy-IPL89-further-optimization* is a KMP matcher. Since the specialized matcher has limited input we cannot say if it is equivalent with KMP or Optimal KMP. But the fully-defined matcher *consel-danvy-IPL89-further-optimization* is equivalent with KMP, meaning we have shown that this matcher implements the KMP algorithm.

4.4.3 Queinnec and Geffroy, 1992

This paper by Queinnec and Geffroy [18] presents a string-matching framework and includes two specialized string matchers: one checking the pattern from left to right and one checking from right to left. We described the string-matching framework from this paper in Section 2.3.1.

The left-to-right matcher is called *queinnec-geffroy-WSA92-babar*. We found that this matcher is trace equivalent with KMP and optimal KMP, but those two matchers are not trace equivalent with each other. As before, this is because the pattern *babar* cannot separate KMP and optimal KMP matchers. From the paper we know that no negative information is pruned, which tells us that the specialized matcher is not intended to be equivalent with KMP.

The right-to-left matcher is called *queinnec-geffroy-WSA92-foo*. We found that this matcher is trace equivalent with the permutation matcher *fw_no-tbl_skip_r2l_pos_neg* which can also be described as a *right-to-left optimal KMP matcher*.

4.4.4 Sørensen et al., 1996

This paper by Sørensen, Glück and Jones [24] presents the positive supercompiler technique and includes two specialized matchers: one specialized using this technique and an example of a specialized KMP matcher. The paper also includes a third binding-time separated string matcher.

The specialized KMP matcher is called *soerensen-al-JFP96-fig-4-aab*. We found that this matcher is trace equivalent with both KMP and optimal KMP, this is, yet again, due to the pattern not separating these two matchers.

The matcher specialized using positive supercompilation is called *soerensen-al-JFP96-fig-11-aab*. We found that this matcher is trace equivalent with MP.

The binding-time separated string matcher is called *soerensen-al-JFP96-fig-18-fixed*. We found that this matcher is trace equivalent with MP.

4.4.5 Amtoft et al., 2002

This paper by Amtoft, Consel, Danvy and Malmkjær [4] presents string-matching framework and includes four matchers: two matchers checking the pattern from left to right and from right to left, respectively, a specialized version of previous left to right matcher and lastly a specialized Partsch-Stomp matcher composed using this string-matching framework. We described the string-matching framework from this paper in Section 2.3.2.

The matcher checking the pattern from left to right is called *amtoft-al-Jones02-left-to-right*. We found that this matcher is trace equivalent with optimal KMP.

The specialized version of the left-to-right matcher is called *amtoft-al-Jones02-fig-2-aaa*. We found that this matcher is trace equivalent with KMP, optimal KMP and Automaton. The reason the Automaton matcher is included is because the pattern consist of a single symbol, which causes a bad character heuristics table to have the same behavior as saving negative information.

The matcher checking the pattern from right to left is called *amtoft-al-Jones02-right-to-left*. We found that this matcher is trace equivalent with the right-to-left optimal KMP matcher.

The specialized Partsch-Stomp matcher is called *amtoft-al-Jones02-fig-3-abb*. This matcher was not trace equivalent with any of our matchers. To have the specialized matcher be trace equivalent with Partsch-Stomp,

we modified the tracing to only access each text index once per matching phase. This modification gave us the matcher called *amtoft-al-Jones02-fig-3-abb-prune-duplicates*, which, as mentioned, is trace equivalent with Partsch-Stomp.

4.4.6 Ager et al., 2002

This paper by Ager, Danvy and Rohde [1] proves that a specialized string matcher is a KMP matcher. The paper includes two KMP matchers, one with tail recursion and one without.

The matcher with tail recursion is called *ager-al-2002-ASIA-PEPM02-fig-3*. We found that this matcher is trace equivalent with KMP.

The matcher without tail recursion is called *ager-al-2002-ASIA-PEPM02-fig-6*. We found that this matcher is also trace equivalent with KMP.

4.4.7 Ager et al., 2006

This paper by Ager, Danvy and Rohde [2] shows how to specialize a KMP matcher in linear time. The paper presents three matchers: a binding-time separated MP matcher, the previous modified to ensure fast specialization and the last matcher which further adds one entry of negative information to make a KMP matcher.

The first matcher is called *ager-al-TOPLAS06-fig-1* and is trace equivalent with MP.

The second matcher is called *ager-al-TOPLAS06-fig-3* and is also trace equivalent with MP.

The third matcher is called *ager-al-TOPLAS06-fig-4* and is trace equivalent with KMP.

4.4.8 Danvy and Rohde, 2006

This paper by Danvy and Rohde [9] presents a binding-time separated naive string matcher that can be specialized into a Boyer-Moore matcher. The paper presents four matchers: a right-to-left naive one, the previous matcher using a bad character shift heuristics table, a specialized version of the previous matcher and, finally, a Boyer-Moore matcher.

The right-to-left naive matcher is called *danvy-rohde-IPL06-sec-2* and is trace equivalent with the permutation matcher *fw_no-tbl_skip_r2l_0pos_0neg*, which is a right-to-left naive matcher.

The matcher using a bad character heuristics table is called *danvy-rohde-IPL06-sec-3*. We found that this matcher is trace equivalent with *horspool-right-to-left*. The *horspool-right-to-left* matcher is a Horspool matcher that compares the pattern from right to left.

The specialized version of the bad character heuristics table matcher is called *danvy-rohde-IPL06-sec-3-aba* and this matcher is trace equivalent

with *horspool-right-to-left* and several other right-to-left matchers due to the pattern not being able to separate them.

The Boyer-Moore matcher is called *danvy-rohde-IPL06-sec-4* and is trace equivalent with Boyer-Moore.

4.4.9 Summary

This section has described our results from applying our trace-based framework to matchers from the literature. An overview over these results are presented in Appendix C.

4.5 Summary and conclusions

This chapter has presented our results from comparing a set of string matchers using our trace-based framework. We have shown how we can investigate and understand string matchers by comparing their traces using different methods.

In the last chapter we conclude our dissertation and present perspectives on our experiences from working with string matching.

