A taxonomy of approximate pattern matching algorithms in strings

Bosman, R.P.

Award date:
2005

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER'S THESIS

A Taxonomy of Approximate Pattern Matching Algorithms in strings

by
R.P. Bosman

Supervisors: prof. dr. B.W. Watson
dr. ir. G. Zwaan

Advisor: ir. L.G.W.A. Cleophas

Eindhoven, March 2005

Abstract

This thesis contains a taxonomy of algorithms in the field of Approximate Pattern Matching in strings.

The Approximate Pattern Matching problem is to find all substrings of a text which have no more than a maximum allowed number of differences with the specified pattern or patterns (basically, text searching algorithms which also find occurrences that differ slightly from the search string). Applications of approximate pattern matching algorithms lie within fields such as text searching, querying and dna analysis.

A taxonomy is an orderly classification of algorithms. This classification takes the form of a tree (or a directed acyclic graph in general), where all elements in the same (sub-)tree are somehow related to each other either because they solve the same problem or because they use the same strategy or datatype to solve the problem.

Taxonomies are created to have a uniform presentation of algorithms, an overview of structural relationships and algorithms in the field, to increase understanding of algorithms and to aid in the construction of new algorithms

Contents

List of Figures

Preface

This document is my master's thesis which I wrote to finish my study of Technische Informatica (Technical Computer Science) at the Eindhoven University of Technology (TU/e). My research was supervised by Prof. Dr. Bruce W. Watson and Dr. Gerard Zwaan within the Software Construction (SoC) group as part of the TABASCO (TAxonomy BAsed Software COnstruction) effort. The goal of the TABASCO effort is to order the algorithms in a domain by constructing taxonomies.

Part of the Software Construction group is closely associated with the FASTAR (Finite Automata Systems — Theoretical and Applied Research) group. The FASTAR group is an international research group with a strong focus on matters related to finite state systems. One of the main projects of the FASTAR group is the SPARE Time / SPARE Parts toolkit; a toolkit (based on a taxonomy) of exact keyword pattern matching algorithms. This thesis is to form the starting point of an effort to extend the SPARE Time / SPARE Parts project with approximate pattern matching.

I would like to thank Bruce Watson for enabling me to perform this research, getting me involved in the FASTAR group and supplying a lot of relevant material on the subjects involved.

I thank Gerard Zwaan for all the thorough reviews of this document and for providing a lot of advice on the problems at hand. Gerard Zwaan is known within our group for his attention to details, and rightfully so. Gerard Zwaan managed to detect almost all the errors I made in my proofs and prevented this text from getting needlessly complex.

Further thanks go to Loek Cleophas for both providing a lot of useful insight from his knowledge on exact keyword pattern matching and for his aid in practical aspects such as advanced LaTeX macros.

Finally, I would like to thank my friends and family for their support. In particular, I thank Martijn van de Rijdt for our brainstorming sessions and the interesting discussions on a large variety of subjects which we have had over mail and in person, and my mother for her patience in listening to me each time I wanted to order my thoughts by explaining my ideas to non computer scientists.

Remi Bosman,
Eindhoven, March 2005

Chapter 1

Introduction

1.1 Thesis structure

This text consists of four chapters and two appendices. Chapter 1 is an introduction to the problem domain and taxonomies.

Chapter 2 contains the actual taxonomy. The taxonomy starts with an overview of all the Approximate Pattern Matching algorithms in this text, ordered according to their relationships. Each algorithm is derived by transforming a previously derived algorithm into a new one and by adding more and more details along the way. The starting point is the problem specification.

Chapter 3 contains several formal proofs of properties which are used throughout this text. Most automata and recurrence relations which are described in the taxonomy are formally defined in this chapter along with proofs of some of their properties. Although the properties themselves are used in the derivation of the algorithms, the text can be read without reading these additional proofs.

Chapter 4 contains the conclusions and directions for future work. Appendix A contains several definitions and notations used in this text. Appendix B contains information on how bit parallel algorithms are derived and presented in this text.

1.2 Problem Description

This thesis gives an overview of algorithms in the field of Approximate Pattern Matching in Strings. Informally, the Approximate Pattern Matching problem is to find all substrings of a text for which the distance (defined by a distance function) between the substring of the text and at least one of the input patterns is bounded by some given value. More formally, given alphabet Σ, text $T \in \Sigma^*$, finite non-empty pattern set $P = \{p_0, p_1, \ldots, p_{|P|-1}\}$ $(p_i \in \Sigma^*(\forall i))$, distance function $dst : \Sigma^* \times \Sigma^* \to \mathbb{N}$ and maximum distance k $(0 \leq k)$, return

$$\langle \textbf{set } l, v, r, p \ : \ lvr = T \ \wedge \ p \in P \ \wedge \ dst(v, p) \leq k \ : \ (l, v, r, dst(v, p)) \rangle \tag{1.1}$$

Some remarks about notation: On several occasions the length of the patterns and text is used in algorithms. To prevent having to write $|p_0|, |p_1|, \ldots, |p_{|P|-1}|$ and $|T|$ all the time, the

letters $m_0, m_1, \ldots, m_{|P|-1}$ and n will be used.

$$\langle \forall i \ : \ 0 \leq i < |P| \ : \ m_i = |p_i| \rangle \tag{1.2}$$

$$n = |T| \tag{1.3}$$

Whenever algorithms are restricted to a single pattern (i.e.: $P = \{p_0\}$), the input of the algorithm will be the pattern p instead of the singleton set $P = \{p_0\}$. The letter m will be used for the length of pattern p (i.e. $m = |p|$).

1.2.1 The edit distance

A distance function which is often used in approximate pattern matching algorithms is the edit distance (or Levenshtein distance). The edit distance is defined as the minimum number of character insertions, deletions or substitutions required to make two strings equal. Formally, the edit distance between two strings is defined as follows: $(a \in \Sigma, b \in \Sigma, x \in \Sigma^*, y \in \Sigma^*)$

$$\begin{aligned}
&dst(\varepsilon, y) \leftarrow |y| \\
&dst(x, \varepsilon) \leftarrow |x| \\
&dst(ax, by) \leftarrow (dst(x, y) + \delta(a, b)) \downarrow (dst(x, by) + 1) \downarrow (dst(ax, y) + 1) \\
&dst(xa, yb) \leftarrow (dst(x, y) + \delta(a, b)) \downarrow (dst(x, yb) + 1) \downarrow (dst(xa, y) + 1)
\end{aligned}$$

Here $\delta(a, b) = 0$ if $a = b$ and 1 otherwise.

The last two lines overlap. This is still a valid definition since both expressions yield the same value. Since the edit distance is the minimum number of edit operation required to make strings equal, it does not matter if one starts with the symbols at the beginning of the word or if one starts at the end of the word.

Remark: In practice the maximum distance k is often restricted to the range $0 < k < m_i$ (where m_i is the length of he shortest pattern). When the maximum distance equals zero, the problem is equal to the exact pattern matching problem. When the maximum distance equals or exceeds the length of a pattern then a lot of uninteresting matches will be produced when using the edit distance as defined above. For example: when $m_i \leq k$ the empty string matches with the pattern (using m_i insertions) and hence a match is found at each text position.

In this text, several properties of the edit distance will be used. See section 3.1 on page 71 for formal definitions of these properties and detailed proofs.

- Reversing the argument strings of the distance function does not change the distance ($dst(x, y) = dst(x^R, y^R)$).
- Adding a single character in front of the first argument of the distance function changes the distance between the first argument and the second argument by at most one ($-1 \leq dst(ax, y) - dst(x, y) \leq 1$)
- If there exists an occurrence with less than k differences then there exist suffixes of that occurrence with up to k differences as well. Formally:

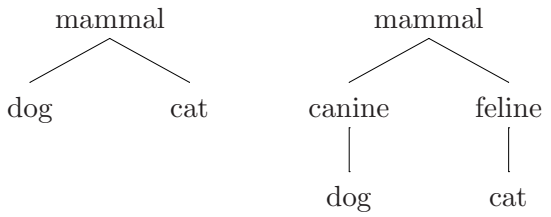
$$(\langle \forall j \ : \ dst(v, p) \leq j \leq k \ : \ \langle \exists s \ : \ s \leq_s v \ : \ dst(s, p) = j \rangle \rangle)$$

1.3 A short introduction to taxonomies

Before describing the taxonomy of Approximate Pattern Matching algorithms, first take a look at what exactly a taxonomy is.

> A taxonomy is an orderly classification of elements in a domain according to their presumed relationships.

Taxonomies have long been used in the field of biology to classify plants and animals. Such a taxonomy divides the plants or animals into different families. Consider the following two taxonomies of cats and dogs.



This example shows some interesting aspects of taxonomies. A taxonomy takes the form of a directed acyclic graph. Each node of this graph represents a family or subfamily of animals based on some characteristic of the object under consideration. In this case, both dogs and cats are mammals because they both give birth to live young.

A taxonomy is not unique. The taxonomy on the right adds the additional branches canine and feline to the taxonomy and as such contains more information than the taxonomy on the left. This shows that taxonomies can vary in detail. Since taxonomies differ in the details they highlight, it might be possible for certain taxonomies to miss some relationships and/or to structure things differently. Extending a taxonomy with additional objects often brings forward such discrepancies and sometimes requires some restructuring.

Why should anyone go through all the effort required to construct a taxonomy?

The answer is of course that taxonomies have several important advantages:

- Taxonomies increase the understanding of a particular domain. They highlight the important variations and commonalities between objects and clearly shows how objects are related.
- Taxonomies provide a uniform description of objects, which makes comparison much easier.
- Taxonomies are an excellent access point into the domain. Since they function as an overview of existing knowledge within the domain. This makes it possible to quickly lookup the required information about specific objects.
- When one encounters an object for which one is not certain if it is new or already discovered, a taxonomy can be used to locate the family of the object by comparing properties (note that the relevant properties are identified by the taxonomy).

- Taxonomies can guide the direction of new research. Because a taxonomy identifies important properties and choices, they sometimes reveal choices which have not yet been explored.
- In domains such as programming or production the close similarity between objects in a family can be used to optimize the library structure or production process.

Software taxonomies are based on exactly the same ideas as taxonomies in biology. The properties on associated with each node in software taxonomies are typically: problem details (changes to pre- and postconditions of the algorithm) and algorithm details (strategies used to establish the postcondition of the algorithms, data structures used etc.).

Taxonomies can be presented in many different styles, for a somewhat different style from the presentation used in this text refer to the taxonomies on exact keyword pattern matching algorithms by Watson [Wat95] or by Cleophas [Cle03].

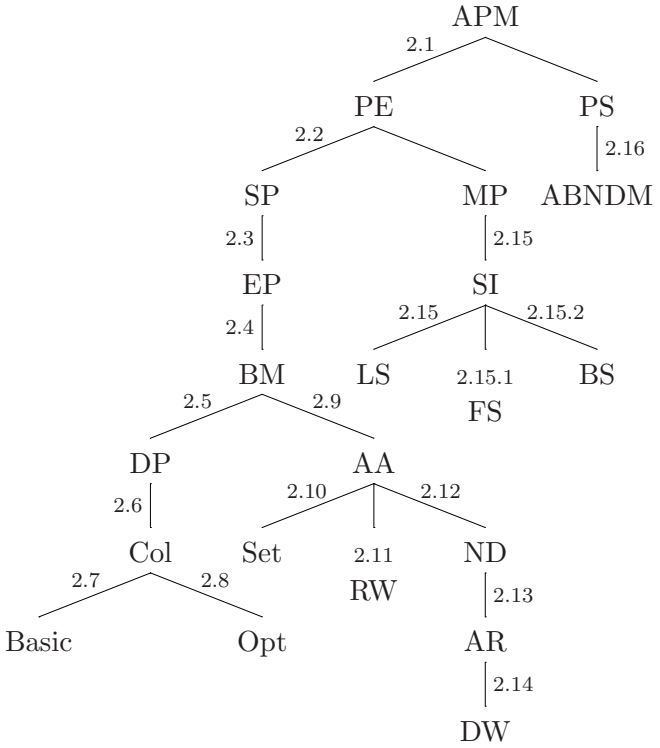
Chapter 2

Taxonomy

The approximate pattern matching problem is specified by the following postcondition:

$$O = \langle \textbf{set}\ l, v, r, p\ :\ lvr = T\ \wedge\ p \in P\ \wedge\ dst(v,p) \leq k\ :\ (l, v, r, p, dst(v,p)) \rangle \qquad (R)$$

The picture below displays the taxonomy tree. Each node represents an algorithm and each edge is labeled with the section which describes how the parent node is transformed into the child node.



The taxonomy by Watson in [Wat95] labels the nodes with sections and the edges with problem details. By labeling the edges with sections rather than the nodes, a stronger emphasis is placed on the transition of one algorithm to the other and on the other hand less emphasis is

placed on the specific details which are introduced along the way. This approach is easier to use when the taxonomy is more course grained. The style used by Watson tends to focus on the differences between algorithms, the style used in this text tends to focus on similarities between algorithms.

The root of the taxonomy is the trivial algorithm (APM):

Algorithm 2.0.1(APM)

$O := \langle \textbf{set}\ l, v, r, p\ :\ lvr = T\ \wedge\ p \in P\ \wedge\ dst(v,p) \leq k\ :\ (l, v, r, p, dst(v,p)) \rangle;$
$\{\ R\ \}$

The nodes in the tree are associated with the following problem and algorithm details.

- APM: The (abstract) ancestor algorithm of all approximate pattern matching algorithms.
- PE: Algorithms which gather all occurrences ending within a prefix of the text.
- PS: Algorithms which gather all occurrences starting within a prefix of the text.
- SP: A single pattern instead of multiple patterns.
- MP: Multiple pattern algorithms.
- EP: Report only end positions of occurrences.
- BM: Compute only the best matching occurrence and use that to report the others as well.
- DP: Find occurrences by computing a matrix using dynamic programming.
- Col: Compute dynamic programming matrix column by column.
- Basic: Compute entire columns of the dynamic programming matrix.
- Opt: Compute only relevant entries of the dynamic programming matrix.
- AA: Find occurrences by simulating a nondeterministic finite automaton.
- Set: Use generic automaton simulation techniques to simulate the automaton.
- RW: Row Wise bit parallel simulation of the automaton.
- ND: Do not report difference count of occurrences.
- AR: Restrict the automaton (only simulate full diagonals).
- DW: Diagonal wise bit parallel simulation of the automaton.
- SI: Find occurrences by simulating a superimposed automaton

- LS: Linear verification of occurrences found by the superimposed automaton.
- FS: Forward (hierarchical) verification of occurrences found by the superimposed automaton.
- BS: Backwards (hierarchical) verification of occurrences found by the superimposed automaton.
- ABNDM: A derivation of the ABNDM algorithm.

2.1 Prefix traversal

The first detail introduced in solving the approximate pattern matching problem is to consider prefixes u of increasing length of text T and gather all occurrences which end within prefix u in set O.

This is formalized by invariant P_0:

$$ur = T \wedge$$
$$O = \langle \textbf{set } x, y, z, p \ : \ xyz = T \ \wedge \ p \in P \ \wedge \ xy \leq_p u \ \wedge \ dst(y, p) \leq k \ : \ (x, y, z, p, dst(y, p)) \rangle$$

When the suffix r of T is empty then prefix u equals T and the conjunct $xy \leq_p u$ becomes trivially true since $xyz = T$. In other words $P_0 \ \wedge \ r = \varepsilon$ imply R.

The algorithm considers prefixes of increasing length, so initially prefix u should be empty.

$$\begin{array}{ll} & P_0(u, r := \varepsilon, T) \\ \equiv & \quad \{ \text{ subst } \} \\ & \varepsilon T = T \wedge \\ & O = \langle \textbf{set } x, y, z, p \ : \ xyz = T \ \wedge \ p \in P \ \wedge \ xy \leq_p \varepsilon \ \wedge \ dst(y, p) \leq k \ : \ (x, y, z, p, dst(y, p)) \rangle \\ \equiv & \quad \{ \ xy \leq_p \varepsilon \equiv (x = \varepsilon \ \wedge \ y = \varepsilon) \ \} \\ & O = \langle \textbf{set } p \ : \ p \in P \ \wedge \ dst(\varepsilon, p) \leq k \ : \ (\varepsilon, \varepsilon, T, p, dst(\varepsilon, p)) \rangle \\ \equiv & \quad \{ \text{ dst } \} \\ & O = \langle \textbf{set } p \ : \ p \in P \ \wedge \ |p| \leq k \ : \ (\varepsilon, \varepsilon, T, p, |p|) \rangle \end{array}$$

As long as r is not empty (i.e. $r :: cw$) the prefix must be extended by a single character at a time. Therefore consider the assignment $u, r := uc, w$ this clearly maintains $ur = T$ therefore consider the change to set O.

$$\begin{array}{ll} & \langle \textbf{set } x, y, z, p \ : \ xyz = T \ \wedge \ p \in P \ \wedge \ xy \leq_p u \ \wedge \ dst(y, p) \leq k \ : \ (x, y, z, p, dst(y, p)) \rangle \\ & (u, r := uc, w) \\ = & \quad \{ \text{ subst } \} \\ & \langle \textbf{set } x, y, z, p \ : \ xyz = T \ \wedge \ p \in P \ \wedge \ xy \leq_p uc \ \wedge \ dst(y, p) \leq k \ : \ (x, y, z, p, dst(y, p)) \rangle \\ = & \quad \{ \ P0, \text{ split off } xy = uc \ (xy \leq_p uc \equiv xy \leq_p u \ \vee \ xy = uc) \ \} \\ & O \cup \langle \textbf{set } x, y, z, p \ : \ xyz = T \ \wedge \ p \in P \ \wedge \ xy = uc \ \wedge \ dst(y, p) \leq k \ : \ (x, y, z, p, dst(y, p)) \rangle \end{array}$$

$$
\begin{array}{ll}
= & \{\ T = ur,\ r :: cw, \text{ therefore } z = w\ \} \\
& O \cup \langle \textbf{set}\ x, y, p\ :\ p \in P\ \wedge\ xy = uc\ \wedge\ dst(y,p) \leq k\ :\ (x, y, w, p, dst(y,p))\rangle
\end{array}
$$

So when prefix u is extended to uc all matches which are suffixes of uc are added to O. Combining this with the initialization and loop guard $r :: cw$ gives the following program:

Algorithm 2.1.2(PE)

$$
\begin{array}{l}
R : O = \langle \textbf{set}\ l, v, r, p\ :\ lvr = T\ \wedge\ p \in P\ \wedge\ dst(v,p) \leq k\ :\ (l, v, r, p, dst(v,p))\rangle \\
P_0 : ur = T\ \wedge \\
\quad O = \langle \textbf{set}\ x, y, z, p\ :\ xyz = T\ \wedge\ p \in P\ \wedge\ xy \leq_p u\ \wedge\ dst(y,p) \leq k\ :\ (x, y, z, p, dst(y,p))\rangle
\end{array}
$$

$$
\begin{array}{l}
O := \langle \textbf{set}\ p\ :\ p \in P\ \wedge\ |p| \leq k\ :\ (\varepsilon, \varepsilon, T, p, |p|)\rangle; \\
\{\ P_0(u, r := \varepsilon, T)\ \} \\
u, r := \varepsilon, T; \\
\textbf{do}\ r :: cw \rightarrow \{\ P_0\ \} \\
\quad O := O \cup \langle \textbf{set}\ x, y, p\ :\ p \in P\ \wedge\ xy = uc\ \wedge\ dst(y,p) \leq k\ :\ (x, y, w, p, dst(y,p))\rangle; \\
\quad \{\ P_0(u, r := uc, w)\ \} \\
\quad u, r := uc, w \\
\textbf{od}\{\ P_0\ \wedge\ r = \varepsilon\ \} \\
\{\ R\ \}
\end{array}
$$

2.2 Single Pattern

This section continues from the previous algorithm by restricting the pattern set P in the original specification to a single pattern (i.e. $P = \{p\}$).

This requires a small modification to the existing formulas. Note that since they make no assumption about P all previous derivations remain valid. The new formulas are acquired by substituting P by $\{p\}$ and applying the one point rule. The first formula is shown in detail the rest follow the same pattern.

- The new initialization:

$$
\begin{array}{ll}
& \langle \textbf{set}\ p'\ :\ p' \in P\ \wedge\ |p'| \leq k\ :\ (\varepsilon, \varepsilon, T, p', |p'|)\rangle(P := \{p\}) \\
= & \quad \{\ \text{subst, one point rule}\ \} \\
& \begin{cases} \{(\varepsilon, \varepsilon, T, p, |p|)\} & \text{if } |p| \leq k \\ \varnothing & \text{if } k < |p| \end{cases} \\
= & \quad \{\ |p| = m\ \} \\
& \begin{cases} \{(\varepsilon, \varepsilon, T, p, m)\} & \text{if } m \leq k \\ \varnothing & \text{if } k < m \end{cases}
\end{array}
$$

- The update of O:
  $O \cup \langle \textbf{set}\ x, y\ :\ xy = uc\ \wedge\ dst(y,p) \leq k\ :\ (x, y, w, p, dst(y,p))\rangle$

- The new invariant P_0:
  $ur = T \wedge O = \langle \textbf{set}\ x, y, z\ :\ xyz = T \wedge xy \leq_p u \wedge dst(y, p) \leq k\ :\ (x, y, z, p, dst(y, p))\rangle$
- The new postcondition R:
  $O = \langle \textbf{set}\ l, v, r\ :\ lvr = T\ \wedge\ dst(v, p) \leq k\ :\ (l, v, r, p, dst(v, p))\rangle$

In most single keyword pattern matching problems it is not very useful to report the pattern of which an occurrence has been found since it is known in advance. Note that the pattern p is a constant in the formulas above. In the rest of this text pattern p will be removed from the result. If in practice the pattern p is required it can be added in an implementation without violating correctness of the algorithms presented in this text (compare the program and formulas below with the formulas presented above).

Algorithm 2.2.3(PE, SP)

$R : O = \langle \textbf{set}\ l, v, r\ :\ lvr = T\ \wedge\ dst(v, p) \leq k\ :\ (l, v, r, dst(v, p))\rangle$
$P_0 : ur = T\ \wedge\ O = \langle \textbf{set}\ x, y, z\ :\ xyz = T\ \wedge\ xy \leq_p u\ \wedge\ dst(y, p) \leq k\ :\ (x, y, z, dst(y, p))\rangle$

if $m \leq k \rightarrow O := \{(\varepsilon, \varepsilon, T, m)\}$
$[\!]\ k < m \rightarrow O := \varnothing$
fi; $\{\ P_0(u, r := \varepsilon, T)\ \}$
$u, r := \varepsilon, T$;
do $r :: cw \rightarrow \{\ P_0\ \}$
  $O := O \cup \langle \textbf{set}\ x, y\ :\ xy = uc\ \wedge\ dst(y, p) \leq k\ :\ (x, y, w, dst(y, p))\rangle$;
  $\{\ P_0(u, r := uc, w)\ \}$
  $u, r := uc, w$
od$\{\ P_0\ \wedge\ r = \varepsilon\ \}$
$\{\ R\ \}$

2.3 End Positions

The postcondition of the algorithm in Section 2.2 can be further restricted by dropping the requirement to report both the begin and the end of an occurrence. By not reporting the start position the length of a match is lost, and hence it is not possible to report exactly which substring produced a match (without some post processing). Just like with the single keyword restriction all invariants remain valid.

Algorithm 2.3.4(PS, SP, EP)

$R : O = \langle \textbf{set}\ l, v, r\ :\ lvr = T\ \wedge\ dst(v, p) \leq k\ :\ (lv, r, dst(v, p))\rangle$
$P_0 : ur = T\ \wedge\ O = \langle \textbf{set}\ x, y, z\ :\ xyz = T\ \wedge\ xy \leq_p u\ \wedge\ dst(y, p) \leq k\ :\ (xy, z, dst(y, p))\rangle$

if $m \leq k \rightarrow O := \{(\varepsilon, T, m)\}$
$[\!]\ k < m \rightarrow O := \varnothing$

fi; $\{\ P_0(u, r := \varepsilon, T)\ \}$
$u, r := \varepsilon, T$;
do $r :: cw \rightarrow \{\ P_0\ \}$
$\quad O := O \cup \langle \textbf{set}\ x, y\ :\ xy = uc\ \wedge\ dst(y,p) \leq k\ :\ (xy, w, dst(y,p))\rangle$;
$\quad \{\ P_0(u, r := uc, w)\ \}$
$\quad u, r := uc, w$
od$\{\ P_0\ \wedge\ r = \varepsilon\ \}$
$\{\ R\ \}$

2.4 Best Match

By dropping the requirement to report the begin position of an occurrence, the triples added at a specific step of the repetition all become of the form (xy, w, i) where xy and w are fixed (within a specific cycle of the loop) and i is the number of differences ($0 \leq i \leq k$). Due to the form of this triple, the update of set O can be rewritten as follows:

$$
\begin{array}{ll}
 & \langle \textbf{set}\ x, y\ :\ xy = uc\ \wedge\ dst(y,p) \leq k\ :\ (xy, w, dst(y,p))\rangle \\
= & \quad \{\ 0 \leq dst(y,p)\ \} \\
 & \langle \textbf{set}\ x, y\ :\ xy = uc\ \wedge\ 0 \leq dst(y,p) \leq k\ :\ (xy, w, dst(y,p))\rangle \\
= & \quad \{\ \text{introduce } i\ \} \\
 & \langle \textbf{set}\ x, y, i\ :\ xy = uc\ \wedge\ dst(y,p) = i\ \wedge\ 0 \leq i \leq k\ :\ (xy, w, i))\rangle \\
= & \quad \{\ xy = uc\ \} \\
 & \langle \textbf{set}\ x, y, i\ :\ xy = uc\ \wedge\ dst(y,p) = i\ \wedge\ 0 \leq i \leq k\ :\ (uc, w, i)\rangle \\
= & \quad \{\ \text{nesting}\ \} \\
 & \langle \textbf{set}\ i\ :\ 0 \leq i \leq k\ \wedge\ \langle \exists x, y\ :\ xy = uc\ :\ dst(y,p) = i\rangle\ :\ (uc, w, i)\rangle
\end{array}
$$

This is an interesting variation, because now a simple repetition can add all tuples (uc, w, i) if the validity of $\langle \exists x, y\ :\ xy = uc\ :\ dst(y,p) = i\rangle$ is known. It will turn out that the validity of this quantification can be computed such that it is known for any value of i.

This variation shows some other interesting possibilities:

- If the actual number of differences is not relevant, this expression can be optimized to an $\mathcal{O}(1)$ update ($O := O \cup \{(uc, w)\}$).
- If it is not important to report all occurrences but rather just the best matching occurrences, this expression can easily be transformed into the tuple:

  $$(uc, w, \langle \downarrow i\ :\ 0 \leq i \leq k\ \wedge\ \langle \exists x, y\ :\ xy = uc\ :\ dst(y,p) = i\rangle\ :\ i\rangle)$$

  This change can be performed independent of the way in which the existence will be computed in the remainder of this text.

Algorithm 2.4.5(PE, SP, EP, BM)

$R : O = \langle \textbf{set}\ l, v, r\ :\ lvr = T\ \wedge\ dst(v,p) \leq k\ :\ (lv, r, dst(v,p))\rangle$
$P_0 : ur = T\ \wedge\ O = \langle \textbf{set}\ x, y, z\ :\ xyz = T\ \wedge\ xy \leq_p u\ \wedge\ dst(y,p) \leq k\ :\ (xy, z, dst(y,p))\rangle$

if $m \leq k \rightarrow O := \{(\varepsilon, T, m)\}$
$[\!]\ \ k < m \rightarrow O := \varnothing$
fi; $\{\ P_0(u, r := \varepsilon, T)\ \}$
$u, r := \varepsilon, T$;
do $r :: cw \rightarrow \{\ P_0\ \}$
$\quad O := O \cup \langle \textbf{set}\ i\ :\ 0 \leq i \leq k\ \wedge\ \langle \exists x, y\ :\ xy = uc\ :\ dst(y,p) = i\rangle\ :\ (uc, w, i)\rangle$
$\quad \{\ P_0(u, r := uc, w)\ \}$
$\quad u, r := uc, w$
od$\{\ P_0\ \wedge\ r = \varepsilon\ \}$
$\{\ R\ \}$

At this point there are several separate ways to continue:

- compute the distance with a matrix (basic and optimized dynamic programming algorithms)
- compute the distance with an automaton (set based, row-wise, diagonal wise simulation of an NFA)

2.5 Dynamic Programming

The Dynamic Programming algorithms in this section search approximate pattern occurrences by computing the edit distance between the pattern p and substrings of the input text T. Refer to Section 1.2.1 for the definition of the edit distance.

The edit distance can be used to search for approximate occurrences in a text by computing the minimal edit distance between a prefix of p and a suffix of a prefix (i.e. a substring or factor) of T. Formally: (for $x \leq_p p$ and $y \leq_p T$)

$$M_{x,y} = \langle \downarrow r\ :\ r \leq_s y\ :\ dst(r, x)\rangle$$

This formula specifies that cell (x, y) of matrix M is equal to the number of mismatches between the best matching suffix of y and prefix x of p. The following derivation shows how this matrix can be used to update set O in the repetition.

$\quad \langle \textbf{set}\ i\ :\ 0 \leq i \leq k\ \wedge\ \langle \exists x, y\ :\ xy = uc\ :\ dst(y,p) = i\rangle\ :\ (uc, w, i)\rangle$
$=$ $\quad \{$ Property 3.1.4 $\}$
$\quad \langle \textbf{set}\ i\ :\ 0 \leq i \leq k\ \wedge\ \langle \downarrow x, y\ :\ xy = uc\ :\ dst(y,p)\rangle \leq i\ :\ (uc, w, i)\rangle$
$=$ $\quad \{\ 0 \leq dst(y,p)$, calc $\}$

$$
\begin{array}{ll}
 & \langle \textbf{set } i \ : \ \langle \downarrow x, y \ : \ xy = uc \ : \ dst(y,p) \rangle \leq i \leq k \ : \ (uc, w, i) \rangle \\
= & \quad \{ \text{ definition } \leq_s \ \} \\
 & \langle \textbf{set } i \ : \ \langle \downarrow y \ : \ y \leq_s uc \ : \ dst(y,p) \rangle \leq i \leq k \ : \ (uc, w, i) \rangle \\
= & \quad \{ \ uc \leq_p T \ \wedge \ p \leq_p p, \text{ def } M \ \} \\
 & \langle \textbf{set } i \ : \ M_{p,uc} \leq i \leq k \ : \ (uc, w, i) \rangle
\end{array}
$$

Algorithm 2.5.6(PE, SP, EP, BM, DP)

$R : O = \langle \textbf{set } l, v, r \ : \ lvr = T \ \wedge \ dst(v,p) \leq k \ : \ (lv, r, dst(v,p)) \rangle$
$P_0 : ur = T \ \wedge \ O = \langle \textbf{set } x, y, z \ : \ xyz = T \ \wedge \ xy \leq_p u \ \wedge \ dst(y,p) \leq k \ : \ (xy, z, dst(y,p)) \rangle$

if $m \leq k \rightarrow O := \{(\varepsilon, T, m)\}$
$[\!]$ $k < m \rightarrow O := \varnothing$
fi; $\{ \ P_0(u, r := \varepsilon, T) \ \}$
$u, r := \varepsilon, T;$
do $r :: cw \rightarrow \{ \ P_0 \ \}$
  $O := O \cup \langle \textbf{set } i \ : \ M_{p,uc} \leq i \leq k \ : \ (uc, w, i) \rangle$
  $\{ \ P_0(u, r := uc, w) \ \}$
  $u, r := uc, w$
od$\{ \ P_0 \ \wedge \ r = \varepsilon \ \}$
$\{ \ R \ \}$

Matrix M is computed by the following recurrence relation:
$M_{\varepsilon,y} \leftarrow 0$
$M_{x,\varepsilon} \leftarrow |x|$
$M_{xa,yb} \leftarrow (M_{x,y} + \delta(a,b)) \downarrow (M_{x,yb} + 1) \downarrow (M_{xa,y} + 1) \quad (xa \leq_p p, yb \leq_p T)$
For a derivation of this relation see Section 3.2 on page 74.

There are several choices to compute this matrix M:

- use a recursive function which directly computes $M_{p,uc}$ as defined above. This is not very efficient and rather trivial, therefore this option is not derived in this text.
- a column wise computation of M
- an optimized column wise computation of M

2.6 Column wise computation

The dynamic programming matrix described in section 2.5 can be computed both column by column and row by row. The column-wise computation used $\mathcal{O}(k)$ memory and the row-wise computation used $\mathcal{O}(n)$ memory. Since for almost all practical applications $k \leq n$, only a derivation of the column wise algorithm will be presented. In the subsequent chapters both a basic and an optimized algorithm will be derived based on the algorithm in this chapter.

Since the basic algorithm is a special case of the optimized algorithm, this chapter will focus slightly on the optimized version.

The columns are stored in an array $C : Pref(p) \mapsto \mathbb{N}$. The array is indexed with strings for readability, in an implementation these strings can be mapped to natural numbers (the length of the string) which is possible since indexing strings are all prefixes of p (which is a linear domain).

Consider a specific column (corresponding to prefix u of the text) of the dynamic programming matrix. This column can be split into a head and a tail. Suppose that the last element of the head is called L, then it is possible to choose L such that all elements in the (possibly empty) tail are guaranteed to be larger than k. This split is formalized by the following invariants:

$$\langle \forall s \ : \ s \leq_p L \ : \ C[s] = M_{s,u} \rangle \tag{P_1}$$

$$\langle \forall s \ : \ L <_p s \leq_p p \ : \ k < M_{s,u} \rangle \ \wedge \ \langle \forall s \ : \ L <_p s \leq_p p \ : \ k < C[s] \rangle \tag{P_2}$$

$$L \leq_p p \tag{P_3}$$

The range predicate P_3 shows the valid choices for L and guarantees that the strings used to index C form a linear domain.

In the subsequent chapters different choices will be made for L (to be more specific: the basic algorithm uses $L = p$ (which makes invariant P_2 trivially true) and the optimized algorithm uses a dynamically changing L).

Consider the statement: $O := O \cup \langle \textbf{set} \ i \ : \ M_{p,uc} \leq i \leq k \ : \ (uc, w, i) \rangle$ of the algorithm in section 2.5. Using invariants P_1, P_2 and P_3 this can be rewritten as follows:

$$\begin{array}{ll}
 & \langle \textbf{set} \ i \ : \ M_{p,uc} \leq i \leq k \ : \ (uc, w, i) \rangle \\
= & \quad \{ \text{ case analysis } \} \\
 & \begin{cases} \langle \textbf{set} \ i \ : \ M_{p,uc} \leq i \leq k \ : \ (uc, w, i) \rangle & \text{if } L = p \\ \langle \textbf{set} \ i \ : \ M_{p,uc} \leq i \leq k \ : \ (uc, w, i) \rangle & \text{if } L \neq p \end{cases} \\
= & \quad \{ \ P_1(u, r := uc, w) \ \wedge \ L = p \ \} \\
 & \begin{cases} \langle \textbf{set} \ i \ : \ C[p] \leq i \leq k \ : \ (uc, w, i) \rangle & \text{if } L = p \\ \langle \textbf{set} \ i \ : \ M_{p,uc} \leq i \leq k \ : \ (uc, w, i) \rangle & \text{if } L \neq p \end{cases} \\
= & \quad \{ \ P_2(u, r := uc, w) \ \wedge \ L \neq p \ \} \\
 & \begin{cases} \langle \textbf{set} \ i \ : \ C[p] \leq i \leq k \ : \ (uc, w, i) \rangle & \text{if } L = p \\ \varnothing & \text{if } L \neq p \end{cases} \\
= & \quad \{ \ P_2(u, r := uc, w) \ \wedge \ L \neq p, \text{ remove case analysis } \} \\
 & \langle \textbf{set} \ i \ : \ C[p] \leq i \leq k \ : \ (uc, w, i) \rangle
\end{array}$$

Before presenting the intermediate algorithm, consider invariant P_1. The initialization is fairly straightforward:

$$\begin{array}{ll}
 & P_1(u, r := \varepsilon, T) \\
\equiv & \quad \{ \text{ subst } \}
\end{array}$$

$$
\begin{array}{ll}
 & \langle \forall s \ : \ s \leq_p L \ : \ C[s] = M_{s,\varepsilon} \rangle \\
\equiv & \quad \{ \text{ def } M \ \} \\
 & \langle \forall s \ : \ s \leq_p L \ : \ C[s] = |s| \rangle
\end{array}
$$

The increment statement $u, r := uc, w$ requires that the next column of the dynamic programming matrix is computed. Since $P_1(u, r := uc, w) \equiv \langle \forall s \ : \ s \leq_p L \ : \ C[s] = M_{s,uc} \rangle$ a separate program fragment S_0 is introduced with the following postcondition:

$$\langle \forall s \ : \ s \leq_p L \ : \ C[s] = M_{s,uc} \rangle \tag{R_0}$$

The following program fragment shows the shape of the algorithm and all remaining prove obligations. The choice for and initialization of L is tackled in Sections 2.7 and 2.8.

if $m \leq k \rightarrow O := \{(\varepsilon, T, m)\}$
$[\!]$ $k < m \rightarrow O := \varnothing$
fi;
$\{ \ P_0(u, r := \varepsilon, T) \ \}$
for $s : s \leq_p L \rightarrow C[s] := |s|$ **rof**;
$\{ \ P_{0..1}(u, r := \varepsilon, T) \ \} \ \{ \ P_{2..3}(u, r := \varepsilon, T)? \ \}$
$u, r := \varepsilon, T$;
do $r :: cw \rightarrow \{ \ P_{0..3} \ \}$
    S_0;
    $\{ \ R_0? \ \}$
    $\{ \ P_1(u, r := uc, w) \ \} \ \{ \ P_2(u, r := uc, w)? \ \}$
    for $i \in [C[p]..k] \rightarrow O := O \cup \{(uc, w, i)\}$ **rof**;
    $\{ \ P_{0..2}(u, r := uc, w) \ \} \ \{ \ P_3? \ \}$
    $u, r := uc, w$
od$\{ \ P_0 \ \wedge \ r = \varepsilon \ \}$
$\{ \ R \ \}$

2.6.1 Computing the next column

This section derives program fragment S_0 which must establish postcondition R_0 (see page 20).

To establish R_0 the following invariants are introduced:

$$\langle \forall s \ : \ s \leq_p f \ : \ C[s] = M_{s,uc} \rangle \ \wedge \ fg = L \tag{Q_0}$$

$$\begin{array}{l}
\langle \forall s \ : \ f <_p s <_p L \ : \ C[s] = M_{s,u} \rangle \ \wedge \\
(C[L] = M_{L,u} \ \vee \ M_{L\downarrow 1,u} \leq (M_{L,u} \downarrow C[L]))
\end{array} \tag{Q_1}$$

$$pC = M_{f,u} \ \vee \ f = L \tag{Q_2}$$

Again, these invariants divide column C into a head and a tail segment. The head segment is equal to the new column and the tail segment is equal to the previous column. Invariant Q_2 buffers the element for which both the previous column and current column are required.

When the tail segment of C becomes empty, the entire next column has been computed and thus R_0 is established. The guard of the repetition should express that the tail segment is nonempty: $(g :: dh)$.

$$
\begin{array}{ll}
 & Q_0(f, g := fd, h) \\
\equiv & \quad \{ \text{ subst } \} \\
 & \langle \forall s \ : \ s \leq_p fd \ : \ C[s] = M_{s,uc} \rangle \ \wedge \ fdh = L \\
\equiv & \quad \{ \ Q_0, \ g :: dh, \text{ split off } s = fd \ \} \\
 & C[fd] = M_{fd,uc}
\end{array}
$$

$$
\begin{array}{ll}
 & M_{fd,uc} \\
\equiv & \quad \{ \text{ def } M \ \} \\
 & (M_{f,u} + \delta(d,c)) \downarrow (M_{f,uc} + 1) \downarrow (M_{fd,u} + 1) \\
\equiv & \quad \{ \ Q_0, Q_2, fg = L \ \wedge \ g :: dh \Rightarrow f \neq L \ \} \\
 & (pC + \delta(d,c)) \downarrow (C[f] + 1) \downarrow (M_{fd,u} + 1) \\
\equiv & \quad \{ \text{ case analysis: } h = \varepsilon, h \neq \varepsilon \ \} \\
 & \left\{ \begin{array}{l} (pC + \delta(d,c)) \downarrow (C[f] + 1) \downarrow (M_{fd,u} + 1) \text{ if } h = \varepsilon \\ (pC + \delta(d,c)) \downarrow (C[f] + 1) \downarrow (M_{fd,u} + 1) \text{ if } h \neq \varepsilon \end{array} \right. \\
\equiv & \quad \{ \ Q_0 \ \wedge \ Q_1, h \neq \varepsilon \text{ thus } f <_p fd <_p L \ \} \\
 & \left\{ \begin{array}{l} (pC + \delta(d,c)) \downarrow (C[f] + 1) \downarrow (M_{fd,u} + 1) \text{ if } h = \varepsilon \\ (pC + \delta(d,c)) \downarrow (C[f] + 1) \downarrow (C[fd] + 1) \text{ if } h \neq \varepsilon \end{array} \right. \\
\equiv & \quad \{ \ Q_0 \ \wedge \ Q_1, h = \varepsilon \text{ thus } fd = L \text{ therefore: } C[fd] = C[L] = M_{L,u} \\
 & \qquad \text{or } pC = M_{f,u} = M_{L\downarrow 1,u} \leq (M_{L,u} \downarrow C[L]) = (M_{fd,u} \downarrow C[fd]) \\
 & \quad \} \\
 & \left\{ \begin{array}{l} (pC + \delta(d,c)) \downarrow (C[f] + 1) \downarrow (C[fd] + 1) \text{ if } h = \varepsilon \\ (pC + \delta(d,c)) \downarrow (C[f] + 1) \downarrow (C[fd] + 1) \text{ if } h \neq e \end{array} \right. \\
\equiv & \quad \{ \text{ remove case analysis } \} \\
 & (pC + \delta(d,c)) \downarrow (C[f] + 1) \downarrow (C[fd] + 1)
\end{array}
$$

$$
\begin{array}{ll}
 & Q_1(f, g := fd, h) \\
\equiv & \quad \{ \text{ subst } \} \\
 & \langle \forall s \ : \ fd <_p s <_p L \ : \ C[s] = M_{s,u} \rangle \ \wedge \\
 & (C[L] = M_{L,u} \ \vee \ M_{L\downarrow 1,u} \leq (M_{L,u} \downarrow C[L])) \\
\equiv & \quad \{ \ Q_1 \ \} \\
 & true
\end{array}
$$

$$
\begin{array}{ll}
 & Q_2(f, g := fd, h) \\
\equiv & \quad \{ \text{ subst } \} \\
 & pC = M_{fd,u} \ \vee \ fd = L
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{ \text{ case analysis } \} \\
& \begin{cases} pC = M_{fd,u} & \text{if } fd \neq L \\ true & \text{if } fd = L \end{cases} \\
\equiv & \{\ Q_0\ \wedge\ Q_1 : fdh = L, fd \neq L \text{ hence } f <_p fd <_p L\ \} \\
& \begin{cases} pC = C[fd] & \text{if } fd \neq L \\ true & \text{if } fd = L \end{cases}
\end{array}
$$

This has proven the following piece of code:

$\{\ Q_{0..2}\ \}$
$C[fd], pC := (pC + \delta(d,c)) \downarrow (C[f]+1) \downarrow (C[fd]+1), C[fd];$
$\{\ Q_{0..2}(f, g := fd, h)\ \}$

All that remains is the initialization of the three invariants:

$$
\begin{array}{ll}
& Q_2(f, g := \varepsilon, L) \\
\equiv & \{ \text{ subst } \} \\
& pC = M_{\varepsilon,u}\ \vee\ \varepsilon = L \\
\equiv & \{ \text{ def } M \ \} \\
& pC = 0\ \vee\ \varepsilon = L
\end{array}
$$

This is clearly initialized by $pC := 0$.

$$
\begin{array}{ll}
& (Q_0\ \wedge\ Q_1)(f, g := \varepsilon, L) \\
\equiv & \{ \text{ subst } \} \\
& \langle \forall s\ :\ s \leq_p \varepsilon\ :\ C[s] = M_{s,uc} \rangle\ \wedge\ \varepsilon \cdot L = L\ \wedge \\
& \langle \forall s\ :\ \varepsilon <_p s <_p L\ :\ C[s] = M_{s,u} \rangle\ \wedge \\
& (C[L] = M_{L,u}\ \vee\ M_{L\restriction 1,u} \leq (M_{L,u} \downarrow C[L])) \\
\equiv & \{ \text{ calc } \} \\
& C[\varepsilon] = M_{\varepsilon,uc}\ \wedge \\
& \langle \forall s\ :\ \varepsilon <_p s <_p L\ :\ C[s] = M_{s,u} \rangle\ \wedge \\
& (C[L] = M_{L,u}\ \vee\ M_{L\restriction 1,u} \leq (M_{L,u} \downarrow C[L])) \\
\equiv & \{\ M_{\varepsilon,uc} = 0 = M_{\varepsilon,u}, \text{ merge domains } \} \\
& \langle \forall s\ :\ s <_p L\ :\ C[s] = M_{s,u} \rangle\ \wedge \\
& (C[L] = M_{L,u}\ \vee\ M_{L\restriction 1,u} \leq (M_{L,u} \downarrow C[L]))
\end{array}
$$

This becomes the precondition of program fragment S_0 and must be satisfied in the outer repetition.

$$
\begin{array}{l}
\langle \forall s\ :\ s <_p L\ :\ C[s] = M_{s,u} \rangle\ \wedge \\
(C[L] = M_{L,u}\ \vee\ M_{L\restriction 1,u} \leq (M_{L,u} \downarrow C[L]))
\end{array}
\qquad (Z)
$$

Note that invariant P_1 implies Z. In the optimized algorithm in Section 2.8 the value of L will change and break validity of Z. For this reason, proving of assertion Z is postponed to the following chapters.

The complete algorithm:

Algorithm 2.6.7(PE, SP, EP, BM, DM, Col)

$R : O = \langle \textbf{set}\ l, v, r\ :\ lvr = T\ \wedge\ dst(v,p) \leq k\ :\ (lv, r, dst(v,p))\rangle$
$P_0 : ur = T\ \wedge\ O = \langle \textbf{set}\ x, y, z\ :\ xyz = T\ \wedge\ xy \leq_p u\ \wedge\ dst(y,p) \leq k\ :\ (xy, z, dst(y,p))\rangle$
$P_1 : \langle \forall s\ :\ s \leq_p L\ :\ C[s] = M_{s,u}\rangle$
$P_2 : \langle \forall s\ :\ L <_p s \leq_p p\ :\ k < M_{s,u}\rangle\ \wedge\ \langle \forall s\ :\ L <_p s \leq_p p\ :\ k < C[s]\rangle$
$P_3 : L \leq_p p$
$R_0 : \langle \forall s\ :\ s \leq_p L\ :\ C[s] = M_{s,uc}\rangle$
$Q_0 : \langle \forall s\ :\ s \leq_p f\ :\ C[s] = M_{s,uc}\rangle\ \wedge\ fg = L$
$Q_1 : \langle \forall s\ :\ f <_p s <_p L\ :\ C[s] = M_{s,u}\rangle\ \wedge\ (C[L] = M_{L,u}\ \vee\ M_{L\downarrow 1,u} \leq (M_{L,u} \downarrow C[L]))$
$Q_2 : pC = M_{f,u}\ \vee\ f = L$
$Z : \langle \forall s\ :\ s <_p L\ :\ C[s] = M_{s,u}\rangle\ \wedge\ (C[L] = M_{L,u}\ \vee\ M_{L\downarrow 1,u} \leq (M_{L,u} \downarrow C[L]))$

if $m \leq k \rightarrow O := \{(\varepsilon, T, m)\}$
$[\!]\ k < m \rightarrow O := \varnothing$
fi;
$\{\ P_0(u, r := \varepsilon, T)\ \}$
for $s : s \leq_p L \rightarrow C[s] := |s|$ **rof**;
$\{\ P_{0..1}(u, r := \varepsilon, T)\ \}\ \{\ P_{2..3}(u, r := \varepsilon, T)?\ \}$
$u, r := \varepsilon, T$;
do $r :: cw \rightarrow \{\ P_{0..3}\ \}$
  $\{\ Z?\ \}$
  $\{\ Q_{0..1}(f, g := \varepsilon, L)\ \}$
  $pC := 0$;
  $\{\ Q_{0..2}(f, g := \varepsilon, L)\ \}$
  $f, g := \varepsilon, L$;
  do $g :: dh \rightarrow \{\ Q_{0..2}\ \}$
    $C[fd], pC := (pC + \delta(d,c)) \downarrow (C[f] + 1) \downarrow (C[fd] + 1), C[fd]$;
    $\{\ Q_{0..2}(f, g := fd, h)\ \}$
    $f, g := fd, h$;
  od$\{\ Q_{0..2}\ \wedge\ g = \varepsilon\ \}$
  $\{\ R_0\ \}$
  $\{\ P_1(u, r := uc, w)\ \}\ \{\ P_2(u, r := uc, w)?\ \}$
  for $i \in [C[p]..k] \rightarrow O := O \cup \{(uc, w, i)\}$ **rof**;
  $\{\ P_{0..2}(u, r := uc, w)\ \}\ \{\ P_3?\ \}$
  $u, r := uc, w$
od$\{\ P_0\ \wedge\ r = \varepsilon\ \}$
$\{\ R\ \}$

2.7 Basic dynamic programming

The algorithm in section 2.6.1 computes columns of the dynamic programming matrix up to some element L. This section shows how a basic dynamic programming algorithm can be derived by computing the entire column.