Chapter 5

Conclusion and Perspectives

5.1 Conclusion

Using our trace-based framework, we have investigated, compared and deconstructed string-matching algorithms. We have investigated string-matching algorithms known from the literature, compared these and many more with each other using traces, and we have deconstructed string-matching algorithms into common concepts that can be combined in different ways to make different string-matching algorithms.

Our trace-based framework lets us investigate, understand and build string matchers through these four contributions:

- the ability to compose string matchers by combining string-matching concepts;
- a method to identify a string matcher by comparing it against a set of known matchers;
- a method that combines matchers into trace-equivalent groups and lets us identify which group a given matcher belongs in; and
- a method that reveals how string matchers are related to each other by building evolutionary trees over them.

Obtaining the Knuth-Morris-Pratt algorithm by specializing a string matcher was first envisioned by Futamura [11]. That families of string matchers can also be obtained by partial evaluation was conjectured by Danvy [8, p. 72]. Composing string matchers by combining concepts was first implemented by Queinnec and Geffroy [18]. Using traces to compare string matchers was first presented by Rohde [21]. Using the number of equivalent traces over a set of inputs to indicate similarity between matchers and generating an evolutionary tree from this data is an original contribution of the present thesis.

5.2 Perspectives

From working on our trace-based framework and string-matching algorithms in general, we have seen several approaches to looking at, and designing string matchers. Being able to compare string matchers designed using different methods has given us a better understanding of the general structure of string matching.

Future work Future work on evolutionary trees over string matchers could be designing new comparison methods to highlight specific differences between matchers. For example, we could implement a technique that determines traces be similar if reversed parts of one trace exist in the other. This technique could cause left-to-right and right-to-left matchers to be closer in evolutionary trees. Such evolutionary trees could reveal new insights into the effects of these two traversal orders on string matchers.

A new approach to understanding string matching Furthermore, our work has revealed a new approach to understanding string matching. The traditional approach to understanding string matching requires one to understand individual string-matching algorithms, going through the list of algorithms one at a time. Using this approach, new insights come in the form of new algorithms. Each new algorithm is added to the list of string-matching algorithms in no particular order.

During this work, however, our approach to understanding string matching has focused on understanding string-matching concepts and how to combine these concepts to obtain composite string matchers. With our approach, new insights come in the form of different concepts or matchers combining concepts in a novel way. These new insights can be examined using our methods of comparing string matchers, which lets us know if the insight is truly new and reveals how a new concept affects string matchers, or how the new matcher is related to our previous string matchers.

Closing words All in all, we have implemented and described a trace-based framework. We have used this framework to compare string matchers and to build evolutionary trees over them. Comparing string matchers has helped us identify matchers from the literature. Evolutionary trees over matchers have given us an overview over how string matchers are related to each other. In short, we have shown that trace-based frameworks are powerful tools for investigating, understanding and building string-matching algorithms.

Bibliography

[1] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. On obtaining Knuth, Morris, and Pratt's string matcher by partial evaluation. In Wei-Ngan Chin, editor, *Proceedings of the 2002 ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation (ASIA-PEPM 2002)*, pages 32–46, Aizu, Japan, September 2002. ACM Press. Extended version available as the research report BRICS RS-02-32. 3, 4, 5, 15, 66

[2] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. Fast partial evaluation of pattern matching in strings. *ACM Transactions on Programming Languages and Systems*, 28(4):696–714, July 2006. A preliminary version was presented at the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2003). 14, 66

[3] Torben Amtoft. *Sharing of Computations*. PhD thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, 1993. Technical report PB-453. 4, 18

[4] Torben Amtoft, Charles Consel, Olivier Danvy, and Karoline Malmkjær. The abstraction and instantiation of string-matching programs. In 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, pages 332–357. Springer, 2002. 4, 17, 65

[5] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20:762–772, October 1977. 3

[6] Christian Charras and Thierry Lecroq. *Handbook of Exact String Matching Algorithms*. King's College Publications, 2004. 6, 28, 56

[7] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, January 1989. 4, 13, 14, 64

[8] Olivier Danvy. *An Analytical Approach to Program as Data Objects*. DSc thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, October 2006. 68

[9] Olivier Danvy and Henning Korsholm Rohde. On obtaining the Boyer-Moore string-matching algorithm by partial evaluation. *Information Processing Letters*, 99(4):158–162, 2006. 14, 18, 66

[10] Edsger W. Dijkstra. Concern for correctness as a guiding principle for program composition. EWD 288, July 1970. Available online at http://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD288.html. 4

[11] 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. 4, 13, 68

[12] R. Nigel Horspool. Practical fast searching in strings. *Softw., Pract. Exper.*, 10(6):501–506, 1980. 19

[13] Kevin Howe, Alex Bateman, and Richard Durbin. Quicktree: building huge neighbour-joining trees of protein sequences. *Bioinformatics*, 18(11):1546–1547, 2002. 6, 55

[14] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. 21

[15] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977. 3, 63

[16] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970. 52

[17] Helmuth Partsch and Frank A. Stomp. A fast pattern matching algorithm derived by transformational and assertional reasoning. *Formal Asp. Comput.*, 2(2):109–122, 1990. 58

[18] Christian Queinnec and Jean-Marie Geffroy. Partial evaluation applied to symbolic pattern matching with intelligent backtrack. In *Workshop in Static Analysis, number 81–82 in Bigre*, 1992. 4, 17, 64, 68

[19] Timo Raita. Tuning the boyer-moore-horspool string searching algorithm. *Softw., Pract. Exper.*, 22(10):879–884, 1992. 42

[20] Henning Korsholm Rohde. *Formal Aspects of Partial Evaluation*. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, December 2005.