The algorithm computes the entire column (i.e. up to p) when $L = p$. Therefore make $L = p$ a system invariant. Both invariants P_2 and P_3 become trivially true. The only remaining proof obligation is the validity of Z. Note that Z is a weakening of invariant P_1 and therefore trivially holds as well. In fact, the additional disjunct in Z is only there to establish correctness of the optimized algorithm.

The basic dynamic programming algorithm: (provided $L = p$ is a system invariant. For a list of invariants refer to the algorithm on page 23)

Algorithm 2.7.8(PE, SP, EP, BM, DP, Col, Basic)

if $m \leq k \rightarrow O := \{(\varepsilon, T, m)\}$
$[\!]$ $k < m \rightarrow O := \varnothing$
fi;
$\{\ P_0(u, r := \varepsilon, T)\ \}$
for $s : s \leq_p L \rightarrow C[s] := |s|$ **rof**;
$\{\ P_{0..3}(u, r := \varepsilon, T)\ \}$
$u, r := \varepsilon, T$;
do $r :: cw \rightarrow \{\ P_{0..3}\ \}$
  $\{\ Z\ \}$
  $\{\ Q_{0..1}(f, g := \varepsilon, L)\ \}$
  $pC := 0$;
  $\{\ Q_{0..2}(f, g := \varepsilon, L)\ \}$
  $f, g := \varepsilon, L$;
  do $g :: dh \rightarrow \{\ Q_{0..2}\ \}$
    $C[fd], pC := (pC + \delta(d, c)) \downarrow (C[f] + 1) \downarrow (C[fd] + 1), C[fd]$;
    $\{\ Q_{0..2}(f, g := fd, h)\ \}$
    $f, g := fd, h$;
  od$\{\ Q_{0..2} \ \wedge\ g = \varepsilon\ \}$
  $\{\ R_0\ \}$
  $\{\ P_{1..3}(u, r := uc, w)\ \}$
  for $i \in [C[p]..k] \rightarrow O := O \cup \{(uc, w, i)\}$ **rof**;
  $\{\ P_{0..3}(u, r := uc, w)\ \}$
  $u, r := uc, w$
od$\{\ P_0 \ \wedge\ r = \varepsilon\ \}$
$\{\ R\ \}$

2.8 Optimized dynamic programming

The optimized dynamic programming algorithm dynamically changes L such that as few elements as possible of the dynamic programming matrix are computed.

Invariant P_2 dictates how L should be initialized as is shown next:

$$
\begin{array}{ll}
 & P_2(u, r := \varepsilon, T) \\
\equiv & \quad \{ \text{ subst } \} \\
 & \langle \forall s \ : \ L <_p s \leq_p p \ : \ k < M_{s,\varepsilon} \rangle \ \wedge \ \langle \forall s \ : \ L <_p s \leq_p p \ : \ k < C[s] \rangle \\
\equiv & \quad \{ \text{ def } M \ \} \\
 & \langle \forall s \ : \ L <_p s \leq_p p \ : \ k < |s| \rangle \ \wedge \ \langle \forall s \ : \ L <_p s \leq_p p \ : \ k < C[s] \rangle \\
\Leftarrow & \quad \{ \ L <_p s, \text{ hence } |L| < |s| \ \} \\
 & \langle \forall s \ : \ L <_p s \leq_p p \ : \ k \leq |L| \rangle \ \wedge \ \langle \forall s \ : \ L <_p s \leq_p p \ : \ k < C[s] \rangle \\
\equiv & \quad \{ \text{ empty domain or not } \} \\
 & L = p \ \vee \ (k \leq |L| \ \wedge \ \langle \forall s \ : \ L <_p s \leq_p p \ : \ k < C[s] \rangle)
\end{array}
$$

From this derivation a possible choice is to initialize L to $p \uparrow k$.

Note that $p \uparrow k$ is either a prefix of length k (if $k < m$) or p itself (if $m \leq k$). In case $p \uparrow k = p$ then $L = p$ is satisfied, in case $p \uparrow k$ is a prefix of length k then $k \leq |p \uparrow k|$ is satisfied.

Note that initialization $L := p$ is possible as well. This initialization causes the column C to be computed all the way up to $C[p]$, this is a decrease in efficiency since it is sufficient to compute C up to $C[p \uparrow k]$. Also note that $(p \uparrow k) \leq_p p$.

Since initialization $L := p \uparrow k$ does not guarantee that $L = p$ column C must be initialized as well.

$$\langle \forall s \ : \ L <_p s \leq_p p \ : \ k < C[s] \rangle$$

There are two choices for this initialization:

- Initialize $C[s]$ to $k + 1$ (for $L <_p s \leq_p p$). The advantage of initializing $C[s]$ to $k + 1$ is that the data type of column C has a strong upper bound (maximum required value is $k + 1$).
- A simpler initialization is acquired by combining this formula with the requirement for C which was derived for the initialization of P_1: (suppose $k \leq |L|$)

$$
\begin{array}{ll}
 & \langle \forall s \ : \ s \leq_p L \ : \ C[s] = |s| \rangle \ \wedge \ \langle \forall s \ : \ L <_p s \leq_p p \ : \ k < C[s] \rangle \\
\Leftarrow & \quad \{ \ k \leq |L| \ \} \\
 & \langle \forall s \ : \ s \leq_p p \ : \ C[s] = |s| \rangle
\end{array}
$$

The second choice will be used in the remainder of this text for the sake of simplicity.

Next look at the increment of u.

$$
\begin{array}{ll}
 & P_2(u, r := uc, w) \\
\equiv & \quad \{ \text{ subst } \} \\
 & \langle \forall s \ : \ L <_p s \leq_p p \ : \ k < M_{s,uc} \rangle \ \wedge \ \langle \forall s \ : \ L <_p s \leq_p p \ : \ k < C[s] \rangle
\end{array}
$$

$\Leftarrow$ { transitivity $\leq$ }

$\langle \forall s \ : \ L <_p s \leq_p p \ : \ k < M_{s\downarrow 1,u} \rangle \ \wedge \ \langle \forall s \ : \ L <_p s \leq_p p \ : \ M_{s\downarrow 1,u} \leq M_{s,uc} \rangle$
$\wedge \ \langle \forall s \ : \ L <_p s \leq_p p \ : \ k < C[s] \rangle$

$\equiv$ { property 3.2.1 on page 76 (diagonals of M are ascending) }

$\langle \forall s \ : \ L <_p s \leq_p p \ : \ k < M_{s\downarrow 1,u} \rangle \ \wedge \ \langle \forall s \ : \ L <_p s \leq_p p \ : \ k < C[s] \rangle$

$\equiv$ { dummy transform }

$\langle \forall s \ : \ L \leq_p s <_p p \ : \ k < M_{s,u} \rangle \ \wedge \ \langle \forall s \ : \ L <_p s \leq_p p \ : \ k < C[s] \rangle$

$\Leftarrow$ { empty domain or smaller domain }

$L = p \ \vee \ P_2(L := L \downarrow 1)$

This derivation shows that L should increase if $L \neq p$. The motivation for this is as follows.

Invariant P_2 shows that for all s in the range $L <_p s \leq_p p$ it holds that $k < M_{s,u}$. Because diagonals are ascending, one can conclude that for s in the range $Ld <_p s \leq_p p$ it holds that $k < M_{s,uc}$. It is impossible to conclude that this holds for $s = Ld$ as well. Therefore L should increase.

if $m \leq k \rightarrow O := \{(\varepsilon, T, m)\}$
$[\!]$ $k < m \rightarrow O := \varnothing$
fi;
{ $P_0(u, r := \varepsilon, T)$ }
for $s : s \leq_p p \rightarrow C[s] := |s|$ **rof**;
{ $P_{0..1}(u, r := \varepsilon, T)$ }
$L := p \uparrow k$;
{ $P_{0..3}(u, r := \varepsilon, T)$ }
$u, r := \varepsilon, T$;
do $r :: cw \rightarrow$ { $P_{0..3}$ }
  { $P_{2..3}$ }
  if $L = p \rightarrow$ skip
  $[\!]$ $L \neq p \rightarrow L := L + p[|L| + 1]$
  fi{ $L = p \ \vee \ P_2(L := L \downarrow 1)$ }
  { $P_{2..3}(u, r := uc, w)$ } { $Z?$ }
  { $Q_{0..1}(f, g := \varepsilon, L)$ }
  $pC := 0$;
  { $Q_{0..2}(f, g := \varepsilon, L)$ }
  $f, g := \varepsilon, L$;
  do $g :: dh \rightarrow$ { $Q_{0..2}$ }
    $C[fd], pC := (pC + \delta(d, c)) \downarrow (C[f] + 1) \downarrow (C[fd] + 1), C[fd]$;
    { $Q_{0..2}(f, g := fd, h)$ }
    $f, g := fd, h$;
  od{ $Q_{0..2} \ \wedge \ g = \varepsilon$ }
  { R_0 }
  { $P_{1..3}(u, r := uc, w)$ }
  for $i \in [C[p]..k] \rightarrow O := O \cup \{(uc, w, i)\}$ **rof**;
  { $P_{0..3}(u, r := uc, w)$ }

$u, r := uc, w$
od$\{\ P_0\ \wedge\ r = \varepsilon\ \}$
$\{\ R\ \}$

All that remains is to establish assertion Z. Note that P_1 implies Z. This covers the case that $L = p$ and L does not change. Next consider the case that L is increased. For brevity, let $d = p[|L| + 1]$.

$$
\begin{array}{ll}
 & Z(L := Ld) \\
\equiv & \quad \{\ \text{subst}\ \} \\
 & \langle \forall s\ :\ s <_p Ld\ :\ C[s] = M_{s,u} \rangle\ \wedge\ (C[Ld] = M_{Ld,u}\ \vee\ M_{Ld\downharpoonright 1,u} \leq (M_{Ld,u} \downarrow C[Ld])) \\
\equiv & \quad \{\ <_p, \downharpoonright\ \} \\
 & \langle \forall s\ :\ s \leq_p L\ :\ C[s] = M_{s,u} \rangle\ \wedge\ (C[Ld] = M_{Ld,u}\ \vee\ M_{L,u} \leq (M_{Ld,u} \downarrow C[Ld])) \\
\equiv & \quad \{\ P_1\ \} \\
 & C[Ld] = M_{Ld,u}\ \vee\ M_{L,u} \leq (M_{Ld,u} \downarrow C[Ld]) \\
\equiv & \quad \{\ \text{transitivity of}\ <\ \} \\
 & C[Ld] = M_{Ld,u}\ \vee\ M_{L,u} \leq k\ \wedge\ k < (M_{Ld,u} \downarrow C[Ld]) \\
\equiv & \quad \{\ L <_p Ld \leq_p p,\ P_2 : k < M_{Ld,u}\ \wedge\ k < C[Ld]\ \} \\
 & C[Ld] = M_{Ld,u}\ \vee\ M_{L,u} \leq k \\
\equiv & \quad \{\ P_1 : C[L] = M_{L,u}\ \} \\
 & C[Ld] = M_{Ld,u}\ \vee\ C[L] \leq k \\
\Leftarrow & \quad \{\ \text{calc}\ \} \\
 & C[L] \leq k
\end{array}
$$

So $\{\ C[L] \leq k\ \wedge\ P_{1..3}\ \}\ L := Ld\ \{\ Z\ \}$ is valid.

Since no values above L should be computed (since they are larger than k), it is impossible to verify if $C[Ld] = M_{Ld,u}$ in the derivation above. Therefore $C[L] \leq k$ must be established. This can be done with a linear search since $C[\varepsilon] = 0$ (provided that $0 < k$, which is a reasonable assumption).

do $k < C[L] \rightarrow L := L \downharpoonright 1$ **od**
$\{\ C[L] \leq k\ \}$

Note that decrementing L maintains P_1 (it is widening). Invariant P_2 is also maintained due to the guard of the linear search and P_1 and P_2:

$$
\begin{array}{ll}
 & P_2(L := L \downharpoonright 1) \\
\equiv & \quad \{\ \text{subst}\ \} \\
 & \langle \forall s\ :\ (L \downharpoonright 1) <_p s \leq_p p\ :\ k < M_{s,u} \rangle\ \wedge\ \langle \forall s\ :\ (L \downharpoonright 1) <_p s \leq_p p\ :\ k < C[s] \rangle \\
\equiv & \quad \{\ P_2, \text{split of}\ s = L\ \}
\end{array}
$$

$$
\begin{array}{ll}
 & k < M_{L,u} \ \wedge \ k < C[L] \\
\equiv & \quad \{\ P_1 : C[L] = M_{L,u}\ \} \\
 & k < C[L] \\
\equiv & \quad \{\ \text{guard}\ \} \\
 & true
\end{array}
$$

The optimized dynamic programming algorithm (for a list of the invariants refer to page 23):

Algorithm 2.8.9(PE, SP, EP, BM, DP, Col, Opt)

if $m \leq k \rightarrow O := \{(\varepsilon, T, m)\}$
$[\!]$ $k < m \rightarrow O := \varnothing$
fi;
$\{\ P_0(u, r := \varepsilon, T)\ \}$
for $s : s \leq_p p \rightarrow C[s] := |s|$ **rof**;
$\{\ P_{0..1}(u, r := \varepsilon, T)\ \}$
$L := p \restriction k$;
$\{\ P_{0..3}(u, r := \varepsilon, T)\ \}$
$u, r := \varepsilon, T$;
do $r :: cw \rightarrow \{\ P_{0..3}\ \}$
  do $k < C[L] \rightarrow L := L \downarrow 1$ **od**
  $\{\ C[L] \leq k\ \}\ \{\ P_{1..3}\ \}$
  if $L = p \rightarrow$ skip
  $[\!]$ $L \neq p \rightarrow L := L + p[|L| + 1]$
  fi$\{\ L = p \ \vee \ P_2(L := L \downarrow 1)\ \}$
  $\{\ P_{2..3}(u, r := uc, w)\ \}\ \{\ Z\ \}$
  $\{\ Q_{0..1}(f, g := \varepsilon, L)\ \}$
  $pC := 0$;
  $\{\ Q_{0..2}(f, g := \varepsilon, L)\ \}$
  $f, g := \varepsilon, L$;
  do $g :: dh \rightarrow \{\ Q_{0..2}\ \}$
    $C[fd], pC := (pC + \delta(d, c)) \downarrow (C[f] + 1) \downarrow (C[fd] + 1), C[fd]$;
    $\{\ Q_{0..2}(f, g := fd, h)\ \}$
    $f, g := fd, h$;
  od$\{\ Q_{0..2} \ \wedge \ g = \varepsilon\ \}$
  $\{\ R_0\ \}$
  $\{\ P_{1..3}(u, r := uc, w)\ \}$
  for $i \in [C[p]..k] \rightarrow O := O \cup \{(uc, w, i)\}$ **rof**;
  $\{\ P_{0..3}(u, r := uc, w)\ \}$
  $u, r := uc, w$
od$\{\ P_0 \ \wedge \ r = \varepsilon\ \}$
$\{\ R\ \}$

2.9 Automaton algorithms

Another approach to refine the algorithm in Section 2.4 on page 16 is to compute the existential quantification ($\langle \exists x, y \ : \ xy = w \ : \ dst(y, p) = i \rangle$) using an automaton. Consider an NFA constructed as follows:

- create a row matching the exact pattern for each l between 0 and the maximum distance k.
- Connect each i^{th} state on level l to the i^{th} state on level $l + 1$ with a Σ transition and to the $i + 1^{th}$ state on level $l + 1$ with a Σ transition and an ε transition.
- Add a Σ loop on the 0^{th} state on level 0 (this changes the automaton from a pattern validater into a pattern searcher).

For an example, see Figure 2.1. The horizontal arrows correspond to matching a character

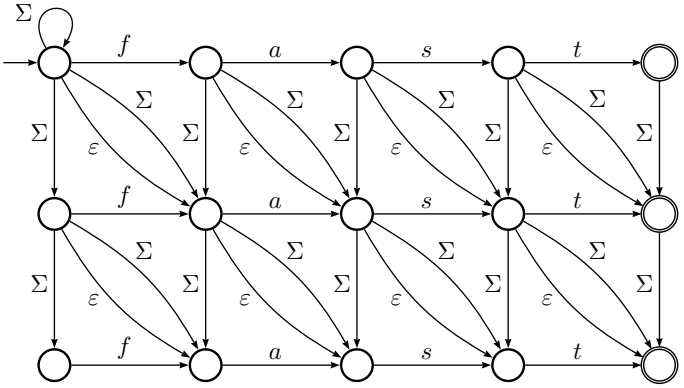


Figure 2.1: NFA for matching string *fast* with a maximum distance of two

in the text to a character in the pattern. The vertical arrows correspond to inserting a character into the pattern. The diagonal Σ transition is a character substitution (substituting a character by itself is allowed) and the ε transitions correspond to character deletion from the pattern.

The automaton can be used to match the pattern in the text. Every time an end state is reached, an approximate occurrence of the pattern has been encountered.

The states of the automaton are identified by their row and column number (state (i, j) is the j^{th} state on the i^{th} row). This automaton is characterized by the following equation:

$$(i, m) \in \delta^*(q_0, w) \equiv \langle \exists x, y \ : \ xy = w \ : \ dst(y, p) = i \rangle$$

For a formal definition of the automaton and proofs of its characterization and several properties refer to Section 3.3 on page 79.

The algorithms using this automaton need to know the value of $\delta^*(q_0, uc)$. This section derives an algorithm using an abstraction function to abstract over the actual representation of $\delta^*(q_0, uc)$. Subsequent chapters will instantiate this abstraction function.

The following invariant P_1 is added to the loop:

$$[[M]] = \delta^*(q_0, u)$$

If invariant P_1 is valid, set O can be updated as is shown next:

$$\begin{array}{ll} & \langle \textbf{set } i \ : \ 0 \leq i \leq k \ \wedge \ \langle \exists x, y \ : \ xy = uc \ : \ dst(y,p) = i \rangle \ : \ (uc, w, i) \rangle \\ = & \quad \{ \text{ automaton } \} \\ & \langle \textbf{set } i \ : \ 0 \leq i \leq k \ \wedge \ (i,m) \in \delta^*(q_0, uc) \ : \ (uc, w, i) \rangle \\ = & \quad \{ \ P_1(u, r := uc, w) \ \} \\ & \langle \textbf{set } i \ : \ 0 \leq i \leq k \ \wedge \ (i,m) \in [[M]] \ : \ (uc, w, i) \rangle \end{array}$$

This set can be computed in a simple loop. By updating the running program with the above result, the following program fragment is acquired:

Algorithm 2.9.10(PE, SP, EP, BM, AA)

$R : O = \langle \textbf{set } l, v, r \ : \ lvr = T \ \wedge \ dst(v,p) \leq k \ : \ (lv, r, dst(v,p)) \rangle$
$P_0 : ur = T \ \wedge \ O = \langle \textbf{set } x, y, z \ : \ xyz = T \ \wedge \ xy \leq_p u \ \wedge \ dst(y,p) \leq k \ : \ (xy, z, dst(y,p)) \rangle$
$P_1 : [[M]] = \delta^*(q_0, u)$

if $m \leq k \rightarrow O := \{(\varepsilon, T, m)\}$
$[\!]$ $k < m \rightarrow O := \varnothing$
fi; $\{ \ P_0(u, r := \varepsilon, T) \ \}$
$M : [[M]] = \delta^*(q_0, \varepsilon)$;
$\{ \ P_{0..1}(u, r := \varepsilon, T) \ \}$
$u, r := \varepsilon, T$;
do $r :: cw \rightarrow \{ \ P_{0..1} \ \}$
    $M : [[M]] = \delta^*(q_0, uc)$;
    $\{ \ P_0 \ \wedge \ P_1(u, r := uc, w) \ \}$
    for $i \in [0..k] \rightarrow$
        if $(i,m) \in [[M]] \rightarrow O := O \cup \{(uc, w, i)\}$
        $[\!]$ $(i,m) \notin [[M]] \rightarrow$ skip
        fi
    rof;
    $\{ \ P_{0..1}(u, r := uc, w) \ \}$
    $u, r := uc, w$
od$\{ \ P_0 \ \wedge \ r = \varepsilon \ \}$
$\{ \ R \ \}$

Remark: There is an optimization possible due to Property 3.3.3 on page 81. If state (i,m) is inactive then all states (j,m) (with $0 \leq j \leq i$) will be inactive as well. Therefore the entire for loop (i.e. the inner loop) can be replaced by a downwards for loop (range $[k..0]$) which terminates once $(i,m) \notin [[M]]$ or $i < 0$.

2.10 A set based approach

The algorithm described in section 2.9 does not describe how the automaton can be implemented. There are many different techniques to simulate a nondeterministic finite automaton. To name a few:

- Use the subset construction to create a DFA (this results in the optimal $\mathcal{O}(n)$ worst case algorithm). The upper bound of the number of states in the resulting DFA is (see [Ukk85]): $\mathcal{O}(min(3^m, m(2m|\Sigma|)^k))$.
- Partially construct the DFA (a lazy runtime construction). This constructs only those states that are actually used during the matching process.
- Maintain sets of active states and perform transitions from each of the active states.

There are many more techniques, but generic NFA simulation is a domain which is large enough to taxonomize by itself and falls outside the scope of this text. Nevertheless, this section will derive an algorithm using sets to simulate a generic automaton (i.e. the algorithm will not use any properties of the specific transition function of the Approximate Pattern Matching automaton).

The set based algorithm is acquired when the most simple abstraction function available is used: (with M a set)

$$[[M]] = M \tag{2.1}$$

For the initialization statement $M : [[M]] = \delta^*(q_0, \varepsilon)$ consider the following derivation:

$$\begin{aligned}
& \delta^*(q_0, \varepsilon) \\
\equiv \quad & \{\ \delta^*\ \} \\
& \varepsilon\text{-closure}(q_0) \\
\equiv \quad & \{\ \varepsilon\text{-closure},\ q_0 = (0,0)\ \} \\
& \langle \textbf{set}\ l\ :\ 0 \leq l \leq (k \downarrow m)\ :\ (l,l) \rangle
\end{aligned}$$

This shows that statement $M := \langle \textbf{set}\ l\ :\ 0 \leq l \leq (k \downarrow m)\ :\ (l,l) \rangle$ is a proper initialization.

Next consider the statement $M : [[M]] = \delta^*(q_0, uc)$:

$$\begin{aligned}
& \delta^*(q_0, uc) \\
\equiv \quad & \{\ \delta^*, \text{states have shape } (i,j)\ \} \\
& \langle \bigcup q, i, j\ :\ q \in \delta^*(q_0, u)\ \wedge\ (i,j) \in \delta(q,c)\ :\ \varepsilon\text{-closure}((i,j)) \rangle \\
\equiv \quad & \{\ P_1,\ [[M]]\ \} \\
& \langle \bigcup q, i, j\ :\ q \in M\ \wedge\ (i,j) \in \delta(q,c)\ :\ \varepsilon\text{-closure}((i,j)) \rangle \\
\equiv \quad & \{\ \varepsilon\text{-closure}\ \} \\
& \langle \bigcup q, i, j\ :\ q \in M\ \wedge\ (i,j) \in \delta(q,c)\ :\ \langle \textbf{set}\ l\ :\ 0 \leq l \leq ((k-i) \downarrow (m-j))\ :\ (i+l, j+l) \rangle \rangle \\
\equiv \quad & \{\ \text{nesting}\ \} \\
& \langle \textbf{set}\ q, i, j, l\ :\ q \in M\ \wedge\ (i,j) \in \delta(q,c)\ \wedge\ 0 \leq l \leq ((k-i) \downarrow (m-j))\ :\ (i+l, j+l) \rangle
\end{aligned}$$

Combining this with the previous program template results in the following program:

Algorithm 2.10.11(PE, SP, EP, BM, AA, Set)

$R : O = \langle \textbf{set } l, v, r \ : \ lvr = T \ \wedge \ dst(v,p) \leq k \ : \ (lv, r, dst(v,p)) \rangle$
$P_0 : ur = T \ \wedge \ O = \langle \textbf{set } x, y, z \ : \ xyz = T \ \wedge \ xy \leq_p u \ \wedge \ dst(y,p) \leq k \ : \ (xy, z, dst(y,p)) \rangle$
$P_1 : [[M]] = \delta^*(q_0, u)$

if $m \leq k \rightarrow O := \{(\varepsilon, T, m)\}$
$[\!]$ $k < m \rightarrow O := \varnothing$
fi; $\{ \ P_0(u, r := \varepsilon, T) \ \}$
$M := \langle \textbf{set } l \ : \ 0 \leq l \leq (k \downarrow m) \ : \ (l, l) \rangle$;
$\{ \ P_{0..1}(u, r := \varepsilon, T) \ \}$
$u, r := \varepsilon, T$;
do $r :: cw \rightarrow \{ \ P_{0..1} \ \}$
    $M := \langle \textbf{set } q, i, j, l \ : \ q \in M \ \wedge \ (i,j) \in \delta(q, c) \ \wedge \ 0 \leq l \leq ((k-i) \downarrow (m-j)) \ : \ (i+l, j+l) \rangle$;
    $\{ \ P_0 \ \wedge \ P_1(u, r := uc, w) \ \}$
    for $i \in [0..k] \rightarrow$
        if $(i, m) \in M \rightarrow O := O \cup \{(uc, w, i)\}$
        $[\!]$ $(i, m) \notin M \rightarrow$ skip
        fi
    rof;
    $\{ \ P_{0..1}(u, r := uc, w) \ \}$
    $u, r := uc, w$
od$\{ \ P_0 \ \wedge \ r = \varepsilon \ \}$
$\{ \ R \ \}$

A possible implementation of the initialization statement is:

$M := \varnothing$;
for $l \in [0..(k \downarrow m)] \rightarrow M := M \cup \{(l, l)\}$ **rof**

The easiest (but certainly not the most efficient) way to compute the update of M is by using several repetitions:

$TS := \varnothing$
for $q \in M \rightarrow$
    for $(i, j) \in \delta(q, c) \rightarrow$
        for $l \in [0, .., k - i \downarrow m - j] \rightarrow$
            $TS := TS \cup \{(i + l, j + l)\}$
        rof
    rof
rof;
$M := TS$

These implementations and their variations are not described in any more detail in this text.