[21] Henning Korsholm Rohde. Measuring the propagation of information in partial evaluation. Research Series RS-05-26, BRICS, Department of Computer Science, University of Aarhus, August 2005. 39 pp. 4, 5, 17, 18, 21, 23, 25, 26, 43, 68

[22] N Saitou and M Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Mol Biol Evol*, 4(4):406–425, 1987. 6, 54

[23] P. D. Smith. On tuning the boyer-moore-horspool string searching algorithm. *Softw. Pract. Exper.*, 24:435–436, April 1994. 28

[24] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996. 4, 13, 15, 16, 65

[25] Daniel M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33:132–142, August 1990. 9

Appendix A

Table separating all our string matchers

This table separates all of the string matchers we have added tracing to. This table consist of most of the handbook matchers, the known matchers defined in our framework, permutations of matchers generated using our framework and matchers implemented from the literature.

pattern	text	trace	matcher
abaa	ababbabaa	0, 1, 2, 3, 0, 4, 1, 5, 2, 6, 3, 7, 4, 8, 5, 6, 7, 8	cl_karp_rabin
abaa	ababbabaa	0, 1, 2, 3, 1, 2, 3, 4, 3, 4, 5, 6, 7, 8	fw_tbl_no-skip_l2r_0pos_neg fw_tbl_no-skip_l2r_0pos_2neg fw_tbl_no-skip_l2r_0pos_1neg fw_tbl_no-skip_l2r_0pos_0neg fw_tbl_skip_l2r_0pos_neg fw_tbl_skip_l2r_0pos_2neg fw_tbl_skip_l2r_0pos_1neg fw_tbl_skip_l2r_0pos_0neg fw_no-tbl_no-skip_l2r_0pos_0neg fw_no-tbl_skip_l2r_0pos_0neg consel-danvy-IPL89-naive-approach cl_naive fw_naive
abaa	ababbabaa	0, 1, 2, 3, 1, 2, 3, 4, 3, 4, 5, 6, 7, 8, 5	cl_galil_seiferas
abaa	ababbabaa	0, 1, 2, 3, 2, 3, 4, 3, 4, 5, 6, 7, 8	fw_no-tbl_no-skip_l2r_0pos_1neg fw_no-tbl_skip_l2r_0pos_1neg
abaa	ababbabaa	0, 1, 2, 3, 2, 3, 4, 4, 5, 6, 7, 8	fw_no-tbl_no-skip_l2r_pos_0neg fw_no-tbl_no-skip_l2r_2pos_0neg fw_no-tbl_no-skip_l2r_1pos_0neg
abaa	ababbabaa	0, 1, 2, 3, 3, 4, 4, 5, 6, 7, 8	fw_no-tbl_skip_l2r_pos_0neg fw_no-tbl_skip_l2r_2pos_0neg fw_no-tbl_skip_l2r_1pos_0neg soerensen-al-JFP96-fig-18-fixed ager-al-TOPLAS06-fig-3 ager-al-TOPLAS06-fig-1 cl_morris_pratt fw_mp
abaa	ababbabaa	0, 1, 2, 3, 4, 3, 6, 7, 5, 6, 7, 8	cl_smith fw_smith
abaa	ababbabaa	0, 1, 2, 3, 4, 3, 7, 4, 8, 5, 6, 7, 8	cl_quick_search

Continued on next page...

Table A.1 continued

pattern	text	trace	matcher
			fw_quick-search
abaa	ababbabaa	0, 1, 2, 3, 4, 5, 3, 7, 8, 4, 8, 5, 6, 7, 8	cl_berry_ravindran
abaa	ababbabaa	0, 1, 2, 3, 4, 5, 6, 7, 8	fw_tbl_skip_l2r_pos_neg fw_tbl_skip_l2r_pos_2neg fw_tbl_skip_l2r_pos_1neg fw_tbl_skip_l2r_pos_0neg fw_tbl_skip_l2r_2pos_neg fw_tbl_skip_l2r_2pos_2neg fw_tbl_skip_l2r_2pos_1neg fw_tbl_skip_l2r_2pos_0neg fw_tbl_skip_l2r_1pos_neg fw_tbl_skip_l2r_1pos_2neg fw_tbl_skip_l2r_1pos_1neg fw_tbl_skip_l2r_1pos_0neg cl_forward_dawg_matching cl_automaton fw_automaton
abaa	ababbabaa	1, 2, 3, 3, 4, 5, 6, 7, 8, 5	cl_not_so_naive fw_not-so-naive
abaa	ababbabaa	1, 2, 3, 3, 4, 6, 7, 8, 5	cl_apostolico_crochemore
abaa	ababbabaa	2, 1, 3, 4, 5, 4, 6, 7, 7, 6, 8, 5	cl_maximal_shift
abaa	ababbabaa	2, 3, 4, 5, 6, 7, 8, 6, 5	cl_two_way
abaa	ababbabaa	3, 2, 1, 5, 4, 3, 8, 7, 6	cl_turbo_reverse_factor
abaa	ababbabaa	3, 2, 3, 4, 7, 4, 5, 6, 7, 8	cl_skip_search
abaa	ababbabaa	3, 2, 3, 4, 7, 5, 6, 7, 8	cl_kmp_skip_search
abaa	ababbabaa	3, 2, 3, 5, 4, 4, 5, 6, 5, 6, 8, 7, 6, 5	cl_zhu_takaoka
abaa	ababbabaa	3, 3, 5, 4, 4, 6, 6, 8, 7, 6, 5	cl_turbo_bm
abaa	ababbabaa	3, 4, 5, 4, 6, 7, 6, 8, 6, 5	fw_no-tbl_skip_r2l_1pos_0neg
abaa	ababbabaa	3, 5, 2, 3, 4, 6, 8, 5, 6, 7	cl_horspool fw_horspool
abaa	ababbabaa	3, 5, 2, 4, 6, 8, 5, 7	cl_tuned_bm
abaa	ababbabaa	3, 5, 2, 4, 6, 8, 5, 7, 6	cl_raita fw_raita
abaa	ababbabaa	3, 5, 4, 6, 8, 7, 5	fw_tbl_skip_r2l_1pos_neg fw_tbl_skip_r2l_1pos_2neg fw_tbl_skip_r2l_1pos_1neg fw_tbl_skip_r2l_1pos_0neg
abaa	ababbabaa	3, 5, 4, 6, 8, 7, 5, 6	cl_reverse_colussi
abaa abaa	ababbabaa abacbabaa	0, 1, 2, 3, 2, 3, 4, 5, 6, 7, 8 0, 1, 2, 3, 2, 3, 3, 4, 5, 6, 7, 8	fw_no-tbl_no-skip_l2r_pos_1neg fw_no-tbl_no-skip_l2r_2pos_1neg fw_no-tbl_no-skip_l2r_1pos_1neg
abaa abaa	ababbabaa abacbabaa	0, 1, 2, 3, 2, 3, 4, 5, 6, 7, 8 0, 1, 2, 3, 4, 5, 6, 7, 8	fw_tbl_no-skip_l2r_pos_neg fw_tbl_no-skip_l2r_pos_2neg fw_tbl_no-skip_l2r_pos_1neg fw_tbl_no-skip_l2r_pos_0neg fw_tbl_no-skip_l2r_2pos_neg fw_tbl_no-skip_l2r_2pos_2neg fw_tbl_no-skip_l2r_2pos_1neg fw_tbl_no-skip_l2r_2pos_0neg fw_tbl_no-skip_l2r_1pos_neg fw_tbl_no-skip_l2r_1pos_2neg fw_tbl_no-skip_l2r_1pos_1neg fw_tbl_no-skip_l2r_1pos_0neg
abaa abaa	ababbabaa abacbabaa	0, 1, 2, 3, 3, 4, 5, 6, 7, 8 0, 1, 2, 3, 3, 4, 5, 6, 7, 8	fw_no-tbl_skip_l2r_pos_neg fw_no-tbl_skip_l2r_pos_2neg fw_no-tbl_skip_l2r_2pos_neg fw_no-tbl_skip_l2r_2pos_2neg fw_no-tbl_skip_l2r_1pos_neg fw_no-tbl_skip_l2r_1pos_2neg

Continued on next page...

Table A.1 continued

pattern	text	trace	matcher
			amtoft-al-Jones02-left-to-right
abaa	ababbabaa	1, 3, 3, 5, 4, 6, 8, 7	cl_galil_giancarlo
abaa	abcaabaa	1, 3, 2, 4, 5, 7, 6	
abaa	ababbabaa	1, 3, 3, 5, 4, 6, 8, 7	cl_colussi
abaa	abcaabaa	1, 3, 2, 4, 5, 7, 6, 4	
abaa	ababbabaa	3, 2, 1, 5, 4, 3, 8, 7, 6, 5, 4	cl_backward_oracle_matching
aabb	ababaaabb	3, 2, 1, 5, 4, 3, 7, 6, 5, 4, 8, 7, 6, 5, 4	
abaa	ababbabaa	3, 2, 1, 5, 4, 3, 8, 7, 6, 5, 4	cl_reverse_factor
aabb	ababaaabb	3, 2, 1, 7, 6, 5, 4, 8, 7, 6, 5, 4	
abaa	ababbabaa	3, 4, 5, 4, 6, 7, 6, 8, 7, 6, 5	fw_tbl_no-skip_r2l_0pos_neg
abaa	abcaabaa	3, 2, 4, 3, 2, 5, 6, 5, 7, 6, 5, 4	fw_tbl_no-skip_r2l_0pos_2neg
			fw_tbl_no-skip_r2l_0pos_1neg
			fw_tbl_no-skip_r2l_0pos_0neg
			fw_tbl_skip_r2l_0pos_neg
			fw_tbl_skip_r2l_0pos_2neg
			fw_tbl_skip_r2l_0pos_1neg
			fw_tbl_skip_r2l_0pos_0neg
			fw_no-tbl_no-skip_r2l_0pos_0neg
			fw_no-tbl_skip_r2l_0pos_0neg
			danvy-rohde-IPL06-sec-2
abaa	ababbabaa	3, 4, 5, 4, 6, 7, 6, 8, 7, 6, 5	fw_no-tbl_no-skip_r2l_1pos_0neg
abaa	abcaabaa	3, 2, 4, 3, 2, 7, 6, 5, 4	
abaa	ababbabaa	3, 4, 5, 4, 6, 8, 7, 6	fw_no-tbl_skip_r2l_2pos_0neg
abbb	abcbaabbb	3, 2, 4, 5, 6, 5, 7, 5, 8, 5	
abaa	ababbabaa	3, 4, 5, 4, 6, 8, 7, 6	fw_no-tbl_skip_r2l_pos_0neg
abbb	abcbaabbb	3, 2, 4, 5, 7, 6, 5, 8, 5	
abaa	ababbabaa	3, 4, 5, 4, 6, 8, 7, 6, 5	fw_no-tbl_no-skip_r2l_2pos_0neg
abbb	abcbaabbb	3, 2, 4, 5, 6, 5, 7, 6, 5, 8, 7, 6, 5	
abaa	ababbabaa	3, 4, 5, 4, 6, 8, 7, 6, 5	fw_no-tbl_no-skip_r2l_pos_0neg
abbb	abcbaabbb	3, 2, 4, 5, 7, 6, 5, 8, 7, 6, 5	
abaa	ababbabaa	3, 5, 4, 6, 8, 7, 6	fw_no-tbl_skip_r2l_pos_1neg
abaa	abacbabaa	3, 5, 4, 6, 8, 7, 6	fw_no-tbl_skip_r2l_2pos_1neg
abaa	ababbabaa	3, 5, 4, 6, 8, 7, 6	cl_apostolico_giancarlo
abaa	abacbabaa	3, 7, 6, 8, 6, 5	
abaa	ababbabaa	3, 5, 4, 6, 8, 7, 6, 5	fw_no-tbl_skip_r2l_1pos_1neg
bbaa	bbcaabbaa	3, 2, 4, 2, 8, 7, 6, 5	
abaa	ababbabaa	3, 5, 4, 6, 8, 7, 6, 5	danvy-rohde-IPL06-sec-3
bbaa	bbcaabbaa	3, 2, 4, 3, 2, 5, 7, 6, 8, 7, 6, 5	fw_horspool-right-to-left
abaa	ababbabaa	3, 5, 4, 6, 8, 7, 6, 5	fw_no-tbl_no-skip_r2l_0pos_1neg
bbaa	bbcaabbaa	3, 2, 4, 3, 2, 6, 8, 7, 6, 5	fw_no-tbl_skip_r2l_0pos_1neg
abaa	ababbabaa	3, 5, 4, 6, 8, 7, 6, 5	fw_tbl_no-skip_r2l_1pos_neg
bbaa	bbcaabbaa	3, 2, 7, 6, 8, 7, 6, 5	fw_tbl_no-skip_r2l_1pos_2neg
			fw_tbl_no-skip_r2l_1pos_1neg
			fw_tbl_no-skip_r2l_1pos_0neg
			fw_partsch-stomp
abaa	ababbabaa	3, 5, 4, 8, 7, 6	fw_no-tbl_skip_r2l_1pos_neg
aabb	aacbaaabb	3, 2, 4, 6, 8, 7, 6, 5	fw_no-tbl_skip_r2l_1pos_2neg
abaa	ababbabaa	3, 5, 4, 8, 7, 6	fw_no-tbl_skip_r2l_pos_neg
aabb	aacbaaabb	3, 2, 4, 7, 6, 8, 6, 5	fw_no-tbl_skip_r2l_pos_2neg
			fw_no-tbl_skip_r2l_2pos_neg
			fw_no-tbl_skip_r2l_2pos_2neg
			amtoft-al-Jones02-right-to-left
abaa	ababbabaa	3, 5, 4, 8, 7, 6, 5	fw_tbl_no-skip_r2l_pos_neg
abaa	abcaabaa	3, 2, 6, 5, 7, 6, 5, 4	fw_tbl_no-skip_r2l_pos_2neg
			fw_tbl_no-skip_r2l_pos_1neg
			fw_tbl_no-skip_r2l_pos_0neg
			fw_tbl_no-skip_r2l_2pos_neg
			fw_tbl_no-skip_r2l_2pos_2neg
			fw_tbl_no-skip_r2l_2pos_1neg