2.11 Row wise bit parallel simulation

This section continues from the algorithm described in Section 2.9.

This algorithm simulates the NFA using bit parallelism (see also Section B). In the best case, this provides a constant factor (the size of the machine words) speedup over non bit parallel automaton simulations. This speedup can be significant if the machine words are large enough or if the pattern is small enough.

The algorithm in this section is a generalization of the shift-and algorithm for exact keyword pattern matching. In this section the mapping between the states in the automaton and the bits in the machine words is as follows.

Each row of the automaton is packed into a machine word. This results in $k+1$ rows of $m+1$ bits. Note that the NFA (see Figure 2.1 on page 29) is shaped in a grid like form. This allows a state to be identified by its row and column indices. So state (i, j) corresponds with machine word i bit j.

In this section the machine words are represented as boolean arrays.

Let M be a boolean matrix: $M : Matrix[0..k, 0..m] : boolean$. The following abstraction function shows how the active states of the automaton are represented by matrix M:

$$[[M]] = \langle \textbf{set } i, j \ : \ 0 \leq i \leq k \ \wedge \ 0 \leq j \leq m \ \wedge \ M[i, j] \ : \ (i, j) \rangle$$

The previous program fragment contains the statement $M : [[M]] = \delta^*(q_0, \varepsilon)$ and the statement $M : [[M]] = \delta^*(q_0, uc)$ and two guards ($(i, m) \in [[M]]$ and $(i, m) \notin [[M]]$) which must be refined.

First consider one of the guards: (for $0 \leq i \leq k$)

$$\begin{array}{ll} & (i, m) \in [[M]] \\ \equiv & \quad \{ \ [[M]] \ \} \\ & (i, m) \in \langle \textbf{set } i, j \ : \ 0 \leq i \leq k \ \wedge \ 0 \leq j \leq m \ \wedge \ M[i, j] \ : \ (i, j) \rangle \\ \equiv & \quad \{ \ 0 \leq i \leq k \ \} \\ & M[i, m] \end{array}$$

From this can be concluded that $(i, m) \in [[M]] \equiv M[i, m]$ and due to the equivalence, the negation holds as well $(i, m) \notin [[M]] \equiv \neg M[i, m]$.

Now focus on statement $M : [[M]] = \delta^*(q_0, uc)$. Invariant P_1 states $[[M]] = \delta^*(q_0, u)$. This assignment corresponds to processing a single character c with an automaton whose active states are equal to $[[M]]$.

Consider the value of a single cell $M[i, j]$ of matrix M:

$$\begin{array}{ll} & M[i, j] \\ \equiv & \quad \{ \ [[M]] \ \} \\ & (i, j) \in [[M]] \\ \equiv & \quad \{ \ [[M]] = \delta^*(q_0, uc) \ \} \\ & (i, j) \in \delta^*(q_0, uc) \end{array}$$

This expression is established by introducing program fragment S_0 with postcondition R_0.

$$\langle \forall i \ : \ 0 \leq i \leq k \ : \ \langle \forall j \ : \ 0 \leq j \leq m \ : \ M[i,j] \equiv (i,j) \in \delta^*(q_0, uc) \rangle \rangle \qquad (R_0)$$

The initialization statement $M : [[M]] = \delta^*(q_0, \varepsilon)$ requires (similar to the proof above) that $M[i,j] \equiv (i,j) \in \delta^*(q_0, \varepsilon)$.

$$\begin{array}{ll} & (i,j) \in \delta^*(q_0, \varepsilon) \\ \equiv & \quad \{ \ q_0 = (0,0), \text{ def } \varepsilon\text{-closure } \} \\ & (i,j) \in \langle \textbf{set } l \ : \ 0 \leq l \leq (k \downarrow m) \ : \ (l,l) \rangle \\ \equiv & \quad \{ \ 0 \leq i \leq k \ \wedge \ 0 \leq j \leq m \ \} \\ & i = j \end{array}$$

Thus P_1 is properly initialized when the following holds:

$$\langle \forall i \ : \ 0 \leq i \leq k \ : \ \langle \forall j \ : \ 0 \leq j \leq m \ : \ M[i,j] \equiv i = j \rangle \rangle$$

This states that initially all states on the diagonal starting at state $(0,0)$ are active and all others are not.

If InitArray is given by:
$InitArray : \mathbb{B}^{m+1}$
$InitArray() = (true, false_1, \ldots, false_m)$
then the following program fragment initializes M to the unity matrix.

$M[0] := InitArray();$
$a := 0;$
do $a \neq k \rightarrow$
    $a, M[a+1] := a+1, RShiftF(M[a]);$
od

Adding these results gives the following program fragment:

if $m \leq k \rightarrow O := \{(\varepsilon, T, m)\}$
$[\!]$ $k < m \rightarrow O := \varnothing$
fi; $\{ \ P_0(u, r := \varepsilon, T) \ \}$
$M[0] := InitArray();$
$a := 0;$
do $a \neq k \rightarrow$
    $a, M[a+1] := a+1, RShiftF(M[a]);$
od;
$\{ \ P_{0..1}(u, r := \varepsilon, T) \ \}$
$u, r := \varepsilon, T;$
do $r :: cw \rightarrow \{ \ P_{0..1} \ \}$
    $S_0;$
    $\{ \ R_0? \ \}$
    $\{ \ P_0 \ \wedge \ P_1(u, r := uc, w) \ \}$

for $i \in [0..k] \rightarrow$
  if $M[i,m] \rightarrow O := O \cup \{(uc, w, i)\}$
  $[\!]$ $\neg M[i,m] \rightarrow$ skip
  fi
rof;
$\{\ P_{0..1}(u, r := uc, w)\ \}$
$u, r := uc, w$
od$\{\ P_0\ \wedge\ r = \varepsilon\ \}$
$\{\ R\ \}$

This concludes the derivation for the adjusted program skeleton. All that remains, is to derive program fragment S_0 in order to establish R_0.

2.11.1 Program fragment S_0

This section presents the derivation of program fragment S_0 which establishes postcondition R_0 on page 34.

This is essentially one step of the simulation. That is, given the current set of active states and the next input symbol, compute the next set of active states.

As discussed in the introduction, each row will be packed into a machine word and updated as a whole. So this postcondition is established row by row. The invariant therefore becomes that all rows up to but not including row a have been updated. Then whenever $a = k + 1$ the postcondition has been established, therefore the guard becomes $a \neq k + 1$ and invariant Q_0 is given by:

$$\langle \forall i\ :\ 0 \leq i < a\ :\ \langle \forall j\ :\ 0 \leq j \leq m\ :\ M[i,j] \equiv (i,j) \in \delta^*(q_0, uc) \rangle \rangle \qquad (Q_0)$$

Because matrix M should be updated row by row, investigate an increment of a by one.

$$\begin{array}{ll} & Q_0(a := a + 1) \\ \equiv & \quad \{\ \text{subst}\ \} \\ & \langle \forall i\ :\ 0 \leq i < a + 1\ :\ \langle \forall j\ :\ 0 \leq j \leq m\ :\ M[i,j] \equiv (i,j) \in \delta^*(q_0, uc) \rangle \rangle \\ \equiv & \quad \{\ \text{split off } i = a, Q_0, 0 \leq a\ \} \\ & \langle \forall j\ :\ 0 \leq j \leq m\ :\ M[a,j] \equiv (a,j) \in \delta^*(q_0, uc) \rangle \end{array}$$

This expression must be split into two cases. Case $j = 0$ and case $1 \leq j \leq m$ because the first column does not depend on any previous columns while all the others do.

- case $1 \leq j \leq m$.

$$\begin{array}{ll} & (a,j) \in \delta^*(q_0, uc) \\ \equiv & \quad \{\ \delta^*,\ 1 \leq a\ \} \\ & (a-1, j-1) \in \delta^*(q_0, u)\ \vee\ (a-1, j) \in \delta^*(q_0, u) \end{array}$$

$$
\begin{array}{ll}
 & \vee\ ((a,j-1) \in \delta^*(q_0,u)\ \wedge\ p[j-1]=c)\ \vee\ (a-1,j-1) \in \delta^*(q_0,uc) \\
\equiv & \quad \{\ Q_0\ \} \\
 & (a-1,j-1) \in \delta^*(q_0,u)\ \vee\ (a-1,j) \in \delta^*(q_0,u) \\
 & \vee\ ((a,j-1) \in \delta^*(q_0,u)\ \wedge\ p[j-1]=c)\ \vee\ M[a-1,j-1] \\
\equiv & \quad \{\ \bullet I_0 : \langle \forall c,j\ :\ c \in \Sigma\ \wedge\ 1 \le j \le m\ :\ B[c,j] \equiv p[j-1]=c \rangle, \\
 & \qquad \bullet Q_1 : \langle \forall i,j\ :\ a \le i \le k\ \wedge\ 0 \le j \le m\ :\ M[i,j] \equiv (i,j) \in \delta^*(q_0,u) \rangle \\
 & \quad \} \\
 & (a-1,j-1) \in \delta^*(q_0,u)\ \vee\ (a-1,j) \in \delta^*(q_0,u) \\
 & \vee\ (M[a,j-1]\ \wedge\ B[c,j])\ \vee\ M[a-1,j-1] \\
\equiv & \quad \{\ \bullet Q_2 : \langle \forall j\ :\ 0 \le j \le m\ :\ D[j] \equiv (a-1,j) \in \delta^*(q_0,u) \rangle\ \} \\
 & D[j-1]\ \vee\ D[j]\ \vee\ (M[a,j-1]\ \wedge\ B[c,j])\ \vee\ M[a-1,j-1]
\end{array}
$$

Invariant I_0 will become a system invariant and is proven in Section 2.11.3. Invariants Q_0, Q_1 and Q_2 are required to efficiently establish the value of expression $(a,j) \in \delta^*(q_0,uc)$. Invariant Q_1 states that matrix M stores the values of the previously active states (before processing the next character) for row a and above, which should be valid already (this will be verified later on). Invariant Q_2 is required because the old value of the previous row is overwritten by the new value.

- case $j = 0$.

$$
\begin{array}{ll}
 & (a,j) \in \delta^*(q_0,uc) \\
\equiv & \quad \{\ j=0, \delta^*\ \} \\
 & (a-1,0) \in \delta^*(q_0,u) \\
\equiv & \quad \{\ \bullet Q_2\ \} \\
 & D[0]
\end{array}
$$

The two cases above can be combined into a single expression E_0 which satisfies Z_1:

$$
\begin{array}{l}
E_0[0] \equiv D[0]\ \wedge \\
\langle \forall j\ :\ 1 \le j \le m\ :\ E_0[j] \equiv \\
\quad D[j-1]\ \vee\ D[j]\ \vee\ (M[a,j-1]\ \wedge\ B[c,j])\ \vee\ M[a-1,j-1] \rangle
\end{array}
\qquad (Z_1)
$$

The following Hoare triple was proven:
$\{\ Q_0\ \wedge\ Q_1\ \wedge\ Q_2\ \}\ M[a] := E_0\ \{\ Q_0(a := a+1)\ \wedge\ Q_1(a := a+1)\ \}$. Note, Q_1 can no longer be valid, since it is violated by the assignment to $M[a]$. But, $Q_1(a := a+1)$ does hold because it is widening.

Invariant Q_2 requires that array D is set to the value of the next row as is derived below.

$$
\begin{array}{ll}
 & Q_2(a := a+1) \\
\equiv & \quad \{\ \text{subst}\ \} \\
 & \langle \forall j\ :\ 0 \le j \le m\ :\ D[j] \equiv (a,j) \in \delta^*(q_0,u) \rangle
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{\ Q_1\ \} \\
& \langle \forall j\ :\ 0 \le j \le m\ :\ D[j] \equiv M[a,j] \rangle
\end{array}
$$

The derivation above shows the validity of the following Hoare triple:
$\{\ Q_1\ \}\ D := M[a]\ \{\ Q_2(a := a+1)\ \}$.

When comparing this Hoare triple with the Hoare triple from the derivation of Q_0 it becomes evident that these two assignments ($M[a] := E_0$ and $D := M[a]$) cannot be executed consecutively and hence must be executed concurrently ($M[a], D := E_0, M[a]$).

Now take a look at the initialization of invariants Q_0, Q_1 and Q_2.

Statement $a := 0$ will correctly initialize invariant Q_0. But, because row zero of the NFA does not have a 'previous' row (i.e. a row which corresponds with a lower distance), the update of the first row is different from all other rows. By initializing a to one the first row must be updated separately from the other rows and the code in the final algorithm is a bit cleaner.

$$
\begin{array}{ll}
& Q_0(a := 1) \\
\equiv & \{\ \text{subst}\ \} \\
& \langle \forall i\ :\ 0 \le i < 1\ :\ \langle \forall j\ :\ 0 \le j \le m\ :\ M[i,j] \equiv (i,j) \in \delta^*(q_0, uc) \rangle \rangle \\
\equiv & \{\ \text{singleton domain}\ \} \\
& \langle \forall j\ :\ 0 \le j \le m\ :\ M[0,j] \equiv (0,j) \in \delta^*(q_0, uc) \rangle
\end{array}
$$

The expression $(0,j) \in \delta^*(q_0, uc)$ is true in case $j = 0$ by definition. In case $1 \le j \le m$ derive:

$$
\begin{array}{ll}
& (0,j) \in \delta^*(q_0, uc) \\
\equiv & \{\ \delta^*\ \} \\
& (0,j-1) \in \delta^*(q_0, u)\ \wedge\ p[j-1] = c \\
\equiv & \{\ Q_2(a := 1), I_0\ \} \\
& D[j-1]\ \wedge\ B[c,j]
\end{array}
$$

Invariant Q_0 is initialized properly by the assignment: $\{\ Q_2(a := 1)?\ \}\ M[0] := E_1\ \{\ Q_0\ \}$ where E_1 satisfies Z_2 as defined next:

$$E_1[0]\ \wedge\ \langle \forall j\ :\ 1 \le j \le m\ :\ E_1[j] \equiv (D[j-1]\ \wedge\ B[c,j]) \rangle \qquad (Z_2)$$

Therefore look at initialization of invariant Q_2.

$$
\begin{array}{ll}
& Q_2(a := 1) \\
\equiv & \{\ \text{subst}\ \} \\
& \langle \forall j\ :\ 0 \le j \le m\ :\ D[j] \equiv (0,j) \in \delta^*(q_0, u) \rangle \\
\equiv & \{\ Q_1(a := 0)\ \} \\
& \langle \forall j\ :\ 0 \le j \le m\ :\ D[j] \equiv M[0,j] \rangle
\end{array}
$$

This is established by array assignment $D := M[0]$ provided that $Q_1(a := 0)$ holds, which follows from invariant $P_1 : [[M]] = \delta^*(q_0, u)$.

$\{\ P_1\ \}$
$\{\ Q_1(a := 0)\ \}$
$D := M[0];$
$\{\ Q_{1..2}(a := 1)\ \}$
$M[0] := E_1?;$
$\{\ Q_{0..2}(a := 1)\ \}$
$a := 1;$
$\{\ Q_{0..2}\ \}$
do $a \neq k + 1 \rightarrow \{\ Q_{0..2}\ \}$
$M[a], D := E_0?, M[a];$
$\{\ Q_{0..2}(a := a + 1)\ \}$
$a := a + 1$
od$\{\ Q_0\ \wedge\ a = k + 1\ \}$
$\{\ R_0\ \}$

Now all that remains to be done to prove correctness of program fragment S_0 is to establish the value of expressions E_0, E_1 and system invariant I_0. Expressions E_0 and E_1 are computed in the following section and invariant I_0 is established in Section 2.11.3.

2.11.2 Computing Bit-parallel expressions E_0 and E_1

In this section expressions are derived for E_0 and E_1 such that they satisfy Z_1 and Z_2 respectively.

Formula Z_2 is the shortest, therefore expression E_1 is computed first.

Recall assertion Z_2 (see page 37) which states the required value for E_1. The goal is to compute the value for each of these elements at once. So expression $D[j-1]\ \wedge\ B[c, j]$ must be generalized to an entire array. So an expression in D and $B[c]$ is required.

This requires several operations on arrays. As can be seen from expression $D[j-1]\ \wedge\ B[c, j]$ an *And* operation and a *Shift* operation are required. The *Shift* operation is required since an *And* of arrays D and $B[c]$ would yield $D[j]\ \wedge\ B[c, j]$ instead of $D[j-1]$. Note that these operations correspond to the bitwise *and* and *shift* operations on machine words. For definitions of these functions and more info about derivation of bit parallel algorithms refer to Section B.

E_1
$=$ $\{$ Expand E_1 $\}$
$(E_1[0], E_1[1], \ldots, E_1[m])$
$=$ $\{\ Z_2\ \}$
$(true, D[0] \wedge B[c, 1], \ldots, D[m-1] \wedge B[c, m])$
$=$ $\{\ \wedge\ \}$

$$
\begin{array}{ll}
 & (true \wedge true, D[0] \wedge B[c,1], \ldots, D[m-1] \wedge B[c,m]) \\
= & \quad \{\ And\ \} \\
 & And(\ (true, D[0], \ldots, D[m-1])\ ,\ (true, B[c,1], \ldots, B[c,m])\ ) \\
= & \quad \{\ RShiftT, \text{ simplify } \} \\
 & And(RShiftT(D),\ (true, B[c,1], \ldots, B[c,m])\ ) \\
= & \quad \{\ \bullet\ B[c,0] \equiv true, \text{ simplify } \} \\
 & And(RShiftT(D), B[c])
\end{array}
$$

As the derivation above shows, $B[c,0]$ should equal $true$ for all c. Therefore redefine I_0 as:

$$\langle \forall c, j\ :\ c \in \Sigma\ \wedge\ 1 \leq j \leq m\ :\ B[c,j] = p[j-1] = c \rangle\ \wedge\ \langle \forall c\ :\ c \in \Sigma\ :\ B[c,0] = True \rangle \qquad (I_0)$$

This does not invalidate any of the previous proofs, since none of them ever references value $B[c,0]$.

The derivation above shows that $E_1(D, B[c]) = And(RShiftT(D), B[c])$. Thus this expression can be substituted in the assignment to $M[0]$.

Next recall formula Z_1 on page 36 which specifies expression E_0.

$$
\begin{array}{ll}
 & E_0 \\
= & \quad \{\ \text{Expand}\ \} \\
 & (E_0[0], E_0[1], \ldots, E_0[m]) \\
= & \quad \{\ Z_1\ \} \\
 & (D[0] \\
 & , D[0] \vee D[1] \vee (M[a,0] \wedge B[c,1]) \vee M[a-1,0] \\
 & \vdots \\
 & , D[m-1] \vee D[m] \vee (M[a,m-1] \wedge B[c,m]) \vee M[a-1,m-1] \\
 &) \\
= & \quad \{\ \vee\ \} \\
 & (false \vee D[0] \vee false \vee false \\
 & , D[0] \vee D[1] \vee (M[a,0] \wedge B[c,1]) \vee M[a-1,0] \\
 & \vdots \\
 & , D[m-1] \vee D[m] \vee (M[a,m-1] \wedge B[c,m]) \vee M[a-1,m-1] \\
 &) \\
= & \quad \{\ Or\ \} \\
 & Or((false, D[0], \ldots, D[m-1]), \\
 & \quad (D[0], \ldots, D[m]), \\
 & \quad (false, M[a,0] \wedge B[c,1], \ldots, M[a,m-1] \wedge B[c,m]), \\
 & \quad (false, M[a-1,0], \ldots, M[a-1,m-1]) \\
 &) \\
= & \quad \{\ RShiftF, \text{ simplify } \}
\end{array}
$$

$$
\begin{array}{ll}
 & Or(RShiftF(D), D, \\
 & \quad (false, M[a,0] \wedge B[c,1], \ldots, M[a,m-1] \wedge B[c,m]) \ , \\
 & \quad RShiftF(M[a-1]) \\
 &) \\
= & \quad \{ \ \wedge \ \} \\
 & Or(RShiftF(D), D, \\
 & \quad (false \wedge B[c,0], M[a,0] \wedge B[c,1], \ldots, M[a,m-1] \wedge B[c,m]), \\
 & \quad RShiftF(M[a-1]) \\
 &) \\
= & \quad \{ \ And \ \} \\
 & Or(RShiftF(D), D, \\
 & \quad And(\ (false, M[a,0], \ldots, M[a,m-1]) \ , \ (B[c,0], B[c,1], \ldots, B[c,m]) \ ), \\
 & \quad RShiftF(M[a-1]) \\
 &) \\
= & \quad \{ \ RShiftF, \text{ simplify } \} \\
 & Or(RShiftF(D), D, And(RShiftF(M[a]), B[c]), RShiftF(M[a-1]))
\end{array}
$$

From this derivation follows that E_0 equals:
$Or(RShiftF(D), D, And(RShiftF(M[a]), B[c]), RShiftF(M[a-1]))$. Substituting these results in the previous program fragment results in the final version for program fragment S_0:

$\{ \ P_1 \ \}$
$\{ \ Q_1(a := 0) \ \}$
$D := M[0];$
$\{ \ Q_{1..2}(a := 1) \ \}$
$M[0] := And(RShiftT(D), B[c]);$
$\{ \ Q_{0..2}(a := 1) \ \}$
$a := 1;$
$\{ \ Q_{0..2} \ \}$
do $a \neq k+1 \rightarrow \{ \ Q_{0..2} \ \}$
    $M[a], D := Or(RShiftF(D), D, And(RShiftF(M[a]), B[c]), RShiftF(M[a-1])), M[a];$
    $\{ \ Q_{0..2}(a := a+1) \ \}$
    $a := a+1$
od$\{ \ Q_0 \ \wedge \ a = k+1 \ \}$
$\{ \ R_0 \ \}$

2.11.3 Final Algorithm

System invariant I_0 can be initialized by the following program fragment:

for $c \in \Sigma \rightarrow$
    $B[c,0] := true;$
    for $j \in [1,m] \rightarrow$
        $B[c,j] := p[j-1] = c$

rof
rof

Combining all derived program fragments results in the following final algorithm.

Algorithm 2.11.12(PE, SP, EP, BM, AA, RW)

$R : O = \langle \textbf{set}\ l, v, r\ :\ lvr = T\ \wedge\ dst(v,p) \leq k\ :\ (lv, r, dst(v,p))\rangle$
$P_0 : ur = T\ \wedge\ O = \langle \textbf{set}\ x, y, z\ :\ xyz = T\ \wedge\ xy \leq_p u\ \wedge\ dst(y,p) \leq k\ :\ (xy, z, dst(y,p))\rangle$
$P_1 : [[M]] = \delta^*(q_0, u)$
$R_0 : \langle \forall i\ :\ 0 \leq i \leq k\ :\ \langle \forall j\ :\ 0 \leq j \leq m\ :\ M[i,j] \equiv (i,j) \in \delta^*(q_0, uc)\rangle\rangle$
$Q_0 : \langle \forall i\ :\ 0 \leq i < a\ :\ \langle \forall j\ :\ 0 \leq j \leq m\ :\ M[i,j] \equiv (i,j) \in \delta^*(q_0, uc)\rangle\rangle$
$Q_1 : \langle \forall i, j\ :\ a \leq i \leq k\ \wedge\ 0 \leq j \leq m\ :\ M[i,j] \equiv (i,j) \in \delta^*(q_0, u)\rangle$
$Q_2 : \langle \forall j\ :\ 0 \leq j \leq m\ :\ D[j] \equiv (a-1, j) \in \delta^*(q_0, u)\rangle$

for $c \in \Sigma \rightarrow$
    $B[c, 0] := true$;
    for $j \in [1, m] \rightarrow$
        $B[c, j] := p[j-1] = c$
    rof
rof;
$\{\ I_0\ \}$
if $m \leq k \rightarrow O := \{(\varepsilon, T, m)\}$
$[\!]$ $k < m \rightarrow O := \varnothing$
fi; $\{\ P_0(u, r := \varepsilon, T)\ \}$
$M[0] := InitArray()$;
$a := 0$;
do $a \neq k \rightarrow$
    $a, M[a+1] := a+1, RShiftF(M[a])$;
od;
$\{\ P_{0..1}(u, r := \varepsilon, T)\ \}$
$u, r := \varepsilon, T$;
do $r :: cw \rightarrow \{\ P_{0..1}\ \}$
    $\{\ Q_1(a := 0)\ \}$
    $D := M[0]$;
    $\{\ Q_{1..2}(a := 1)\ \}$
    $M[0] := And(RShiftT(D), B[c])$;
    $\{\ Q_{0..2}(a := 1)\ \}$
    $a := 1$;
    $\{\ Q_{0..2}\ \}$
    do $a \neq k+1 \rightarrow \{\ Q_{0..2}\ \}$
        $M[a], D := Or(RShiftF(D), D, And(RShiftF(M[a]), B[c]), RShiftF(M[a-1])), M[a]$;
        $\{\ Q_{0..2}(a := a+1)\ \}$
        $a := a+1$
    od$\{\ Q_0\ \wedge\ a = k+1\ \}$
    $\{\ R_0\ \}\ \{\ P_0\ \wedge\ P_1(u, r := uc, w)\ \}$
    for $i \in [0..k] \rightarrow$

```
        if M[i,m] → O := O ∪ {(uc, w, i)}
        ▯ ¬M[i,m] → skip
        fi
    rof;
    { P_{0..1}(u, r := uc, w) }
    u, r := uc, w
od{ P_0 ∧ r = ε }
{ R }
```

A final remark: in the literature this algorithm was encountered in a slightly different form. The version presented here simulates the entire approximate pattern matching automaton. The version found in the literature does not simulate all states below the first full diagonal (through state q_0) because they do not influence the final answer anyway (provided $k \leq m$). When using bit parallelism, not simulating those states has no performance advantage and only serves to complicate the abstraction function (since some bits in the words are then ignored).

It is an interesting difference, since the only difference lies in the initialization and not simulating the states below the first diagonal make the initialization more costly! In this text, the states on the full diagonal through q_0 are the only states initialized to *true*, in the other version, all states below the diagonal were also initialized to *true*.

2.12 No Difference count

The algorithm in Section 2.9 can be simplified by only reporting all occurrences but not reporting the number of differences between an occurrence and the pattern.

This restriction is formalized by dropping $dst(v, p)$ from the formulas and modifying the following loop: (this is the inner for loop with the mismatch count removed)

```
for i ∈ [0..k] →
    if (i,m) ∈ [[M]] → O := O ∪ {(uc, w)}
    ▯ (i,m) ∉ [[M]] → skip
    fi
rof;
```

This loop computes $\langle \textbf{set}\ i\ :\ 0 \leq i \leq k\ \wedge\ (i,m) \in [[M]]\ :\ (uc, w)\rangle$.

$$
\begin{array}{ll}
 & \langle \textbf{set}\ i\ :\ 0 \leq i \leq k\ \wedge\ (i,m) \in [[M]]\ :\ (uc, w)\rangle \\
\equiv & \quad \{\ (uc, w) \text{ independent of } i\ \} \\
 & \begin{cases} (uc, w) & \text{if } \langle \exists i\ :\ 0 \leq i \leq k\ :\ (i,m) \in [[M]]\rangle \\ \varnothing & \text{if } \neg\langle \exists i\ :\ 0 \leq i \leq k\ :\ (i,m) \in [[M]]\rangle \end{cases} \\
\equiv & \quad \{\ \text{Property 3.3.3 on page 81}\ \} \\
 & \begin{cases} (uc, w) & \text{if } (k,m) \in [[M]] \\ \varnothing & \text{if } (k,m) \notin [[M]] \end{cases}
\end{array}
$$

This leads to the following algorithm

Algorithm 2.12.13(PE, SP, EP, BM, AA, ND)

$R : O = \langle \textbf{set } l, v, r \ : \ lvr = T \ \wedge \ dst(v, p) \leq k \ : \ (lv, r) \rangle$
$P_0 : ur = T \ \wedge \ O = \langle \textbf{set } x, y, z \ : \ xyz = T \ \wedge \ xy \leq_p u \ \wedge \ dst(y, p) \leq k \ : \ (xy, z) \rangle$
$P_1 : [[M]] = \delta^*(q_0, u)$

if $m \leq k \rightarrow O := \{(\varepsilon, T)\}$
$[\!]$ $k < m \rightarrow O := \varnothing$
fi; $\{ \ P_0(u, r := \varepsilon, T) \ \}$
$M : [[M]] = \delta^*(q_0, \varepsilon)$;
$\{ \ P_{0..1}(u, r := \varepsilon, T) \ \}$
$u, r := \varepsilon, T$;
do $r :: cw \rightarrow \{ \ P_{0..1} \ \}$
    $M : [[M]] = \delta^*(q_0, uc)$;
    $\{ \ P_0 \ \wedge \ P_1(u, r := uc, w) \ \}$
    if $(k, m) \in [[M]] \rightarrow O := O \cup \{(uc, w)\}$
    $[\!]$ $(k, m) \notin [[M]] \rightarrow$ skip
    fi;
    $\{ \ P_{0..1}(u, r := uc, w) \ \}$
    $u, r := uc, w$
od$\{ \ P_0 \ \wedge \ r = \varepsilon \ \}$
$\{ \ R \ \}$

2.13 A restriction on the basic automaton

In this section some consequences of simulating a slightly smaller automaton are investigated. At the end of this section the formulas of the algorithm in Section 2.12 are updated to reflect this restriction.

Suppose $k < m$. The diagonal containing the starting state is always active due to the presence of the ε-transitions and the Σ loop on the start. Therefore none of the states below the first full diagonal can influence whether any of the end-states are active or not. These states can be safely ignored.

Holub indicates the possibility to not simulate any states above the last full diagonal (diagonal $m - k$) when the exact number of errors is not relevant [Hol96]. This section explores the consequences of removing these states.

Suppose the states above diagonal $m - k$ are not simulated. Since diagonal $m - k$ depends on diagonal $m - k + 1$ these states must either be considered active or inactive. Suppose a state $(j, m - k + 1 + j)$ with $0 \leq j \leq k - 1$ on diagonal $m - k + 1$ is active. When a character is processed, state $(j + 1, m - k + j + 1)$ on diagonal $m - k$ becomes active and therefore the final state (k, m) as well (due to the ε-transitions). In other words if the upper right corner of

the automaton is not simulated, all states on diagonal $m-k+1$ must be considered inactive, otherwise each text position will produce a (likely invalid) match.

Fixing all states on diagonal $m-k+1$ to inactive does not come without a price however. Some occurrences will be lost. Below an attempt is made to characterize the lost occurrences to inspect the severity of the loss.

Suppose state (j, m) (with $0 \leq j \leq k$) is active, then there exists a path from the start state to (j, m). Suppose a part of the path consists out of states and transitions which are to be removed. Let $(i, m-k+i) \xrightarrow{c} (i, m-k+1+i)$ be the first transition which should be removed (note, the first transition which should be removed is always a horizontal transition). If the path between $(i, m-k+i)$ and (j, m) does not contain any vertical transitions, then there exists a path to state (k, m) which ignores the to be removed states altogether.

$$
\begin{aligned}
hc &= \text{horizontal transitions} \\
dc &= \text{diagonal } (\Sigma) \text{ transitions} \\
ec &= \varepsilon - \text{transitions} \\
vc &= \text{vertical transitions}
\end{aligned}
$$

The following properties follow directly from the automaton and the assumption that $vc = 0$:

$$m - (m-k+i) = hc + dc + ec \tag{2.2}$$

$$j - i = dc + ec \tag{2.3}$$

The number of horizontal transitions equals the vertical distance between (j, m) and (k, m). Proof:

$$
\begin{aligned}
& hc \\
= & \quad \{ \text{ (2.2), calc } \} \\
& k - i - (dc + ec) \\
= & \quad \{ \text{ (2.3) } \} \\
& k - i - (j - i) \\
= & \quad \{ \text{ calc } \} \\
& k - j
\end{aligned}
$$

By replacing all these horizontal transitions with diagonal transitions the final state becomes (k, m) instead of (j, m). Note that in the newly constructed path there are no horizontal or vertical transitions after state $(i, m-k+i)$ and therefore this path does not contain any removed transitions (it follows the $m-k^{th}$ diagonal). This concludes the proof.

In the case that the path from $(i, m-k+i)$ and (j, m) does contain vertical transitions, then the match may or may not be found by the reduced automaton.

For an example of which states are removed in the reduced automaton refer to Figure 2.2.

I suspect that all those occurrences are lost for which holds:

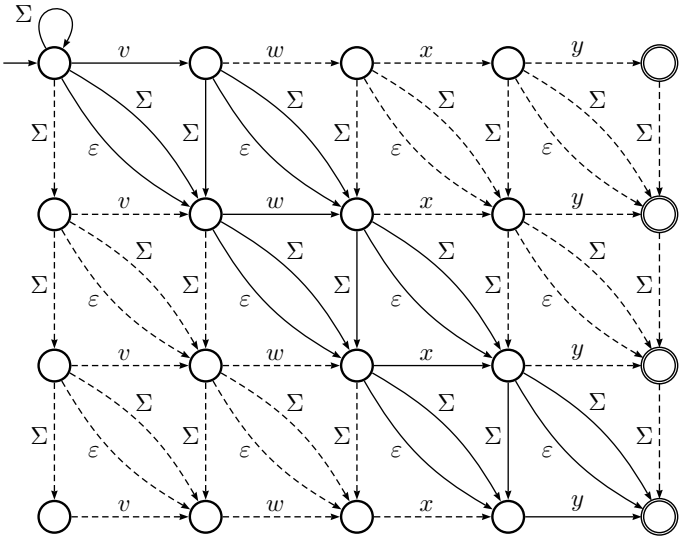


Figure 2.2: NFA for matching string *vxwy* with a maximum distance of three

- The only way they are accepted is by a path with a suffix through the removed upper right triangle.
- That suffix contains a vertical transition.

Comparing all the strings accepted by reduced and full automata for some randomly generated pattern (of length at most five and with k at most three) shows the following results: all matches which are not accepted by the reduced automaton, but which are accepted by the full automaton have insertions added to a prefix which is accepted by both automata. These tests were not representative enough however to be able to draw any trustworthy conclusion from the results.

Suppose LM is the set of all matches which are lost by this restriction. Then the restricted algorithm becomes:

Algorithm 2.13.14(PE, SP, EP, BM, AA, ND, AR)

$R : O = \langle \textbf{set}\ l, v, r\ :\ lvr = T\ \wedge\ dst(v,p) \leq k\ :\ (lv, r)\rangle \backslash LM$
$P_0 : ur = T\ \wedge\ O = \langle \textbf{set}\ x, y, z\ :\ xyz = T\ \wedge\ xy \leq_p u\ \wedge\ dst(y,p) \leq k\ :\ (xy, z)\rangle \backslash LM$
$P_1 : [[M]] = \delta^*(q_0, u)$

if $m \leq k \rightarrow O := \{(\varepsilon, T)\}$
$[\!]\ \ k < m \rightarrow O := \varnothing$
fi; $\{\ P_0(u, r := \varepsilon, T)\ \}$
$M : [[M]] = \delta^*(q_0, \varepsilon)$;
$\{\ P_{0..1}(u, r := \varepsilon, T)\ \}$
$u, r := \varepsilon, T$;
do $r :: cw \rightarrow \{\ P_{0..1}\ \}$
$\quad M : [[M]] = \delta^*(q_0, uc)$;
$\quad \{\ P_0\ \wedge\ P_1(u, r := uc, w)\ \}$

if $(k,m) \in [[M]] \rightarrow O := O \cup \{(uc, w)\}$
$[\!]$ $(k,m) \notin [[M]] \rightarrow$ skip
fi;
$\{\ P_{0..1}(u, r := uc, w)\ \}$
$u, r := uc, w$
od$\{\ P_0\ \wedge\ r = \varepsilon\ \}$
$\{\ R\ \}$

2.14 Diagonal-wise bit parallel simulation

This section continues from the algorithm presented in Section 2.13.

When simulating a NFA with row-wise bit parallelism there exists a dependency between the different rows of the automaton (due to ε-transitions). More precisely, the value of row i in the j^{th} iteration depends on the value of row $i-1$ in both the j^{th} and the $(j-1)^{th}$ iteration. This dependency forces a linear time ($\mathcal{O}(k)$) update of the active states of the automaton. When the states of the automaton are packed diagonal-wise this dependency is removed, since the value of a diagonal after processing the next input character only depends on the previous values of the automaton's diagonals.

The diagonal-wise bit parallel simulation of the NFA is derived in this Section. Most definitions of Section 2.11 remain valid.

The diagonal-wise bit parallel simulation algorithm only checks if a substring matches with a distance at most k. The exact distance between the match and the pattern is not reported. This is because only the full diagonals are represented and therefore the only final state is state (k,m). This does not really matter since if $k < m$ then state (k,m) is active whenever any final state is active (see property 3.3.3 on page 81).

Let M be a boolean matrix: $M : Matrix[0..(m-k), 0..k] : boolean$. The following abstraction function shows how the active states of the automaton are represented by matrix M:

$$[[M]] = \langle \textbf{set}\ i, d\ :\ 0 \leq i \leq k\ \wedge\ 0 \leq d \leq m-k\ \wedge\ \neg M[d,i]\ :\ (i, i+d) \rangle \tag{2.4}$$

This abstraction states that if the i^{th} state on the d^{th} diagonal is active, then $M[d,i]$ is false.

First consider the guard $(k,m) \in [[M]]$. By applying the concrete abstraction function, this statement can be refined:

$$\begin{array}{ll} & (k,m) \in [[M]] \\ \equiv & \quad \{\ [[M]]\ \} \\ & (k,m) \in \langle \textbf{set}\ i, d\ :\ 0 \leq i \leq k\ \wedge\ 0 \leq d \leq m-k\ \wedge\ \neg M[d,i]\ :\ (i, i+d) \rangle \\ \equiv & \quad \{\ \text{calculus}\ \} \\ & \neg M[m-k, k] \end{array}$$

Next, consider the statement $M : [[M]] = \delta^*(q_0, \varepsilon)$. For $0 \leq i \leq k$ and $0 \leq d \leq m-k$:

$$
\begin{array}{ll}
 & M[d,i] \\
\equiv & \quad \{\ [[M]]\ \} \\
 & (i,i+d) \notin [[M]] \\
\equiv & \quad \{\ M : [[M]] = \delta^*(q_0,\varepsilon)\ \} \\
 & (i,i+d) \notin \delta^*(q_0,\varepsilon) \\
\equiv & \quad \{\ \delta^*\ \} \\
 & (i,i+d) \notin \varepsilon\text{-closure}(q_0) \\
\equiv & \quad \{\ (i,i+d) \in \varepsilon\text{-closure}(q_0) \equiv d = 0\ \} \\
 & d \neq 0
\end{array}
$$

This derivation shows that M can be initialized by any program fragment which establishes: $\langle \forall i,d \ :\ 0 \leq i \leq k \ \wedge\ 0 \leq d \leq m-k \ :\ M[d,i] \equiv d \neq 0 \rangle$. In other words, only the states on the first diagonal (starting in state q_0) are initialized to $false$, all other states are initialized to $true$.

Last, but not least, consider statement $M : [[M]] = \delta^*(q_0, uc)$.

Note that due to the Σ-transition on state q_0, the first diagonal ($M[0]$) is always active, regardless of the processed input text. Diagonal $M[0]$ will not be explicitly calculated and stored by the algorithm. Since only the full diagonals are considered, (partial) diagonal $m-k+1$ should not influence diagonal $m-k$, therefore define $M[m-k+1]$ as the $true$ vector (see Section 2.13 on page 43 for details).

This derivation is split into two cases both with d in the range $[1..(m-k)]$.

The first case: $(i = 0)$:

$$
\begin{array}{ll}
 & M[d,0] \\
\equiv & \quad \{\ [[M]], M : [[M]] = \delta^*(q_0,uc)\ \} \\
 & (0,d) \notin \delta^*(q_0,uc) \\
\equiv & \quad \{\ \delta^*, 1 \leq d\ \} \\
 & \neg((0,d-1) \in \delta^*(q_0,u) \ \wedge\ p[d-1] = c) \\
\equiv & \quad \{\ \text{De Morgan}\ \} \\
 & ((0,d-1) \notin \delta^*(q_0,u)) \ \vee\ p[d-1] \neq c \\
\equiv & \quad \{\ \bullet Z_1 : \langle \forall d \ :\ 1 \leq d \leq m \ :\ X[d,0] \equiv ((0,d-1) \notin \delta^*(q_0,u)) \ \vee\ p[d-1] \neq c \rangle\ \} \\
 & X[d,0]
\end{array}
$$

The second case: $(1 \leq i)$

$$
\begin{array}{ll}
 & M[d,i] \\
\equiv & \quad \{\ [[M]], M : [[M]] = \delta^*(q_0,uc)\ \} \\
 & (i,i+d) \notin \delta^*(q_0,uc) \\
\equiv & \quad \{\ \text{def } \delta^*, \text{ De Morgan}\ \}
\end{array}
$$

$$
\begin{array}{ll}
& (i-1, i+d-1) \notin \delta^*(q_0, u) \wedge \\
& (i-1, i+d) \notin \delta^*(q_0, u) \wedge \\
& ((i, i+d-1) \notin \delta^*(q_0, u) \ \vee \ p[i+d-1] \neq c) \wedge \\
& (i-1, i+d-1) \notin \delta^*(q_0, uc) \\
\equiv & \quad \{ \ \bullet Z_0 : \langle \forall i, d \ : \ 1 \leq i \leq k \ \wedge \ 1 \leq d \leq m-k \ : \ X[d,i] \equiv (i-1, i+d-1) \notin \delta^*(q_0, u) \wedge \\
& \quad (i-1, i+d) \notin \delta^*(q_0, u) \ \wedge \ ((i, i+d-1) \notin \delta^*(q_0, u) \ \vee \ p[i+d-1] \neq c) \rangle \\
& \quad \} \\
& X[d,i] \ \wedge \ (i-1, i+d-1) \notin \delta^*(q_0, uc) \\
\equiv & \quad \{ \text{ Property 3.3.2 on page 80 } \} \\
& X[d,i] \ \wedge \ (i-1, i+d-1) < \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ (l, l+d) \in \delta^*(q_0, uc) \ : \ (l, l+d) \rangle \\
\equiv & \quad \{ \ (a, a+d) < (b, b+d) \equiv a < b \ \} \\
& X[d,i] \ \wedge \ (i-1 < \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ (l, l+d) \in \delta^*(q_0, uc) \ : \ l \rangle) \\
\equiv & \quad \{ \ \downarrow, \text{ Property 3.3.5 on page 83, } g(j, l, uc) = \neg(X[d,l]) \ \} \\
& X[d,i] \ \wedge \ (i-1 < \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ \neg(X[d,l]) \ : \ l \rangle) \\
\equiv & \quad \{ \text{ calc } \} \\
& X[d,i] \ \wedge \ (i \leq \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ \neg(X[d,l]) \ : \ l \rangle) \\
\equiv & \quad \{ \text{ case analysis } \} \\
& \left\{ \begin{array}{l}
X[d,i] \ \wedge \ false \text{ if } i > \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ \neg(X[d,l]) \ : \ l \rangle \\
X[d,i] \ \wedge \ true \ \text{ if } i = \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ \neg(X[d,l]) \ : \ l \rangle \\
X[d,i] \ \wedge \ true \ \text{ if } i < \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ \neg(X[d,l]) \ : \ l \rangle
\end{array} \right. \\
\equiv & \quad \{ \ x \ \wedge \ false = false \\
& \quad \text{case } i = \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ \neg(X[d,l]) \ : \ l \rangle \text{ then } X[d,i] \equiv false \\
& \quad \text{case } i < \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ \neg(X[d,l]) \ : \ l \rangle \text{ then } X[d,i] \equiv true \\
& \quad \} \\
& \left\{ \begin{array}{l}
false \text{ if } i > \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ \neg(X[d,l]) \ : \ l \rangle \\
false \text{ if } i = \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ \neg(X[d,l]) \ : \ l \rangle \\
true \ \text{ if } i < \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ \neg(X[d,l]) \ : \ l \rangle
\end{array} \right. \\
\equiv & \quad \{ \ calc \ \} \\
& i < \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ \neg(X[d,l]) \ : \ l \rangle \\
\equiv & \quad \{ \ \bullet Flood : \mathbb{B}^{k+1} \to \mathbb{B}^{k+1} \\
& \quad Flood((a_0, a_1, \ldots, a_k)) = (0 < b, 1 < b, \ldots, k < b) \\
& \quad \text{where } b = \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ \neg a_l \ : \ l \rangle \\
& \quad \} \\
& Flood(X[d])[i]
\end{array}
$$

The above computations show the required values for $M[d,i]$:

$$M[d,i] \equiv Flood(X[d])[i] \qquad \text{if } 1 \leq i \leq k \tag{2.5}$$

$$M[d,0] \equiv X[d,0] \qquad \text{if } i = 0 \tag{2.6}$$

This would look a lot nicer if $X[d,0]$ is equivalent to $Flood(X[d])[0]$, which fortunately is true (see proof below).

$$
\begin{array}{ll}
 & Flood(X[d])[0] \\
\equiv & \quad \{ \text{ specification of } Flood \ \} \\
 & 0 < \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ \neg X[d] \ : \ l \rangle \\
\equiv & \quad \{ \text{ caseanalysis } \} \\
 & \begin{cases} false \text{ if } \neg X[d] \\ true \ \text{ if } X[d] \end{cases} \\
\equiv & \quad \{ \text{ calc } \} \\
 & X[d]
\end{array}
$$

This shows that formulas 2.5 and 2.6 can be simplified to the following (for $0 \leq i \leq k$):

$$M[d,i] \equiv Flood(X[d])[i] \tag{2.7}$$

This will be used in the next section to derive a function which computes this in a bit-parallel manner.

The derivations above introduced the following two new assertions:

$$
\begin{array}{l}
\langle \forall i,d \ : \ 1 \leq i \leq k \ \wedge \ 1 \leq d \leq m-k \ : \\
\quad X[d,i] \equiv \ (i-1, i+d-1) \notin \delta^*(q_0,u) \\
\qquad\qquad \wedge \ (i-1,i+d) \notin \delta^*(q_0,u) \\
\qquad\qquad \wedge \ ((i,i+d-1) \notin \delta^*(q_0,u) \ \vee \ p[i+d-1] \neq c) \\
\rangle
\end{array} \tag{Z_0}
$$

$$\langle \forall d \ : \ 1 \leq d \leq m-k \ : \ X[d,0] \equiv (0,d-1) \notin \delta^*(q_0,u) \ \vee \ p[d-1] \neq c \rangle \tag{Z_1}$$

To establish these recall invariant P_1 and the abstraction function and derive expressions for Z_0 and Z_1.

First consider the right hand side of the equality in Z_0:

$$
\begin{array}{ll}
 & (i-1,i+d-1) \notin \delta^*(q_0,u) \ \wedge \\
 & (i-1,i+d) \notin \delta^*(q_0,u) \ \wedge \\
 & ((i,i+d-1) \notin \delta^*(q_0,u) \ \vee \ p[i+d-1] \neq c) \\
\equiv & \quad \{ \ P_1 \ \} \\
 & (i-1,i+d-1) \notin [[M]] \ \wedge \\
 & (i-1,i+d) \notin [[M]] \ \wedge \\
 & ((i,i+d-1) \notin [[M]] \ \vee \ p[i+d-1] \neq c) \\
\equiv & \quad \{ \ [[M]] \ \} \\
 & M[d,i-1] \ \wedge \ M[d+1,i-1] \ \wedge \ (M[d-1,i] \ \vee \ p[i+d-1] \neq c) \\
\equiv & \quad \{ \ \bullet I_0 : \\
 & \quad \langle \forall c,d,i \ : \ c \in \Sigma \ \wedge \ 0 \leq d \leq m-k \ \wedge \ 0 \leq i \leq k \ : \ B[c,d,i] \equiv p[i+d-1] \neq c \rangle \\
 & \quad \} \\
 & M[d,i-1] \ \wedge \ M[d+1,i-1] \ \wedge \ (M[d-1,i] \ \vee \ B[c,d,i])
\end{array}
$$

Now consider the right hand side of the equality in Z_1:

$$
\begin{array}{ll}
 & (0, d-1) \notin \delta^*(q_0, u) \ \vee \ p[d-1] \neq c \\
\equiv & \quad \{ \ P_1 \ \} \\
 & (0, d-1) \notin [[M]] \ \vee \ p[d-1] \neq c \\
\equiv & \quad \{ \ [[M]] \ \} \\
 & M[d-1, 0] \ \vee \ p[d-1] \neq c \\
\equiv & \quad \{ \ I_0 \ \} \\
 & M[d-1, 0] \ \vee \ B[c, d, 0]
\end{array}
$$

The above computations show the required values for $X[d, i]$:

$$X[d, i] \equiv M[d, i-1] \ \wedge \ M[d+1, i-1] \ \wedge \ (M[d-1, i] \ \vee \ B[c, d, i]) \quad \text{if } 1 \leq i \leq k \quad (2.8)$$

$$X[d, 0] \equiv M[d-1, 0] \ \vee \ B[c, d, 0] \quad \text{if } i = 0 \quad (2.9)$$

Table B is specified by system invariant I_0:

$$\langle \forall c, d, i \ : \ c \in \Sigma \ \wedge \ 0 \leq d \leq m-k \ \wedge \ 0 \leq i \leq k \ : \ B[c, d, i] \equiv p[i+d-1] \neq c \rangle \quad (I_0)$$

2.14.1 Computing the bit-parallel functions

In the previous section several expressions were derived. In this section these expressions are transformed into expressions which can be implemented efficiently using bit-parallelism.