Continued on next page...

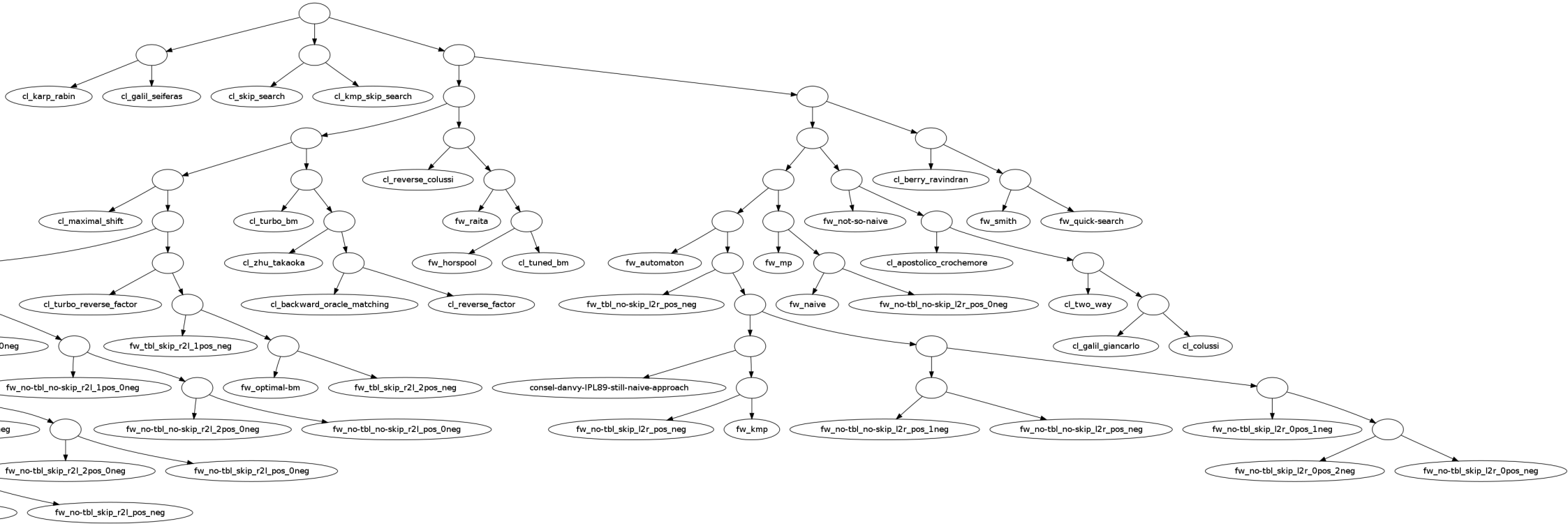
Table A.1 continued

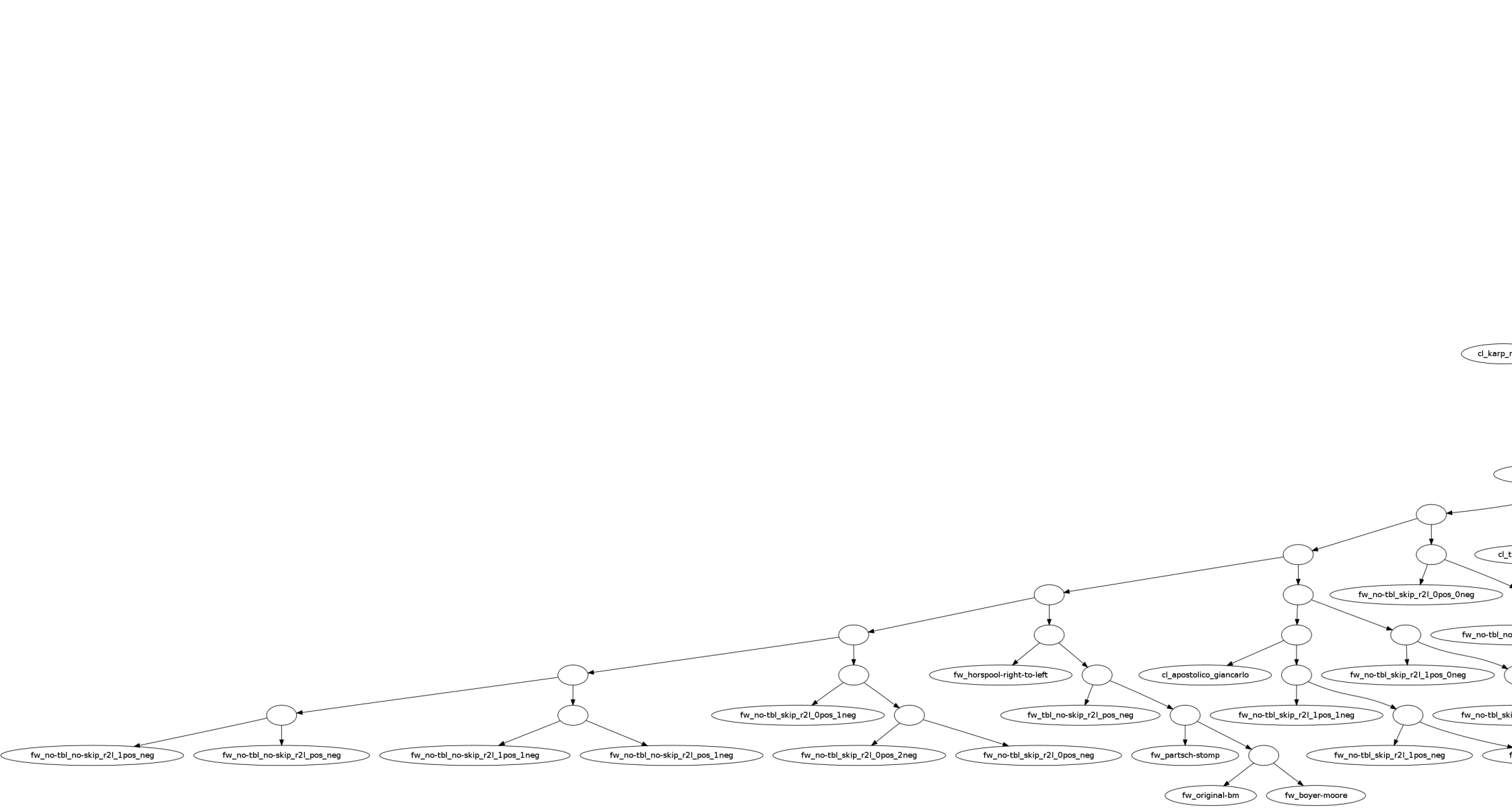
pattern	text	trace	matcher
			fw_tbl_no-skip_r2l_2pos_0neg
abaa	ababbabaa	0, 1, 2, 3, 2, 3, 4, 5, 6, 7, 8	fw_no-tbl_no-skip_l2r_0pos_2neg
abaa	abacbabaa	0, 1, 2, 3, 2, 3, 4, 5, 6, 7, 8	fw_no-tbl_skip_l2r_0pos_2neg
abba	aabbbabba	0, 1, 1, 2, 3, 4, 2, 3, 4, 5, 6, 7, 8	
abaa	ababbabaa	0, 1, 2, 3, 2, 3, 4, 5, 6, 7, 8	fw_no-tbl_no-skip_l2r_0pos_neg
abaa	abacbabaa	0, 1, 2, 3, 2, 3, 4, 5, 6, 7, 8	fw_no-tbl_skip_l2r_0pos_neg
abba	aabbbabba	0, 1, 1, 2, 3, 4, 2, 3, 5, 6, 7, 8	
abaa	ababbabaa	0, 1, 2, 3, 2, 3, 4, 5, 6, 7, 8	fw_no-tbl_no-skip_l2r_pos_neg
abaa	abacbabaa	0, 1, 2, 3, 2, 3, 4, 5, 6, 7, 8	fw_no-tbl_no-skip_l2r_pos_2neg
abba	aabbbabba	0, 1, 1, 2, 3, 4, 5, 6, 7, 8	fw_no-tbl_no-skip_l2r_2pos_neg
			fw_no-tbl_no-skip_l2r_2pos_2neg
			fw_no-tbl_no-skip_l2r_1pos_neg
			fw_no-tbl_no-skip_l2r_1pos_2neg
abaa	ababbabaa	0, 1, 2, 3, 3, 4, 5, 6, 7, 8	consel-danvy-IPL89-still-naive-approach
abaa	abacbabaa	0, 1, 2, 3, 3, 3, 4, 5, 6, 7, 8	
abab	acabaabab	0, 1, 1, 2, 3, 4, 5, 5, 5, 6, 7, 8	
abaa	ababbabaa	0, 1, 2, 3, 3, 4, 5, 6, 7, 8	fw_no-tbl_skip_l2r_pos_1neg
abaa	abacbabaa	0, 1, 2, 3, 3, 3, 4, 5, 6, 7, 8	fw_no-tbl_skip_l2r_2pos_1neg
abab	acabaabab	0, 1, 1, 2, 3, 4, 5, 5, 6, 7, 8	fw_no-tbl_skip_l2r_1pos_1neg
			consel-danvy-IPL89-further-optimization
			ager-al-TOPLAS06-fig-4
			ager-al-2002-ASIA-PEPM02-fig-6
			ager-al-2002-ASIA-PEPM02-fig-3
			cl_knuth_morris_pratt
			fw_kmp
abaa	ababbabaa	3, 5, 4, 6, 8, 7, 6, 5	fw_no-tbl_no-skip_r2l_1pos_1neg
bbaa	bbcaabbaa	3, 2, 4, 3, 2, 8, 7, 6, 5	
aabb	aacbaaabb	3, 2, 4, 6, 8, 7, 6, 5	
abaa	ababbabaa	3, 5, 4, 6, 8, 7, 6, 5	fw_no-tbl_no-skip_r2l_pos_1neg
bbaa	bbcaabbaa	3, 2, 4, 3, 2, 8, 7, 6, 5	fw_no-tbl_no-skip_r2l_2pos_1neg
aabb	aacbaaabb	3, 2, 4, 7, 6, 8, 7, 6, 5	
abaa	ababbabaa	3, 5, 4, 6, 8, 7, 6, 5	fw_original-bm
bbaa	bbcaabbaa	3, 2, 6, 8, 7, 6, 5	
abab	abcbaabab	3, 2, 6, 5, 4, 8, 7, 6, 5	
abaa	ababbabaa	3, 5, 4, 6, 8, 7, 6, 5	danvy-rohde-IPL06-sec-4
bbaa	bbcaabbaa	3, 2, 6, 8, 7, 6, 5	cl_boyer_moore
abab	abcbaabab	3, 2, 7, 8, 7, 6, 5	fw_boyer-moore
abaa	ababbabaa	3, 5, 4, 8, 7, 6	fw_tbl_skip_r2l_pos_neg
aabb	aacbaaabb	3, 2, 7, 6, 8, 5	fw_tbl_skip_r2l_pos_2neg
aaab	aabaaaaab	3, 4, 5, 6, 7, 8	fw_tbl_skip_r2l_pos_1neg
			fw_tbl_skip_r2l_pos_0neg
			fw_optimal-bm
abaa	ababbabaa	3, 5, 4, 8, 7, 6	fw_tbl_skip_r2l_2pos_neg
aabb	aacbaaabb	3, 2, 7, 6, 8, 5	fw_tbl_skip_r2l_2pos_2neg
aaab	aabaaaaab	3, 4, 5, 6, 7, 8, 5	fw_tbl_skip_r2l_2pos_1neg
			fw_tbl_skip_r2l_2pos_0neg
abaa	ababbabaa	3, 5, 4, 8, 7, 6, 5	fw_no-tbl_no-skip_r2l_0pos_2neg
abaa	abcaabaa	3, 2, 4, 3, 2, 6, 5, 7, 6, 5, 4	fw_no-tbl_skip_r2l_0pos_2neg
abba	aabbbabba	3, 4, 5, 4, 3, 2, 6, 7, 8, 7, 6, 5	
abaa	ababbabaa	3, 5, 4, 8, 7, 6, 5	fw_no-tbl_no-skip_r2l_0pos_neg
abaa	abcaabaa	3, 2, 4, 3, 2, 6, 5, 7, 6, 5, 4	fw_no-tbl_skip_r2l_0pos_neg
abba	aabbbabba	3, 4, 5, 4, 3, 2, 8, 7, 6, 5	
abaa	ababbabaa	3, 5, 4, 8, 7, 6, 5	fw_no-tbl_no-skip_r2l_1pos_neg
abaa	abcaabaa	3, 2, 4, 3, 2, 7, 6, 5, 4	fw_no-tbl_no-skip_r2l_1pos_2neg
aabb	aacbaaabb	3, 2, 4, 6, 8, 7, 6, 5	
abaa	ababbabaa	3, 5, 4, 8, 7, 6, 5	fw_no-tbl_no-skip_r2l_pos_neg
abaa	abcaabaa	3, 2, 4, 3, 2, 7, 6, 5, 4	fw_no-tbl_no-skip_r2l_pos_2neg
aabb	aacbaaabb	3, 2, 4, 7, 6, 8, 7, 6, 5	fw_no-tbl_no-skip_r2l_2pos_neg
			fw_no-tbl_no-skip_r2l_2pos_2neg