The derivation below uses specification 2.7 to compute the assignment to $M[d]$:

$$
\begin{array}{ll}
 & M[d] \\
= & \quad \{ \text{ Expand } \} \\
 & (M[d, 0], \ldots, M[d, k]) \\
= & \quad \{ \text{ (2.7) on page 49 } \} \\
 & (Flood(X[d])[0], \ldots, Flood(X[d])[k]) \\
= & \quad \{ \text{ simplify } \} \\
 & Flood(X[d])
\end{array}
$$

From this follows the proper assignment: $M[d] := Flood(X[d])$.

To understand what the Flood function does, recall its definition:
$Flood : \mathbb{B}^{k+1} \rightarrow \mathbb{B}^{k+1}$
$Flood((a_0, a_1, \ldots, a_k)) = (0 < b, 1 < b, \ldots, k < b)$
where $b = \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ \neg a_l \ : \ l \rangle$

There are two cases, either there is an argument a_i which equals $false$ or there is not. If there is no i for which argument a_i equals $false$ then:
$Flood((true_0, \ldots, true_k)) = (true_0, \ldots, true_k)$

Suppose a_i is the first argument that equals $false$, then all arguments preceding a_i (if any) equal $true$ and all arguments after (if any) a_i are either $true$ or $false$:
$Flood((true_0, \ldots, true_{i-1}, false, a_{i+1}, \ldots, a_k)) = (true_0, \ldots, true_{i-1}, false_i, \ldots, false_k)$

So basically, $\neg X[d]$ is a representation of the set of all active states on the d^{th} diagonal without having computed the ε-transitions yet. The function $Flood$ computes the ε-closure of $X[d]$.

Consider the following function:
$Increment : \mathbb{B}^{k+1} \rightarrow \mathbb{B}^{k+1}$
$Increment((true_0, \ldots, true_{i-1}, false, a_{i+1}, \ldots, a_k)) = (false_0, \ldots, false_{i-1}, true, a_{i+1}, \ldots, a_k)$
$Increment((true_0, \ldots, true_k)) = (false_0, \ldots, false_k)$

$$
\begin{array}{ll}
 & (true_0, \ldots, true_{i-1}, false_i, \ldots, false_k) \\
= & \quad \{\ RShiftT\ \} \\
 & RShiftT((true_0, \ldots, true_{i-1}, true, false_{i+1}, \ldots, false_k)) \\
= & \quad \{\ Xor\ \} \\
 & RShiftT(Xor((false_0, \ldots, false_{i-1}, true, a_{i+1}, \ldots, a_k), \\
 & \qquad\qquad (true_0, \ldots, true_{i-1}, false, a_{i+1}, \ldots, a_k))) \\
= & \quad \{\ Increment\ \} \\
 & RShiftT(Xor(Increment((true_0, \ldots, true_{i-1}, false, a_{i+1}, \ldots, a_k)), \\
 & \qquad\qquad (true_0, \ldots, true_{i-1}, false, a_{i+1}, \ldots, a_k)))
\end{array}
$$

In the case of $Flood((true_0, \ldots, true_k))$ this gives (follow same steps as above):

$$
\begin{array}{ll}
 & (true_0, \ldots, true_k) \\
= & \quad \{\ RShiftT\ \} \\
 & RShiftT((true_0, \ldots, true_k)) \\
= & \quad \{\ Xor\ \} \\
 & RShiftT(Xor(\ (false_0, \ldots, false_k)\ ,\ (true_0, \ldots, true_k)\ )) \\
= & \quad \{\ Increment\ \} \\
 & RShiftT(Xor(Increment((true_0, \ldots, true_k)), (true_0, \ldots, true_k)))
\end{array}
$$

Using this expression as the implementation for the $Flood$ function[1] the assignment $M[d] := Flood(X[d])$ can be replaced by $M[d] := RShiftT(Xor(Increment(x), x))$.

Next, a bit parallel expression for $X[d]$ is derived using formulas (2.8) and (2.9) as specification.

$$
\begin{array}{ll}
 & X[d] \\
= & \quad \{\ \text{expand}\ \} \\
 & (X[d,0], X[d,1], \ldots, X[d,k])
\end{array}
$$

[1]The $Increment$ function can be implemented as an addition operation on a machine word. In case the argument is the $true$ vector the overflow bit will be set, take care not to violate correctness.

$$
\begin{array}{ll}
= & \{ \text{ (2.8) and (2.9) on page 50 } \} \\
& (M[d-1,0] \vee B[c,d,0] \\
& , M[d,0] \wedge M[d+1,0] \wedge (M[d-1,1] \vee B[c,d,1]) \\
& \vdots \\
& , M[d,k-1] \wedge M[d+1,k-1] \wedge (M[d-1,k] \vee B[c,d,k]) \\
&) \\
= & \{ \quad \wedge \quad \} \\
& (True \wedge True \wedge (M[d-1,0] \vee B[c,d,0]) \\
& , M[d,0] \wedge M[d+1,0] \wedge (M[d-1,1] \vee B[c,d,1]) \\
& \vdots \\
& , M[d,k-1] \wedge M[d+1,k-1] \wedge (M[d-1,k] \vee B[c,d,k]) \\
&) \\
= & \{ \; And \; \} \\
& And((True, M[d,0], \ldots, M[d,k-1]), \\
& \qquad (True, M[d+1,0], \ldots, M[d+1,k-1]), \\
& \qquad (M[d-1,0] \vee B[c,d,0], M[d-1,1] \vee B[c,d,1], \ldots, M[d-1,k] \vee B[c,d,k]) \\
&) \\
= & \{ \; Or \; \} \\
& And((True, M[d,0], \ldots, M[d,k-1]), \\
& \qquad (True, M[d+1,0], \ldots, M[d+1,k-1]), \\
& \qquad Or(\; (M[d-1,0], \ldots, M[d-1,k]) \;,\; (B[c,d,0], \ldots, B[c,d,k]) \;) \\
&) \\
= & \{ \; RShiftT \; \} \\
& And(RShiftT(M[d]), RShiftT(M[d+1]), Or(M[d-1], B[c,d]))
\end{array}
$$

This derivation shows the proper assignment:
$X[d] := And(RShiftT(M[d]), RShiftT(M[d+1]), Or(M[d-1], B[c,d]))$

It is important to notice that both assignments: $M[d] := RShiftT(Xor(Increment(X[d]), X[d]))$ and $X[d] := And(RShiftT(M[d]), RShiftT(M[d+1]), Or(M[d-1], B[c,d]))$ do not depend on any values they modify. More precise, the assignment to $M[d]$ does not depend on any other value of M. Because of this both assignments can be computed for all d in parallel. The derivation below shows how to compute the bit parallel expression for the assignment to M:

$$
\begin{array}{ll}
& M \\
= & \{ \text{ expand } \} \\
& (M[1], \ldots, M[m-k]) \\
= & \{ \text{ spec. of assignment to } M[d] \; \} \\
& (RShiftT(Xor(Increment(X[1]), X[1])), \\
& \vdots \\
& , RShiftT(Xor(Increment(X[m-k]), X[m-k])))
\end{array}
$$

$$
\begin{aligned}
= \quad & \{\ RShiftTs(x) = (RShiftT(x[1]), \ldots, RShiftT(x[m-k])),\ \text{simplify}\ \} \\
& RShiftTs((Xor(Increment(X[1]), X[1]), \ldots, Xor(Increment(X[m-k]), X[m-k]))) \\
= \quad & \{\ Xors(x,y) = (Xor(x[1], y[1]), \ldots, Xor(x[m-k], y[m-k])),\ \text{simplify}\ \} \\
& RShiftTs(Xors((Increment(X[1]), \ldots, Increment(X[m-k])), X)) \\
= \quad & \{\ Increments(x) = (Increment(x[1]), \ldots, Increment(x[m-k])),\ \text{simplify}\ \} \\
& RShiftTs(Xors(Increments(X), X))
\end{aligned}
$$

Thus $M := RShiftTs(Xors(Increments(X), X))$.

Similarly, the assignment to X can be expressed as a function in M. Expanding the assignments $X[j] := And(RShiftT(M[j]), RShiftT(M[j+1]), Or(M[j-1], B[c,j]))$ gives the following:

$$
\begin{aligned}
& X \\
= \quad & \{\ \text{expand}\ \} \\
& (X[1], \ldots, X[m-k]) \\
= \quad & \{\ \text{spec. of assignment to } X[d]\ \} \\
& (And(RShiftT(M[1]), RShiftT(M[2]), Or(M[0], B[c,1])) \\
& \vdots \\
& , And(RShiftT(M[m-k]), RShiftT(M[m-k+1]), Or(M[m-k-1], B[c,m-k])) \\
&) \\
= \quad & \{\ Ands(x,y) = (And(x[1], y[1]), \ldots, And(x[m-k], y[m-k]))\ \} \\
& Ands((RShiftT(M[1]), \ldots, RShiftT(M[m-k])), \\
& \qquad (RShiftT(M[2]), \ldots, RShiftT(M[m-k+1])), \\
& \qquad (Or(M[0], B[c,1]), \ldots, Or(M[m-k-1], B[c,m-k])) \\
&) \\
= \quad & \{\ Ors(x,y) = (Or(x[1], y[1]), \ldots, Or(x[m-k], y[m-k]))\ \} \\
& Ands((RShiftT(M[1]), \ldots, RShiftT(M[m-k])), \\
& \qquad (RShiftT(M[2]), \ldots, RShiftT(M[m-k+1])), \\
& \qquad Ors(\ (M[0], \ldots, M[m-k-1])\ ,\ (B[c,1], \ldots, B[c,m-k])\ ) \\
&) \\
= \quad & \{\ RShiftTs,\ \text{simplify}\ \} \\
& Ands(RShiftTs(M), \\
& \qquad RShiftTs((M[2], \ldots, M[m-k+1])), \\
& \qquad Ors(\ (M[0], \ldots, M[m-k-1])\ ,\ (B[c,1], \ldots, B[c,m-k])\ ) \\
&) \\
= \quad & \{\ M[m-k+1] = true^{k+1}, M[0] = false^{k+1}\ \} \\
& Ands(RShiftTs(M), \\
& \qquad RShiftTs((M[2], \ldots, M[m-k], true^{k+1})), \\
& \qquad Ors(\ (false^{k+1}, M[1], \ldots, M[m-k-1])\ ,\ (B[c,1], \ldots, B[c,m-k])\ )
\end{aligned}
$$

$$
\begin{array}{ll}
&) \\
= & \quad \{\ ShiftUpT((M[1],\ldots,M[m-k])) = (M[2],\ldots,M[m-k],true^{k+1}) \\
& \qquad ShiftDnF((M[1],\ldots,M[m-k])) = (false^{k+1},M[1],\ldots,M[m-k-1]) \\
& \qquad \text{simplify} \\
& \quad \} \\
& \quad Ands(RShiftTs(M), RShiftTs(ShiftUpT(M)), Ors(ShiftDnF(M), B[c]))
\end{array}
$$

Algorithm 2.14.15(PE, SP, EP, BM, AA, ND, AR, DW)

$R : O = \langle \textbf{set}\ l,v,r\ :\ lvr = T\ \wedge\ dst(v,p) \le k\ :\ (lv,r)\rangle \backslash LM$
$P_0 : ur = T\ \wedge\ O = \langle \textbf{set}\ x,y,z\ :\ xyz = T\ \wedge\ xy \le_p u\ \wedge\ dst(y,p) \le k\ :\ (xy,z)\rangle \backslash LM$
$P_1 : [[M]] = \delta^*(q_0,u)$
$Z_1 : \langle \forall i,d\ : 1 \le i \le k\ \wedge\ 1 \le d \le m-k\ :$
$\qquad X[d,i] \equiv (i-1,i+d-1) \notin \delta^*(q_0,u)\ \wedge\ (i-1,i+d) \notin \delta^*(q_0,u)$
$\qquad\qquad \wedge\ ((i,i+d-1) \notin \delta^*(q_0,u)\ \vee\ p[i+d-1] \neq c)\rangle$
$Z_2 : \langle \forall d\ :\ 1 \le d \le m-k\ :\ X[d,0] \equiv (0,d-1) \notin \delta^*(q_0,u)\ \vee\ p[d-1] \neq c\rangle$

if $m \le k \to O := \{(\varepsilon,T)\}$
$[\!]\ k < m \to O := \varnothing$
fi;
$\{\ P_0(u,r := \varepsilon,T)\ \}$
for $i \in [0,k], j \in [0,m-k] \to$
$\quad M[j,i] := j \neq 0$
rof;
$\{\ P_{0..1}(u,r := \varepsilon,T)\ \}$
$u,r := \varepsilon,T;\ \{\ P_{0..1}\ \}$
do $r :: cw \to \{\ P_{0..1}\ \}$
$\quad X := Ands(RShiftTs(M), RShiftTs(ShiftUpT(M)), Ors(ShiftDnF(M), B[c]))$
$\quad \{\ Z_0\ \wedge\ Z_1\ \}$
$\quad M := RShiftTs(Xors(Increments(X), X));$
$\quad \{\ P_1(u,r := uc,w)\ \}$
$\quad$ **if** $\neg M[m-k,k] \to O := O \cup \{(uc,w)\}$
$\quad [\!]\ M[m-k,k] \to skip$
$\quad$ **fi**; $\{\ P_{0..1}(u,r := uc,w)\ \}$
$\quad u,r := uc,w$
od$\{\ R\ \}$

2.14.2 Comparison with the literature

Consider the assignments to X and M in the algorithm presented in the previous section:

$X := Ands(RShiftTs(M), RShiftTs(ShiftUpT(M)), Ors(ShiftDnF(M), B[c]));$
$M := RShiftTs(Xors(Increments(X), X))$

The following intuition can be associated with the above statements:

- Expression $RShiftTs(M)$ represents diagonal transitions.
- Expression $RShiftTs(ShiftUpT(M))$ represents vertical transitions.
- Expression $Ors(ShiftDnF(M), B[c])$ represents horizontal transitions.
- Expression $RShiftTs(Xors(Increment(X), X))$ represents the ε-closure over X.

This clearly shows that what happens in the algorithm: First X is set to all states which can be reached from the active states in M by performing a transition on character c. Then M is set to the ε-closure over X. This is obvious a single step in the simulation of the automaton.

The following version was found in the literature [NR02] (rewritten to the same representation for easy comparison):

$X := Ors(ShiftDnF(M), B[c]);$
$M := Ands(RShiftTs(M), RShiftTs(ShiftUpT(M)), RShiftTs(Xors(Increments(X), X)))$

This only computes the ε-closure over states which become active due to horizontal transitions. Correctness of this form has not verified, however the following observations suggest it might be valid:

- If a tail of a diagonal is active, then the diagonal transitions will decrease the length of the tail by one. The ε-closure over this tail, is the same tail portion of the diagonal. Therefore it is not necessary to compute the ε-closure over the tail.
- If a tail of a diagonal is active, then a series of vertical transitions will make a tail section of the column below active. Since the an entire tail segment is active, the ε-closure will not add any new active states to the vertical transition.

In other words, only the ε-closure over the horizontal transitions has to be computed. Because the ε-closure is computed over an entire diagonal with bit parallelism, this observations does not improve performance in any way.

The exact reason for this organization remains unclear. Since there is no performance impact and because the former statement order is conceptually simpler, the former order is the preferred choice.

2.15 Superimposed automata algorithm

This section derives a multiple keyword approximate pattern matching algorithm starting at the algorithm presented in Section 2.1.

This section describes a multi-keyword approximate pattern matching algorithm. This algorithm filters the text, to quickly discard all portions of the text in which no approximate matches can be found. It requires a different approximate pattern matching algorithm to find the occurrences in the remainder of the text (for example, a dynamic programming algorithm).

This algorithm works by superimposing the automata for each of the patterns. The resulting automaton can be used as a filter. This filter is formalised and proven to be a proper filter.

Let P be the set of patterns ($P = \{p_0, p_1, \ldots, p_r\}$) each of length m. Note that all the patterns in P must be of the same length. If this is not the case the longer patterns must be reduced in length by taking a prefix of length equal to the length of the shortest pattern. The larger the differences in length between the patterns in P, the worse the performance of the algorithm becomes. In case the pattern lengths strongly differ from each other, consider using separate searching for all patterns of (almost) equal length.

Given set P as above, then the superimposed pattern SP_P is given by
$\langle \forall j \ : \ 0 \leq j < m \ : \ SP_P[j] = \langle \textbf{set } i \ : \ 0 \leq i \leq r \ : \ p_i[j] \rangle \rangle$.

Define a domain D_P as the set of all words over pattern SP_P: $D = \langle \prod j \ : \ 0 \leq j < m \ : \ SP_P[j] \rangle$

First the property $p \in P \Rightarrow p \in D_P$ is required ($p_k \in P \equiv 0 \leq k \leq r$):

$$
\begin{array}{ll}
 & p_k \in D_P \\
\equiv & \quad \{ \ D_P \ \} \\
 & \langle \forall j \ : \ 0 \leq j < m \ : \ p_k[j] \in SP_P[j] \rangle \\
\equiv & \quad \{ \ SP_P \ \} \\
 & \langle \forall j \ : \ 0 \leq j < m \ : \ p_k[j] \in \langle \textbf{set } i \ : \ 0 \leq i \leq r \ : \ p_i[j] \rangle \rangle \\
\Leftarrow & \quad \{ \text{ predicate calculus } \} \\
 & 0 \leq k \leq r \\
\equiv & \quad \{ \text{ def } P \ \} \\
 & p_k \in P
\end{array}
$$

The contraposition gives: $p \notin D_P \Rightarrow p \notin P$. From this can be concluded that by replacing set P by D_P a valid filter is constructed.

This filter can be used to modify the algorithm previously derived. Consider the set which is added to O.

$$
\begin{array}{ll}
 & \langle \textbf{set } x, y, p \ : \ p \in P \ \wedge \ xy = uc \ \wedge \ dst(y,p) \leq k \ : \ (x, y, w, p, dst(y,p)) \rangle = \varnothing \\
\Leftarrow & \quad \{ \ p \in P \Rightarrow p \in D_P \ \} \\
 & \langle \textbf{set } x, y, p \ : \ p \in D_P \ \wedge \ xy = uc \ \wedge \ dst(y,p) \leq k \ : \ (x, y, w, p, dst(y,p)) \rangle = \varnothing \\
\equiv & \quad \{ \text{ automaton: } \delta_P^*(q_0, u) \cap F \neq \varnothing \equiv \langle \exists x, y, p \ : \ xy = u \ \wedge \ p \in D_P \ : \ dst(y,p) \leq k \rangle \ \} \\
 & \delta_P^*(q_0, uc) \cap F = \varnothing
\end{array}
$$

$O := \langle \textbf{set } p \ : \ p \in P \ \wedge \ |p| \leq k \ : \ (\varepsilon, \varepsilon, T, p, |p|) \rangle;$
$\{ \ P_0(u, r := \varepsilon, T) \ \}$
$u, r := \varepsilon, T;$
$\{ \ P_0 \ \}$
do $r :: cw \rightarrow \{ \ P_0 \ \}$

if $\delta_P^*(q_0, uc) \cap F = \varnothing \rightarrow$ skip
$[\!]$ $\delta_P^*(q_0, uc) \cap F \neq \varnothing \rightarrow$
$O := O \cup \langle \textbf{set}\ x, y, p\ :\ p \in P\ \wedge\ xy = uc\ \wedge\ dst(y, p) \leq k\ :\ (x, y, w, p, dst(y, p)) \rangle$
fi;
$\{\ P_0(u, r := uc, w)\ \}$
$u, r := uc, w$
od$\{\ P_0\ \wedge\ \neg(r :: cw)\ \}$
$\{\ R\ \}$

Some notes about the above algorithm:

- for efficiency $\delta_P^*(q_0, uc)$ should not be computed over and over again, rather maintain invariance that $\delta_P^*(q_0, u)$ equals some variable.
- a good choice is simulating $\delta_P^*(q_0, uc)$ with bit parallelism. For example, the system invariant I_0 of the row-wise bit parallel algorithm (see page 39) should be changed to:

$$\begin{aligned} &\langle \forall c, j\ :\ c \in \Sigma\ \wedge\ 1 \leq j \leq m\ :\ B[c, j] = c \in SP_P[j-1] \rangle \\ &\wedge\ \langle \forall c\ :\ c \in \Sigma\ :\ B[c, 0] = True \rangle \end{aligned}$$

- in case $\delta_P^*(q_0, uc) \cap F$ is not empty, only suffixes of uc of length at most $m + k$ must be verified for matches, since no substring of length at least $m + k + 1$ can match to a pattern with at most k differences.

Regarding the update:

$$O := O \cup \langle \textbf{set}\ x, y, p\ :\ p \in P\ \wedge\ xy = uc\ \wedge\ dst(y, p) \leq k\ :\ (x, y, w, p, dst(y, p)) \rangle$$

This line is required to verify if the match which was found at position uc is an actual match with a pattern in P.

To compute this result, introduce function *Verify* as follows:

$$Verify(l, v, r, P) = \langle \textbf{set}\ x, y, p\ :\ p \in P\ \wedge\ xy = v\ \wedge\ dst(y, p) \leq k\ :\ (lx, y, r, p, dst(y, p)) \rangle$$

The most simple form of verification is to iteratively inspect each of the patterns in P. This lineair form of verification is derived next.

$$\begin{aligned} & Verify(l, v, r, P) \\ = & \quad \{\ \text{specification}\ \} \\ & \langle \textbf{set}\ x, y, p\ :\ p \in P\ \wedge\ xy = v\ \wedge\ dst(y, p) \leq k\ :\ (lx, y, r, p, dst(y, p)) \rangle \\ = & \quad \{\ \text{nesting}\ \} \\ & \langle \bigcup p\ :\ p \in P\ :\ \langle \textbf{set}\ x, y\ :\ xy = v\ \wedge\ dst(y, p) \leq k\ :\ (lx, y, r, p, dst(y, p)) \rangle \rangle \end{aligned}$$

This directly leads to the implementation:

$Verify(l, v, r, P)$:
$[\![$
  $result := \varnothing$
  for $p \in P \rightarrow$
    $result := result \cup \langle$**set** $x, y \ : \ xy = v \ \wedge \ dst(y,p) \leq k \ : \ (lx, y, r, p, dst(y,p))\rangle$
  rof;
  return $result$
$]\!]$

There is a slight optimization by changing the initial call to verify by using the knowledge that $dst(y,p) \leq k$ if and only if $m - k \leq |y| \leq m + k$.

$\langle$**set** $x, y, p \ : \ p \in P \ \wedge \ xy = uc \ \wedge \ dst(y,p) \leq k \ : \ (x, y, w, p, dst(y,p))\rangle$
$\equiv$ { $\lfloor$ and $\lceil$ }
$\langle$**set** $x, y, p \ : \ p \in P \ \wedge \ xy = (uc \lfloor (m+k))(uc \lceil (m+k))$
$\wedge \ dst(y,p) \leq k \ : \ (x, y, w, p, dst(y,p))\rangle$
$\equiv$ { dummy transform $x := (uc \lfloor (m+k))x$,
use $dst(y,p) \leq k$ if and only if $m - k \leq |y| \leq m + k$
}
$\langle$**set** $x, y, p \ : \ p \in P \ \wedge \ xy = (uc \lceil (m+k))$
$\wedge \ dst(y,p) \leq k \ : \ ((uc \lfloor (m+k))x, y, w, p, dst(y,p))\rangle$
$\equiv$ { specification *Verify* }
$Verify(uc \lfloor (m+k), uc \lceil (m+k), w, P)$

This derivation shows that the assignment to O can be implemented as $O := O \cup Verify(uc \lfloor m + k, uc \lceil m + k, w, P)$.

The lineair verification function used here verifies against each pattern in P. The following sections derive several different hierarchical verification functions.

2.15.1 Forward verification

Whenever the superimposed automaton algorithm finds an occurrence this occurrence must be verified because it may be a false positive. In the lineair verification approach in the previous section a test is performed against each of the patterns in set P. Suppose set P is split in sets P_a and P_b. If superimposing all patterns in P_a does not generate a match, then all pattern in P_a can be ignored. This suggest that a hierarchical verification function might yield a significant improvement. This section derives a hierarchical verification method.

First, consider the basic case where set P is a singleton set.