Appendix B

Evolutionary trees

B.1 Evolutionary tree of all our distinct string matchers

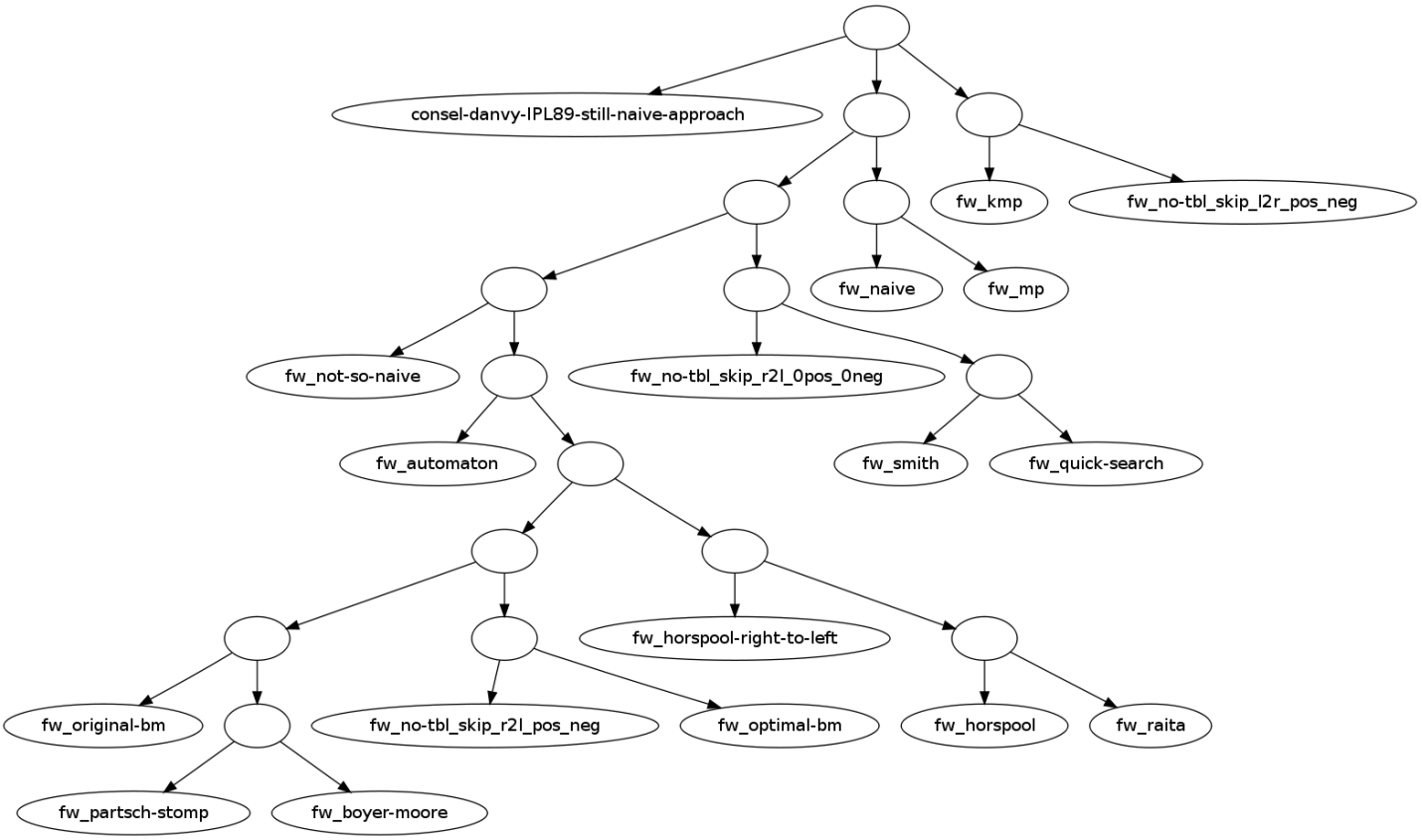

cl_karp_rabin
cl_galil_seiferas
cl_skip_search
cl_kmp_skip_search
cl_reverse_colussi
cl_berry_ravindran
cl_maximal_shift
cl_turbo_bm
fw_raita
fw_not-so-naive
fw_smith
fw_quick-search
cl_zhu_takaoka
fw_horspool
cl_tuned_bm
fw_automaton
fw_mp
cl_apostolico_crochemore
cl_turbo_reverse_factor
cl_backward_oracle_matching
cl_reverse_factor
fw_tbl_no-skip_l2r_pos_neg
fw_naive
fw_no-tbl_no-skip_l2r_pos_0neg
cl_two_way
0neg
fw_tbl_skip_r2l_1pos_neg
cl_galil_giancarlo
cl_colussi
fw_no-tbl_no-skip_r2l_1pos_0neg
fw_optimal-bm
fw_tbl_skip_r2l_2pos_neg
consel-danvy-IPL89-still-naive-approach
fw_no-tbl_no-skip_r2l_2pos_0neg
fw_no-tbl_no-skip_r2l_pos_0neg
fw_no-tbl_skip_l2r_pos_neg
fw_kmp
fw_no-tbl_no-skip_l2r_pos_1neg
fw_no-tbl_no-skip_l2r_pos_neg
fw_no-tbl_skip_l2r_0pos_1neg
fw_no-tbl_skip_r2l_2pos_0neg
fw_no-tbl_skip_r2l_pos_0neg
fw_no-tbl_skip_l2r_0pos_2neg
fw_no-tbl_skip_l2r_0pos_neg
fw_no-tbl_skip_r2l_pos_neg


fw_no-tbl_no-skip_r2l_1pos_neg
fw_no-tbl_no-skip_r2l_pos_neg
fw_no-tbl_no-skip_r2l_1pos_1neg
fw_no-tbl_no-skip_r2l_pos_1neg
fw_no-tbl_skip_r2l_0pos_1neg
fw_no-tbl_skip_r2l_0pos_2neg
fw_no-tbl_skip_r2l_0pos_neg
fw_horspool-right-to-left
fw_tbl_no-skip_r2l_pos_neg
fw_partsch-stomp
fw_original-bm
fw_boyer-moore
cl_apostolico_giancarlo
fw_no-tbl_skip_r2l_1pos_1neg
fw_no-tbl_skip_r2l_1pos_neg
fw_no-tbl_skip_r2l_0pos_0neg
fw_no-tbl_skip_r2l_1pos_0neg

B.2 Evolutionary tree with high gap and low difference cost

This evolutionary tree is built using the pairwise-alignment comparison method. The comparison method used a gap cost of 100 and a difference cost of 1.

v



Appendix C

Identification of string matchers from the literature

This table gives an overview over the results described in Section 4.4.

Matcher	Matchers equivalent with it
knuth-morris-pratt-SIAM70-abcabcacab	KMP Optimal-KMP†
consel-danvy-IPL89-naive-approach	Naive
consel-danvy-IPL89-still-naive	soerensen-al-JFP96-fig-4-aab*
consel-danvy-IPL89-still-naive-ababc	consel-danvy-IPL89-still-naive-approach
consel-danvy-IPL89-further-optimization	KMP
consel-danvy-IPL89-further-optimization-abcabcacab	consel-danvy-IPL89-further-optimization KMP Optimal KMP*
queinnec-geffroy-WSA92-babar	KMP† Optimal KMP
queinnec-geffroy-WSA92-foo	Optimal-KMP-right-to-left
sorensen-al-JFP96-fig-4-aab	KMP Optimal KMP†
sorensen-al-JFP96-fig-11-aab	Morris-Pratt
sorensen-al-JFP96-fig-18-fixed	Morris-Pratt
amtoft-al-Jones02-left-to-right	Optimal KMP
amtoft-al-Jones02-fig-2-aaa	amtoft-al-Jones02-left-to-right KMP* Optimal KMP Automaton*
amtoft-al-Jones02-right-to-left	Optimal-KMP-right-to-left
amtoft-al-Jones02-fig-3-abb	
amtoft-al-Jones02-fig-3-abb-prune-duplicates	Partsch-Stomp
ager-al-2002-ASIA-PEPM02-fig-3	KMP
ager-al-2002-ASIA-PEPM02-fig-6	KMP
ager-al-TOPLAS06-fig-1	Morris-Pratt
ager-al-TOPLAS06-fig-3	Morris-Pratt
ager-al-TOPLAS06-fig-4	KMP
danvy-rohde-IPL06-sec-2	Naive-right-to-left
danvy-rohde-IPL06-sec-3	horspool-right-to-left
danvy-rohde-IPL06-sec-3-aba	danvy-rohde-IPL06-sec-3 horspool-right-to-left Boyer-Moore* Partsch-Stomp* Original-BM*
danvy-rohde-IPL06-sec-4	Boyer-Moore

* We know from other results in the table that these matchers were not intended to be trace equivalent with their left-column counter-part.

† We know from the papers that these matchers were not intended to be trace equivalent with their left-column counter-part.

Figure C.1: Summary over matchers from the literature

Index