$Verify(l, v, r, \{p_0\})$
$\equiv$ { spec *verify* }
$\langle$**set** $x, y, p \ : \ xy = v \ \wedge \ p \in \{p_0\} \ \wedge \ dst(y,p) \leq k \ : \ (lx, y, r, p, dst(y,p))\rangle$

$\equiv$ { one point rule }

$\langle \textbf{set}\ x, y\ :\ xy = v\ \wedge\ dst(y, p_0) \leq k\ :\ (lx, y, r, p_0, dst(y, p_0))\rangle$

This quantification is the same in all verification methods and is not described in further detail here.

Next, consider the case $P = \{p_0, .., p_r\}$ with $(1 \leq r)$.

$Verify(l, v, r, P)$

$\equiv$ { spec $verify$ }

$\langle \textbf{set}\ x, y, p\ :\ xy = v\ \wedge\ p \in P\ \wedge\ dst(y, p) \leq k\ :\ (lx, y, r, p, dst(y, p))\rangle$

$\equiv$ { domain split: $\bullet Split : Set \mapsto \mathcal{P}(Set)$(see below) }

$\langle \bigcup s\ :\ s \in Split(P)\ :$
$\quad \langle \textbf{set}\ x, y, p\ :\ xy = v\ \wedge\ p \in s\ \wedge\ dst(y, p) \leq k\ :\ (lx, y, r, p, dst(y, p))\rangle\rangle$

$\equiv$ { $Verify,\ s \subset P$ }

$\langle \bigcup s\ :\ s \in Split(P)\ :\ Verify(l, v, r, s)\rangle$

The function $Split$ mentioned above should satisfy the following properties (for $2 \leq |S|$):

- $\langle \forall s\ :\ s \in Split(S)\ :\ \varnothing \subset s \subset S\rangle$
- $\langle \bigcup s\ :\ s \in Split(S)\ :\ s\rangle = S$

Note that the splits do not need to be disjoint.

The derivation as presented above leads to a recursive variant of the lineair verification method. But by applying the filter the following recursive verification function is found:

```
verify(l, v, r, P)
|[
  if P :: {p} → result := ⟨set x, y : xy = v ∧ dst(y, p) ≤ k : (lx, y, r, p, dst(y, p))⟩
  [] ¬(P :: {p}) →
     result := ∅
     if δ*_P(q_0, v) ∩ F = ∅ → skip
     [] δ*_P(q_0, v) ∩ F ≠ ∅ →
        for s ∈ Split(S) →
            result := result ∪ Verify(l, v, r, s)
        rof
     fi
  fi
  return result
]|
```

Note that the function δ^*_P depends on the parameter P. More precise, δ^*_P is the transition function of the automaton which is constructed by superimposing the automata of all patterns in P.

2.15.2 Backwards verification

The main search algorithm finds the end positions of occurrences which match with a pattern in D_P. The task of the verification function is to determine whether or not there is also a match with a pattern in P. Automata used in forward scans typically find the end position of an occurrence. The verification function checks whether there is an approximate occurrence of a pattern in P which ends at the same position as the filter reported. The algorithm in this section verifies this by searching a starting position rather then an end position.

The backwards verification function uses a modified automaton:

- There is no Σ loop on the starting state. Because a possible occurrence has already been found, there is no need to search for it. The automaton without the Σ loop acts as an acceptor not as an automaton for searching.
- The automaton is built on the reverse patterns because the text is scanned backwards.

This automaton is characterized by the following equation:

$$\delta^*_P(q_0, u) \cap F \neq \varnothing \equiv \langle \exists p \ : \ p \in D_P \ : \ dst(u, p^R) \leq k \rangle$$

Just like in the case of forward verification, the automaton is used to short-circuit the verification in case $\neg(P :: \{p\})$.

$$\begin{array}{ll} & \langle \textbf{set}\ x, y, p \ : \ p \in P \ \wedge \ xy = v \ \wedge \ dst(y, p) \leq k \ : \ (lx, y, r, p, dst(y, p)) \rangle = \varnothing \\ \Leftarrow & \quad \{\ p \in P \Rightarrow p \in D_P\ \} \\ & \langle \textbf{set}\ x, y, p \ : \ p \in D_P \ \wedge \ xy = v \ \wedge \ dst(y, p) \leq k \ : \ (lx, y, r, p, dst(y, p)) \rangle = \varnothing \\ \equiv & \quad \{\ \bullet R_0 : o = \langle \downarrow i \ : \ 0 \leq i \leq |v| \ \wedge \ \langle \exists p \ : \ p \in D_P \ : \ dst(v[i..|v|), p) \leq k \rangle \ : \ i \rangle\ \} \\ & o = \infty \end{array}$$

If there does not exist a suffix of v which matches with a pattern in D_P then the verification can be stopped. The backwards scan determines such a suffix of v with maximum length (in order not to miss any occurrences).

Postcondition R_0 is established by the validity of guard $t = 0$ and invariant P_0:

$$o = \langle \downarrow i \ : \ t \leq i \leq |v| \ \wedge \ \langle \exists p \ : \ p \in D_P \ : \ dst(v[i..|v|), p) \leq k \rangle \ : \ i \rangle \qquad (P_0)$$

Initialization of invariant P_0:

$$\begin{array}{ll} & P0(t := |v|) \\ \equiv & \quad \{\ \text{subst}\ \} \\ & o = \langle \downarrow i \ : \ |v| \leq i \leq |v| \ \wedge \ \langle \exists p \ : \ p \in D_P \ : \ dst(v[i..|v|), p) \leq k \rangle \ : \ i \rangle \\ \equiv & \quad \{\ \text{one point domain}\ \} \\ & o = \left\{ \begin{array}{l} |v| \text{ if } \langle \exists p \ : \ p \in D_P \ : \ dst(\varepsilon, p) \leq k \rangle \\ \infty \text{ if } \neg \langle \exists p \ : \ p \in D_P \ : \ dst(\varepsilon, p) \leq k \rangle \end{array} \right. \end{array}$$

$$
\begin{array}{ll}
\equiv & \{ \text{ def } dst \text{ } \} \\
& o = \left\{ \begin{array}{l} |v| \text{ if } \langle \exists p \ : \ p \in D_P \ : \ |p| \leq k \rangle \\ \infty \text{ if } \neg \langle \exists p \ : \ p \in D_P \ : \ |p| \leq k \rangle \end{array} \right.
\end{array}
$$

The scan proceeds backwards through the text, therefore decrease t by one:

$$
\begin{array}{ll}
& \langle \downarrow i \ : \ t \leq i \leq |v| \ \wedge \ \langle \exists p \ : \ p \in D_P \ : \ dst(v[i..|v|), p) \leq k \rangle \ : \ i \rangle (t := t-1) \\
= & \{ \text{ subst } \} \\
& \langle \downarrow i \ : \ t-1 \leq i \leq |v| \ \wedge \ \langle \exists p \ : \ p \in D_P \ : \ dst(v[i..|v|), p) \leq k \rangle \ : \ i \rangle \\
= & \{ \ P_0, \text{ split off } i = t-1 \ \} \\
& \left\{ \begin{array}{ll} o \downarrow t-1 & \text{if } \langle \exists p \ : \ p \in D_P \ : \ dst(v[t-1..|v|), p) \leq k \rangle \\ o & \text{if } \neg \langle \exists p \ : \ p \in D_P \ : \ dst(v[t-1..|v|), p) \leq k \rangle \end{array} \right. \\
= & \{ \ dst(x,y) = dst(x^R, y^R) \ \} \\
& \left\{ \begin{array}{ll} o \downarrow t-1 & \text{if } \langle \exists p \ : \ p \in D_P \ : \ dst(v[t-1..|v|)^R, p^R) \leq k \rangle \\ o & \text{if } \neg \langle \exists p \ : \ p \in D_P \ : \ dst(v[t-1..|v|)^R, p^R) \leq k \rangle \end{array} \right. \\
= & \{ \ \bullet P1(t := t-1) \text{ where } P1 : [[S]] = \delta_P^*(q_0, ()v[t..|v|)^R) \ \} \\
& \left\{ \begin{array}{ll} o \downarrow t-1 & \text{if } [[S]] \cap F \neq \varnothing \\ o & \text{if } [[S]] \cap F = \varnothing \end{array} \right.
\end{array}
$$

Invariant P_1 is initialized by statement $[[S]] := \delta_P^*(q_0, \varepsilon)$ and updated by statement $[[S]] := \delta_P^*([[S]], v[t-1])$.

Combining all these results gives the following verification function:

```
Verify(l, v, r, P) :
[[
  if P :: {p} →
     result := ⟨set x, y : xy = v ∧ dst(y, p) ≤ k : (lx, y, r, p, dst(y, p))⟩
  [] ¬(P :: {p}) →
     if ⟨∃p : p ∈ D_P : |p| ≤ k⟩ → o := |v|
     [] ¬⟨∃p : p ∈ D_P : |p| ≤ k⟩ → o := ∞
     fi; { P_0(t := |v|) }
     [[S]] := δ*_P(q_0, ε);
     { P_{0..1}(t := |v|) }
     t := |v|;
     do t ≠ 0 → { P_{0..1} }
        [[S]] := δ*_P([[S]], v[t − 1]);
        { P_0 ∧ P_1(t := t − 1) }
        if [[S]] ∩ F ≠ ∅ → o := o ↓ t − 1
        [] [[S]] ∩ F = ∅ → skip
        fi; { P_{0..1}(t := t − 1) }
        t := t − 1
     od; { P_0 ∧ p_1 ∧ t = 0 }
     { R_0 }
```

```
    result := ∅
    if o = ∞ → skip
    [] o ≠ ∞ → for s ∈ Split(P) → result := result ∪ Verify(l(v[0..o)), v[o..|v|), r, s) rof;
    fi;
  fi;
  return result
]|
```

Some remarks about this algorithm:

- The simulation loop can be short-circuited (once $[[S]] = \varnothing$ it will remain so).
- The quantification $\langle \textbf{set } x, y \ : \ xy = v \ \wedge \ dst(y,p) \leq k \ : \ (lx, y, r, p, dst(y,p)) \rangle$ can easily be computed using this transition function δ_P^* by processing v in reverse and reporting a match each time a final state becomes active.
- Note that the parameters of the recursive call are adjusted. This ensures a tight lower bound on the window which is used for verification (the forward verification function uses a window of length $m+k$, this window lies somewhere between $m+k$ and $m-k$).

A slight optimization is possible when using either the forward or the backwards hierarchical verification. The call to *Verify* in the main search algorithm can be replaced by a call to *Verify* on the splitted sets. I.e. instead of statement: $O := O \cup Verify(uc \lfloor m+k, uc \rceil m+k, w, P)$, one can write:

for $s \in Split(S) \rightarrow$
  $O := O \cup Verify(uc \lfloor m+k, uc \rceil m+k, w, s)$
rof

This is because the first call of verify checks against the entire set P. This was already verified by the search algorithm itself.

2.15.3 The Split Function

The hierarchical verification functions use a function *Split* to split the pattern set into smaller sets. This function must satisfy the following conditions:

- $\langle \forall s \ : \ s \in Split(S) \ : \ s \subset S \rangle$ (no new elements and the individual sets must be smaller (for termination of the recursion))
- $\langle \bigcup s \ : \ s \in Split(S) \ : \ s \rangle = S$ (no elements are left out)

There are several ways to implement this split function.

- The most simple split function is specified by:

  $$Split(\{p_0, ..p_r\}) = \langle \textbf{set } i \ : \ 0 \leq i \leq r \ : \ \{p_i\} \rangle$$

  This creates a lineair verification against all patterns, though it is more efficient to use a real lineair verification function.

- A more efficient and easy to implement function is to split the set into two sets: (typically $i = r/2$)
$$Split(\{p_0, ..p_i, ..p_r\}) = \{\{p_0, ..p_i\}, \{p_{i+1}..p_r\}\}$$
- The following example suggests that more efficient Split functions exist, although they might be expensive to compute.

  Consider a set $P = \{aaa, bbb, abb, baa\}$ and alphabet $\Sigma = a, b$ This set can be split as follows (restricting the split to two sets):

  - $\{\{aaa, bbb\}, \{abb, baa\}\}$ This gives transitions: $q_0 \xrightarrow{a,b} q_1 \xrightarrow{a,b} q_2 \xrightarrow{a,b} q_3$ This results in a lot of false positives and is the worst possible split which can be made, since it will give no new information.
  - $\{\{aaa, abb\}, \{bbb, baa\}\}$ This gives transitions: $q_0 \xrightarrow{a} q_1 \xrightarrow{a,b} q_2 \xrightarrow{a,b} q_3$ for the one half and $q_0 \xrightarrow{b} q_1 \xrightarrow{a,b} q_2 \xrightarrow{a,b} q_3$ for the other. This split is significantly better than the previous split, however there are still some false positives.
  - $\{\{aaa, baa\}, \{bbb, abb\}\}$ This gives transitions: $q_0 \xrightarrow{a,b} q_1 \xrightarrow{a} q_2 \xrightarrow{a} q_3$ for the one half and $q_0 \xrightarrow{a,b} q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_3$ for the other. This split is optimal, since it does not generate any false positives at all.

The computation of an optimal *Split* function can be very expensive. An approach based on heuristics might yield reasonable results. I suspect that such an algorithm might look like the following: (for some boolean function f)

$Split(P)$:
$[\![$
  $result := \varnothing$;
  for $p \in P \rightarrow$
    if $s : s \in result \ \wedge \ f(p, s) \rightarrow s := s \cup \{p\}$
    $[\!]$ $\neg\langle \exists s \ : \ s \in result \ : \ f(p, s)\rangle \rightarrow result := result \cup \{\{p\}\}$
    fi
  rof
  return $result$
$]\!]$

The function $f(p, s)$ must be some indication if p fits properly in set s. For example, the number of false positives which are accepted by s after adding p must be below some threshold. Another option is to add p to the set s which fits best. This problem requires further research.

2.16 ABNDM derivation

In this section a derivation of the ABNDM algorithm is presented. The ABNDM algorithm is the extension of the BNDM algorithm to approximate pattern matching. The algorithm processes the text by shifting a window from left to right and reading the characters in the window from right to left.

There are two important observations for understanding the ABNDM algorithm:

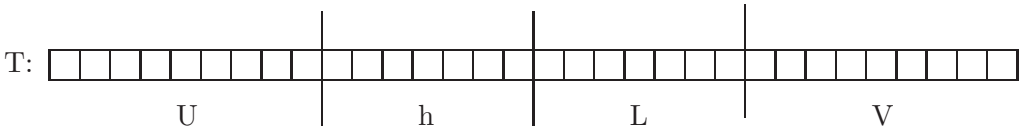
- if a string matches with the pattern with at most k differences, then any prefix of the string matches with some prefix of the pattern with at most k differences. Formally:

$$dst(v,p) \leq k \Rightarrow \langle \forall s \ : \ s \leq_p v \ : \ \langle \exists t \ : \ t \leq_p p \ : \ dst(s,t) \leq k \rangle \rangle \tag{2.10}$$

- any approximate occurrence of the pattern has length at least $m-k$

The text can be filtered by considering all substrings w of the text such that a prefix of length $m-k$ of w matches with a prefix of p with at most k differencess. If such a substring is found, a check for an occurrence must be performed. Why check prefixes of length $m-k$? The longer the prefixes the less occurrence checks are required (the number of false positives decreases) and the maximum prefix length which each occurrence is guaranteed to have is equal to $m-k$.

The ABNDM algorithm solves the approximate pattern matching problem by shifting a window of length $m-k$ over the text from left to right. It checks for an occurrence if the substring in the window is a match with a prefix of p with no more than k differences. The following figure shows how the text is divided into four partitions (the window W is equal to hL).



The invariant of the main loop is that all matches starting before the window are stored in a set O:

$$\begin{aligned} O = \langle \textbf{set } l,v,r \ : \ lvr = T \ \wedge \ l <_p U \ \wedge \ dst(v,p) \leq k \ : \ (l,v,r,dst(v,p)) \rangle \\ \wedge \ UWV = T \ \wedge \ (|W| = m-k \ \vee \ |WV| < m-k) \end{aligned} \tag{P_0}$$

Since there are no occurrences of length smaller than $m-k$, invariant P_0 and $|WV| < m-k$ imply the postcondition R. The guard of the repetition therefore becomes $|W| = m-k$.

Since the windows should move over the text from left to right, the window should be initialized to the first $m-k$ characters of the text. Invariant P_0 is therefore initialized by statement: $U,W,V := \varepsilon, T \uparrow (m-k), T \downarrow (m-k)$ provided that $O := \varnothing$.

While it is possible to shift the window to the right one character at a time, it is not the most efficient to do so. The window should be shifted to the start of the first suffix of the window which is a possible prefix of an occurrence. The following invariant identifies this possible occurrence prefix as L:

$$\langle \forall f \ : \ \varepsilon <_s f <_s h \ : \ \neg \langle \exists s \ : \ s \leq_p p \ : \ dst(fL,s) \leq k \rangle \rangle \ \wedge \ hL = W \ \wedge \ h \neq \varepsilon \tag{R_1}$$

The conjunct $h \neq \varepsilon$ (or equivalently $L \neq W$) is required to guarantee progress.

Now there is sufficient information to derive the main repetition. The derivation below shows how set O should change if the window is shifted to the beginning of L. Set O should either remain the same or change if a match was found at the start of the window.

For brevity, the different conjuncts of P_0 are proven separately.

$(UWV = T \ \wedge \ (|W| = m - k \ \vee \ |WV| < m - k))(U, W, V := Uh, L(V \uparrow |h|), V \downarrow |h|)$

$\equiv$ { subst }

$UhL(V \uparrow |h|)(V \downarrow |h|) = T \ \wedge \ (|L(V \uparrow |h|)| = m - k \ \vee \ |L(V \uparrow |h|)(V \downarrow |h|)| < m - k)$

$\equiv$ { property of $\uparrow$ and $\downarrow$ }

$UhLV = T \ \wedge \ (|L(V \uparrow |h|)| = m - k \ \vee \ |LV| < m - k)$

$\equiv$ { $R_1 : hL = W, guard : |W| = m - k, P_0$ }

$true$

$\langle \textbf{set} \ l, v, r \ : \ lvr = T \ \wedge \ l <_p U \ \wedge \ dst(v, p) \leq k \ : \ (l, v, r, dst(v, p)) \rangle$
$(U, W, V := Uh, L(V \uparrow |h|), V \downarrow |h|)$

$=$ { subst }

$\langle \textbf{set} \ l, v, r \ : \ lvr = T \ \wedge \ l <_p Uh \ \wedge \ dst(v, p) \leq k \ : \ (l, v, r, dst(v, p)) \rangle$

$=$ { domain split ($l <_p U, l = U, U <_p l <_p Uh$) use $h \neq \varepsilon, P_0$ }

$O \cup$
$\langle \textbf{set} \ v, r \ : \ Uvr = T \ \wedge \ dst(v, p) \leq k \ : \ (U, v, r, dst(v, p)) \rangle \cup$
$\langle \textbf{set} \ l, v, r \ : \ lvr = T \ \wedge \ U <_p l <_p Uh \ \wedge \ dst(v, p) \leq k \ : \ (l, v, r, dst(v, p)) \rangle$

$=$ { $l := Ul, T = UWV \ (P_0)$ }

$O \cup$
$\langle \textbf{set} \ v, r \ : \ vr = WV \ \wedge \ dst(v, p) \leq k \ : \ (U, v, r, dst(v, p)) \rangle \cup$
$\langle \textbf{set} \ l, v, r \ : \ lvr = WV \ \wedge \ \varepsilon <_p l <_p h \ \wedge \ dst(v, p) \leq k \ : \ (Ul, v, r, dst(v, p)) \rangle$

$=$ { $dst(v, p) \leq k \Rightarrow m - k \leq |v|, |W| = m - k, W = hL$
1^{st} formula: $v := Wv$(allowed because $vr = WV, |W| \leq |v|$)
2^{nd} formula: intro u such that $lu = h, v := uLv$
$(luLvr = WV = hLV \Leftarrow lu = h \ \wedge \ vr = V)$
}

$O \cup$
$\langle \textbf{set} \ v, r \ : \ vr = V \ \wedge \ dst(Wv, p) \leq k \ : \ (U, Wv, r, dst(Wv, p)) \rangle \cup$
$\langle \textbf{set} \ l, u, v, r \ : \ lu = h \ \wedge \ vr = V \ \wedge \ \varepsilon <_p l <_p h \ \wedge \ dst(uLv, p) \leq k \ :$
$(Ul, uLv, r, dst(uLv, p)) \rangle$

$=$ { $W \leq_p Wv, uL \leq_p uLv$, lemma 2.10 on page 64 }

$O \cup$
$\langle \textbf{set} \ v, r \ : \ vr = V \ \wedge \ dst(Wv, p) \leq k \ \wedge$
$\langle \exists s \ : \ s \leq_p p \ : \ dst(W, s) \leq k \rangle \ : \ (U, Wv, r, dst(v, p)) \rangle \cup$
$\langle \textbf{set} \ l, u, v, r \ : \ lu = h \ \wedge \ vr = V \ \wedge \ \varepsilon <_p l <_p h \ \wedge \ dst(uLv, p) \leq k \ \wedge$
$\langle \exists s \ : \ s \leq_p p \ : \ dst(uL, s) \leq k \rangle \ : \ (Ul, uLv, r, dst(uv, p)) \rangle$

$=$ { $R_1, \varepsilon <_s u <_s h$ because $\varepsilon <_p l <_p h$ and $lu = h$ }

$O \cup$
$\langle \textbf{set} \ v, r \ : \ vr = V \ \wedge \ dst(Wv, p) \leq k \ \wedge$
$\langle \exists s \ : \ s \leq_p p \ : \ dst(W, s) \leq k \rangle \ : \ (U, Wv, r, dst(v, p)) \rangle$

$=$ { automaton: $\delta^*(q_0, u^R) \cap F \neq \varnothing \equiv \langle \exists s \ : \ s \leq_p p \ : \ dst(u, s) \leq k \rangle$,

$$
\begin{array}{ll}
 & \quad R_2 : [[S]] = \delta^*(q_0, (W)^R) \\
 & \} \\
 & O \cup \langle \textbf{set } v, r \ : \ vr = V \ \wedge \ dst(Wv, p) \leq k \ \wedge \ [[S]] \cap F \neq \varnothing \ : \\
 & \quad (U, Wv, r, dst(Wv, p)) \rangle \\
= & \quad \{ \text{ case analysis } \} \\
 & O \cup \begin{cases} \langle \textbf{set } v, r \ : \ vr = V \ \wedge \ dst(Wv, p) \leq k \ : \ (U, Wv, r, dst(Wv, p)) \rangle & \text{if } [[S]] \cap F \neq \varnothing \\ \varnothing & \text{if } [[S]] \cap F = \varnothing \end{cases}
\end{array}
$$

The automaton in the proof above is described in Section 3.4 on page 84. The above derivation introduces a new postcondition R_2:

$$[[S]] = \delta^*(q_0, (W)^R) \qquad (R_2)$$

Here δ^* is the transition function of an automaton such that:

$$\delta^*(q_0, u^R) \cap F \neq \varnothing \equiv \langle \exists s \ : \ s \leq_p p \ : \ dst(u, s) \leq k \rangle$$

This leads to the following program template:

$O := \varnothing$;
$\{ \ P_0(U, W, V := \varepsilon, T \uparrow (m-k), T \downarrow (m-k)) \ \}$
$U, W, V := \varepsilon, T \uparrow (m-k), T \downarrow (m-k)$;
$\{ \ P_0 \ \}$
do $|W| = m - k \rightarrow \{ \ P_0 \ \}$
  $\{ \ (R_1 \ \wedge \ R_2)? \ \}$
  if $[[S]] \cap F \neq \varnothing \rightarrow$
    $O := O \cup \langle \textbf{set } v, r \ : \ vr = V \ \wedge \ dst(W \cdot v, p) \leq k \ : \ (U, Wv, r, dst(Wv, p)) \rangle$
  $[\!]$ $[[S]] \cap F = \varnothing \rightarrow$ skip
  fi; $\{ \ P_0(U, W, V := Uh, L(V \uparrow |h|), V \downarrow |h|) \ \}$
  $U, W, V := Uh, L(V \uparrow |h|), V \downarrow |h|$;
od; $\{ \ P_0 \ \wedge \ |W| \neq m - k \ \}$
$\{ \ R \ \}$

2.16.1 The inner repetition

The inner repetition of the ABNDM algorithm must establish postconditions R_1 and R_2. In the inner repetition the window is split into three different parts:

- a prefix t which has not yet been scanned
- a factor x in which no occurrences start
- a suffix L which may contain prefixes of occurrences.

The postconditions will be established by a backward scan through the window and three invariants, one for postcondition R_1, one for R_2, and a third for the boundary conditions.

$$\langle \forall f \ : \ \varepsilon <_s f <_s x \ : \ \neg\langle \exists s \ : \ s \leq_p p \ : \ dst(fL, s) \leq k \rangle \rangle \tag{P_1}$$

$$[[S]] = \delta^*(q_0, (xL)^R) \tag{P_2}$$

$$tx = h \ \wedge \ hL = W \ \wedge \ x \neq \varepsilon \tag{P_3}$$

Note that postcondition R_1 is established by $P_1 \ \wedge \ P_3 \ \wedge \ t = \varepsilon$ and that postcondition R_2 is established by $P_2 \ \wedge \ P_3 \ \wedge \ t = \varepsilon$. The guard of the inner repetition is $t \neq \varepsilon$ or equivalently: $t :: wa$.

Invariants P_1 and P_3 are established by statement $t, x, h, L := W \lfloor 1, W \rceil 1, W, \varepsilon)$ (P_1 due to an empty domain. P_3 due to the properties of $\lfloor$ and $\rceil$ and the fact that $0 < |W|$ which follows from the outer guard and the fact that $k < m$).

The backward scan processes the window character by character, therefore inspect a modification of t and x. Note that P_3 is maintained by $t, x := w, ax$.

$$\begin{array}{ll}
 & P_1(t, x := w, ax) \\
\equiv & \quad \{ \text{ subst } \} \\
 & \langle \forall f \ : \ \varepsilon <_s f <_s ax \ : \ \neg\langle \exists s \ : \ s \leq_p p \ : \ dst(fL, s) \leq k \rangle \rangle \\
\equiv & \quad \{ \ \mathit{suff}, P_3 : x \neq \varepsilon \ \} \\
 & \langle \forall f \ : \ \varepsilon <_s f <_s x \ \vee \ f = x \ : \ \neg\langle \exists s \ : \ s \leq_p p \ : \ dst(fL, s) \leq k \rangle \rangle \\
\equiv & \quad \{ \text{ domain split, } P_1 \ \} \\
 & \neg\langle \exists s \ : \ s \leq_p p \ : \ dst(xL, s) \leq k \rangle \\
\equiv & \quad \{ \text{ automaton: } \delta^*(q_0, u^R) \cap F \neq \varnothing \equiv \langle \exists s \ : \ s \leq_p p \ : \ dst(u, s) \leq k \rangle \ \} \\
 & \delta^*(q_0, (xL)^R) \cap F = \varnothing \\
\equiv & \quad \{ \ P_2 \ \} \\
 & [[S]] \cap F = \varnothing
\end{array}$$

If $[[S]] \cap F \neq \varnothing$ then variable L must be changed. Assignment $t, x, h, L := w, a, wa, xL$ maintains both P_1 and P_3 (This creates an empty domain in P_1. The proof of invariance of P_3 requires validity of P_3 and the guard).

Initialization of P_2 is not very difficult:

$$\begin{array}{ll}
 & P_2(t, x, h, L := W \lfloor 1, W \rceil 1, W, \varepsilon) \\
\equiv & \quad \{ \text{ subst } \} \\
 & [[S]] = \delta^*(q_0, (W \rceil 1)^R) \\
\equiv & \quad \{ \text{ property of reverse } \} \\
 & [[S]] = \delta^*(q_0, W \rceil 1)
\end{array}$$

The assignments $t, x := w, ax$ and $t, x, h, L := w, a, wa, xL$ should both maintain invariant P_2. Note that $P_2(t, x := w, ax) \equiv [[S]] = \delta^*(q_0, (axL)^R)$ and $P_2(t, x, h, L := w, a, wa, xL) \equiv [[S]] = \delta^*(q_0, (axL)^R)$ as well. Therefore both assignments can be derived simultaneously (this is just simulating an automaton):

$$
\begin{array}{ll}
 & \delta^*(q_0, (axL)^R) \\
= & \quad \{ \text{ property reverse } \} \\
 & \delta^*(q_0, (xL)^R \cdot a) \\
= & \quad \{ \ \delta^* \ \} \\
 & \delta^*(\delta^*(q_0, (xL)^R), a) \\
= & \quad \{ \ P_2 \ \} \\
 & \delta^*([[S]], a)
\end{array}
$$

This proves correctness of:

- $\{ \ P_2 \ \}\ [[S]] := \delta^*([[S]], a)\ \{ \ P_2(t, x := w, ax) \ \}$
- $\{ \ P_2 \ \}\ [[S]] := \delta^*([[S]], a)\ \{ \ P_2(t, x, h, L := w, a, wa, xL) \ \}$

Currently, the inner repetition looks as follows:

$[[S]] := \delta^*(q_0, W \uparrow 1);$
$\{ \ P_2(t, x, h, L := W \downarrow 1, W \uparrow 1, W, \varepsilon) \ \}$
$t, x, h, L := W \downarrow 1, W \uparrow 1, W, \varepsilon;$
$\{ \ P_{1..3} \ \}$
do $t :: wa \rightarrow \{ \ P_{1..3} \ \}$
  if $[[S]] \cap F \neq \varnothing \rightarrow t, x, h, L := w, a, wa, xL;$
  $[\!]$ $[[S]] \cap F = \varnothing \rightarrow t, x := w, ax;$
  fi; $\{ \ P_1 \ \wedge \ P_3 \ \wedge \ P_2([[S]] := \delta^*([[S]], a)) \ \}$
  $[[S]] := \delta^*([[S]], a)$
od; $\{ \ P_1 \ \wedge \ P_2 \ \wedge \ P_3 \ \wedge \ t = \varepsilon \ \}$
$\{ \ R_1 \ \wedge \ R_2 \ \}$

Combining this with the previous template gives a valid algorithm, however there is a better version.

Suppose P_2 holds and $[[S]] = \varnothing$. Then it holds that $\langle \forall y \ : \ y \in \Sigma^* \ : \ \delta^*(q_0, (yxL)^R) = \varnothing \rangle$ (once $[[S]]$ is empty, it remains empty). The proof of this is trivial ($\delta^*(q_0, (yxL)^R) = \delta^*([[S]], y^R)$).

In other words, once $[[S]] = \varnothing$ postcondition R_2 holds ($W = txL$ take $y = t$). If postcondition R_1 holds as well then the loop execution can be cut short.

$$
\begin{array}{ll}
 & R_1 \\
\equiv & \quad \{ \text{ def } \}
\end{array}
$$

$\langle \forall f \ : \ \varepsilon <_s f <_s h \ : \ \neg\langle \exists s \ : \ s \leq_p p \ : \ dst(fL, s) \leq k \rangle \rangle \ \wedge \ hL = W \ \wedge \ h \neq \varepsilon$

$\equiv$ { $P_3 : hL = W \ \wedge \ tx = h \ \wedge \ x \neq \varepsilon$ }

$\langle \forall f \ : \ \varepsilon <_s f <_s tx \ : \ \neg\langle \exists s \ : \ s \leq_p p \ : \ dst(fL, s) \leq k \rangle \rangle$

$\equiv$ { domain split, dummy transformation }

$\langle \forall f \ : \ \varepsilon <_s f <_s x \ : \ \neg\langle \exists s \ : \ s \leq_p p \ : \ dst(fL, s) \leq k \rangle \rangle$
$\wedge \ \langle \forall f \ : \ f <_s t \ : \ \neg\langle \exists s \ : \ s \leq_p p \ : \ dst(fxL, s) \leq k \rangle \rangle$

$\equiv$ { P_1 }

$\langle \forall f \ : \ f <_s t \ : \ \neg\langle \exists s \ : \ s \leq_p p \ : \ dst(fxL, s) \leq k \rangle \rangle$

$\equiv$ { automaton: $\delta^*(q_0, u^R) \cap F \neq \varnothing \equiv \langle \exists s \ : \ s \leq_p p \ : \ dst(u, s) \leq k \rangle$ }

$\langle \forall f \ : \ f <_s t \ : \ \delta^*(q_0, (fxL)^R) \cap F = \varnothing \rangle$

$\equiv$ { $P_2 \ \wedge \ [[S]] = \varnothing$, hence $\langle \forall y \ : \ y \in \Sigma^* \ : \ \delta^*(q_0, (yxL)^R) = \varnothing \rangle$ }

$true$

So besides: { $P_{1..3} \ \wedge \ t = \varepsilon$ } { $(R_1 \ \wedge \ R_2)$ } it also holds that: { $P_{1..3} \ \wedge \ [[S]] = \varnothing$ } { $(R_1 \ \wedge \ R_2)$ }. Thus by strengthening the guard of the inner repetition to $t :: wa \ \wedge \ [[S]] \neq \varnothing$ a more efficient version is found.

Combining this with the template results in the ABNDM algorithm:

Algorithm 2.16.16(PS, ABNDM)

$P_0 : O = \langle \textbf{set } l, v, r \ : \ lvr = T \ \wedge \ l <_p U \ \wedge \ dst(v, p) \leq k \ : \ (l, v, r, dst(v, p)) \rangle$
$\wedge \ UWV = T \ \wedge \ (|W| = m - k \ \vee \ |WV| < m - k)$
$P_1 : \langle \forall f \ : \ \varepsilon <_s f <_s x \ : \ \neg\langle \exists s \ : \ s \leq_p p \ : \ dst(fL, s) \leq k \rangle \rangle$
$P_2 : [[S]] = \delta^*(q_0, (xL)^R)$
$P_3 : tx = h \ \wedge \ hL = W \ \wedge \ x \neq \varepsilon$
$R_1 : \langle \forall f \ : \ \varepsilon <_s f <_s h \ : \ \neg\langle \exists s \ : \ s \leq_p p \ : \ dst(fL, s) \leq k \rangle \rangle \ \wedge \ hL = W \ \wedge \ h \neq \varepsilon$
$R_2 : [[S]] = \delta^*(q_0, (W)^R)$

$O := \varnothing;$
{ $P_0(U, W, V := \varepsilon, T \uparrow (m - k), T \downarrow (m - k))$ }
$U, W, V := \varepsilon, T \uparrow (m - k), T \downarrow (m - k);$
{ P_0 }
do $|W| = m - k \rightarrow$ { P_0 }
  $[[S]] := \delta^*(q_0, W \uparrow 1);$
  { $P_2(t, x, h, L := W \downarrow 1, W \uparrow 1, W, \varepsilon)$ }
  $t, x, h, L := W \downarrow 1, W \uparrow 1, W, \varepsilon;$
  { $P_{1..3}$ }
  do $t :: wa \ \wedge \ [[S]] \neq \varnothing \rightarrow$ { $P_{1..3}$ }
    if $[[S]] \cap F \neq \varnothing \rightarrow t, x, h, L := w, a, wa, xL;$
    $[]$ $[[S]] \cap F = \varnothing \rightarrow t, x := w, ax;$
    fi; { $P_1 \ \wedge \ P_3 \ \wedge \ P_2([[S]] := \delta^*([[S]], a))$ }
    $[[S]] := \delta^*([[S]], a)$
  od; { $P_{1..3} \ \wedge \ (t = \varepsilon \ \vee \ [[S]] = \varnothing)$ }

$\{\ R_1\ \wedge\ R_2\ \}$
if $[[S]] \cap F \neq \varnothing \rightarrow$
$O := O \cup \langle \textbf{set}\ v, r\ :\ vr = TL\ \wedge\ dst(Wv, p) \leq k\ :\ (U, Wv, r, dst(Wv, p))\rangle$
$[\!]$ $[[S]] \cap F = \varnothing \rightarrow$ skip
fi; $\{\ P_0(U, W, V := Uh, L(V \uparrow |h|), V \downarrow |h|)\ \}$
$U, W, V := Uh, L(V \uparrow |h|), V \downarrow |h|$;
od; $\{\ P_0\ \wedge\ |W| \neq m - k\ \}$
$\{\ R\ \}$

Chapter 3

Auxiliary proofs

This chapter contains several proofs of properties related to various definitions and automata in the text. Those readers that have no interest in formal proofs can freely skip this chapter.

3.1 Properties of the edit distance

In this section the properties of the edit distance which are used in the rest of the document are proven.

Property 3.1.1. *For $s \in \Sigma^*$ and $t \in \Sigma^*$:*

$$\langle \forall s, t \ : \ : \ dst(s,t) = dst(s^R, t^R) \rangle \tag{3.1}$$

Proof: Proof with induction on s and t where s, t, x, y are strings and a, b are characters:

- base $s = \varepsilon$:

  $dst(\varepsilon^R, t^R)$
  $=$ { property of Reverse }
  $dst(\varepsilon, t^R)$
  $=$ { def dst }
  $|t^R|$
  $=$ { property of Reverse }
  $|t|$
  $=$ { def dst }
  $dst(\varepsilon, t)$

- base $t = \varepsilon$: Similar to case $s = \varepsilon$.

- Induction Hypothesis:

  $$dst(x,y) = dst(x^R, y^R) \ \wedge \ dst(x, yb) = dst(x^R, (yb)^R) \ \wedge \ dst(xa, y) = dst((xa)^R, y^R)$$

- step: $s = xa$ and $t = yb$:

$$
\begin{array}{ll}
 & dst((xa)^R, (yb)^R) \\
= & \quad \{ \text{ Reverse } \} \\
 & dst(a(x^R), b(y^R)) \\
= & \quad \{ \text{ def } dst \ (3^{rd} \text{ rule}) \ \} \\
 & (dst(x^R, y^R) + \delta(a,b)) \downarrow (dst(x^R, b(y^R)) + 1) \downarrow (dst(a(x^R), y^R) + 1) \\
= & \quad \{ \text{ Reverse } \} \\
 & (dst(x^R, y^R) + \delta(a,b)) \downarrow (dst(x^R, (yb)^R) + 1) \downarrow (dst((xa)^R, y^R) + 1) \\
= & \quad \{ \text{ induction hypothesis } \} \\
 & (dst(x,y) + \delta(a,b)) \downarrow (dst(x, yb) + 1) \downarrow (dst(xa, y) + 1) \\
= & \quad \{ \text{ def } dst(4^{th} \text{ rule}) \ \} \\
 & dst(xa, yb)
\end{array}
$$

□

Property 3.1.2. *For all $x \in \Sigma^*, a \in \Sigma$ and $y \in \Sigma^*$:*

$$-1 \leq dst(ax, y) - dst(x, y) \leq 1$$

Proof: This property has the following two proof obligations:

$$dst(ax, y) \leq dst(x, y) + 1 \qquad (a)$$

$$dst(x, y) - 1 \leq dst(ax, y) \qquad (b)$$

Proof of (a):

- Case $y = \varepsilon$:

$$dst(ax, \varepsilon) = |ax| = |x| + 1 = dst(x, \varepsilon) + 1$$

- Case $y = ct$:

$$
\begin{array}{ll}
 & dst(ax, ct) \\
= & \quad \{ \text{ dst } \} \\
 & (dst(x,t) + \delta(a,c)) \downarrow (dst(ax, t) + 1) \downarrow (dst(x, ct) + 1) \\
\leq & \quad \{ \ \downarrow \ \} \\
 & dst(x, ct) + 1
\end{array}
$$

Proof of (b): $\langle \forall x, y \ : \ : \ dst(x,y) - 1 \leq dst(ax, y) \rangle$
Proof with structural induction on y:

- The base case is $y = \varepsilon$. The proof obligation becomes: $\langle \forall x \ : : \ dst(x, \varepsilon) - 1 \leq dst(ax, \varepsilon) \rangle$. This proof is trivial, since $dst(x, \varepsilon) = |x|$ $(\forall x)$.
- Induction Hypothesis: $\langle \forall x, z \ : \ z <_s ct \ : \ dst(x, z) - 1 \leq dst(ax, z) \rangle$
- Case $y = ct$. To prove: $\langle \forall x, y \ : : \ dst(x, ct) - 1 \leq dst(ax, ct) \rangle$ There are two cases to consider:
  - Case $x = \varepsilon$: $dst(\varepsilon, ct) - 1 \leq dst(a, ct)$

    $$\begin{array}{ll} & dst(a, ct) \\ = & \quad \{ \text{ dst } \} \\ & (dst(\varepsilon, t) + \delta(a, c)) \downarrow (dst(a, t) + 1) \downarrow (dst(\varepsilon, ct) + 1) \\ \geq & \quad \{ \text{ Induction Hypothesis } \} \\ & (dst(\varepsilon, t) + \delta(a, c)) \downarrow dst(\varepsilon, t) \downarrow (dst(\varepsilon, ct) + 1) \\ \geq & \quad \{ \ 0 \leq \delta(a, c) \leq 1, \text{ dst } \} \\ & |t| \\ = & \quad \{ \text{ dst } \} \\ & dst(\varepsilon, ct) - 1 \end{array}$$

  - Case $x = bs$. The proof obligation: $dst(bs, ct) - 1 \leq dst(abs, ct)$.

    $$\begin{array}{ll} & dst(abs, ct) \\ = & \quad \{ \text{ dst } \} \\ & (dst(bs, t) + \delta(a, c)) \downarrow (dst(bs, ct) + 1) \downarrow (dst(abs, t) + 1) \\ \geq & \quad \{ \text{ Induction Hypothesis } \} \\ & (dst(bs, t) + \delta(a, c)) \downarrow (dst(bs, ct) + 1) \downarrow dst(bs, t) \\ = & \quad \{ \ \downarrow, \ 0 \leq \delta(a, c) \leq 1 \ \} \\ & dst(bs, t) \downarrow (dst(bs, ct) + 1) \\ \geq & \quad \{ \ \downarrow, \text{ see remark below } \} \\ & (dst(s, t) + \delta(b, c) - 1) \downarrow dst(bs, t) \downarrow dst(s, ct) \downarrow (dst(bs, ct) - 1) \\ = & \quad \{ \text{ dst, } - \text{ over } \downarrow \ \} \\ & (dst(bs, ct) - 1) \downarrow (dst(bs, ct) - 1) \\ = & \quad \{ \ \downarrow \text{ is idempotent } \} \\ & dst(bs, ct) - 1 \end{array}$$

Remark: The step which replaces $dst(bs, t) \downarrow (dst(bs, ct) + 1)$ with $dst(s, t) \downarrow dst(bs, t) \downarrow dst(s, ct) \downarrow (dst(bs, ct) - 1)$ might not appear to be the most logical to make at first glance. The goal at that point is to show that $dst(bs, t) \downarrow (dst(bs, ct) + 1$ is larger than $dst(bs, ct) - 1$. This clearly holds for the right term. It is not so obvious for the left term. The key to this step is to recognize the left term as a term of $dst(bs, ct) - 1$ which is by definition equal to $(dst(s, t) + \delta(b, c) - 1) \downarrow (dst(bs, t) + 1 - 1) \downarrow (dst(b, ct) + 1 - 1)$. Therefore introduce the remaining terms and head towards $dst(bs, ct) - 1$.

□

Property 3.1.3. *For any string $v \in \Sigma^*$, pattern $p \in \Sigma^*$ and $k \leq m$:*

$$\langle \forall j \ : \ dst(v,p) \leq j \leq k \ : \ \langle \exists s \ : \ s \leq_s v \ : \ dst(s,p) = j \rangle \rangle$$

Proof: Case $dst(v,p) = j$ is trivial, therefore suppose $dst(v,p) < j \leq k$:
The length of v is at least $m - dst(v,p)$. Since $k \leq m$ and $dst(v,p) < j \leq k$ it also holds that $0 \leq m - j \leq m - dst(v,p)$. Therefore v has a suffix z of length $m - j$. Because $|z| = m - j$ the distance between z and p is at least j.

So there exists a y with $z \leq_s y \leq_s v$ with $dst(y,p) = j$ because $dst(v,p) < j \leq dst(z,p)$ and the fact that for all $a \in \Sigma, x \in \Sigma^*$: $dst(ax,p)$ and $dst(x,p)$ differ at most 1 (Property 3.1.2).

□

Property 3.1.4. *For $w \in \Sigma^*, p \in \Sigma^*$ and $0 \leq i \leq k$*

$$\langle \exists y \ : \ y \leq_s w \ : \ dst(y,p) = i \rangle \equiv \langle \downarrow y \ : \ y \leq_s w \ : \ dst(y,p) \rangle \leq i$$

Proof:

- Suppose $\langle \exists y \ : \ y \leq_s w \ : \ dst(y,p) = i \rangle$. To prove: $\langle \downarrow y \ : \ y \leq_s w \ : \ dst(y,p) \rangle \leq i$. This is trivial, the assumption provides a witness with distance equal to i, therefore the minimum is at most i.
- Suppose $\langle \downarrow y \ : \ y \leq_s w \ : \ dst(y,p) \rangle \leq i$. To prove: $\langle \exists y \ : \ y \leq_s w \ : \ dst(y,p) = i \rangle$.

$$\begin{array}{ll} & \langle \downarrow y \ : \ y \leq_s w \ : \ dst(y,p) \rangle \leq i \\ \equiv & \quad \{ \ \downarrow \ \} \\ & \langle \exists y \ : \ y \leq_s w \ : \ dst(y,p) \leq i \rangle \\ \Rightarrow & \quad \{ \text{ let } z : z \leq_s w \ \wedge \ dst(z,p) \leq i, \text{ Property 3.1.3, } i \leq k \leq m \ \} \\ & \langle \exists y \ : \ y \leq_s z \ : \ dst(y,p) = i \rangle \\ \Rightarrow & \quad \{ \ z \leq_s w \ \} \\ & \langle \exists y \ : \ y \leq_s w \ : \ dst(y,p) = i \rangle \end{array}$$

□

3.2 Dynamic Programming Matrix

The dynamic programming matrix is characterized by the following formula: (for $x \leq_p p, y \leq_p T$)

$$M_{x,y} = \langle \downarrow r \ : \ r \leq_s y \ : \ dst(r,x) \rangle$$

Cell (x,y) of matrix M is equal to the distance between x and the minimal distance between x and a suffix of y (i.e. the best matching suffix of y).

Below a recurrence relation for computing matrix M is derived:

Proof: (for $x \leq_p p, y \leq_p T$): Proof with structural induction on x and y.

- base $x = \varepsilon$:

$$
\begin{array}{ll}
 & M_{\varepsilon,y} \\
= & \quad \{ \text{ specification } \} \\
 & \langle \downarrow r \ : \ r \leq_s y \ : \ dst(r, \varepsilon) \rangle \\
= & \quad \{ \text{ def } dst \ \} \\
 & \langle \downarrow r \ : \ r \leq_s y \ : \ |r| \rangle \\
= & \quad \{ \ \downarrow, \ \varepsilon \leq_s y \ \} \\
 & 0
\end{array}
$$

- base $y = \varepsilon$:

$$
\begin{array}{ll}
 & M_{x,\varepsilon} \\
= & \quad \{ \text{ specification } \} \\
 & \langle \downarrow r \ : \ r \leq_s \varepsilon \ : \ dst(r, x) \rangle \\
= & \quad \{ \text{ one point rule } \} \\
 & dst(\varepsilon, x) \\
= & \quad \{ \text{ def } dst \ \} \\
 & |x|
\end{array}
$$

- Induction Hypothesis:

$$
\begin{array}{l}
M_{s,t} = \langle \downarrow r \ : \ r \leq_s t \ : \ dst(r, s) \rangle \ \wedge \\
M_{sa,t} = \langle \downarrow r \ : \ r \leq_s t \ : \ dst(r, sa) \rangle \ \wedge \\
M_{s,tb} = \langle \downarrow r \ : \ r \leq_s tb \ : \ dst(r, s) \rangle
\end{array}
$$

- step $x = sa, y = tb$:

$$
\begin{array}{ll}
 & M_{sa,tb} \\
= & \quad \{ \text{ specification } \} \\
 & \langle \downarrow r \ : \ r \leq_s tb \ : \ dst(r, sa) \rangle \\
= & \quad \{ \text{ split off } r = \varepsilon \ \} \\
 & \langle \downarrow r \ : \ \varepsilon <_s r \leq_s tb \ : \ dst(r, sa) \rangle \downarrow dst(\varepsilon, sa) \\
= & \quad \{ \text{ dummy transformation } r := rb \ \} \\
 & \langle \downarrow r \ : \ r \leq_s t \ : \ dst(rb, sa) \rangle \downarrow dst(\varepsilon, sa) \\
= & \quad \{ \text{ def } dst \ \} \\
 & \langle \downarrow r \ : \ r \leq_s t \ : \ (dst(r, s) + \delta(b, a)) \downarrow (dst(r, sa) + 1) \downarrow (dst(rb, s) + 1) \rangle \downarrow |sa| \\
= & \quad \{ \text{ associativity and commutativity of } \downarrow \ \} \\
 & \langle \downarrow r \ : \ r \leq_s t \ : \ dst(r, s) + \delta(b, a) \rangle \downarrow
\end{array}
$$

$$
\begin{array}{ll}
 & \langle \downarrow r \ : \ r \leq_s t \ : \ dst(r, sa) + 1 \rangle \downarrow \\
 & \langle \downarrow r \ : \ r \leq_s t \ : \ dst(rb, s) + 1 \rangle \downarrow \\
 & |sa| \\
= & \quad \{ \text{ dummy transformation } \} \\
 & \langle \downarrow r \ : \ r \leq_s t \ : \ dst(r, s) + \delta(b, a) \rangle \downarrow \\
 & \langle \downarrow r \ : \ r \leq_s t \ : \ dst(r, sa) + 1 \rangle \downarrow \\
 & \langle \downarrow r \ : \ \varepsilon <_s r \leq_s tb \ : \ dst(r, s) + 1 \rangle \\
 & |sa| \\
= & \quad \{ \ dst(\varepsilon, s) + 1 = |s| + 1 = |sa| \ \} \\
 & \langle \downarrow r \ : \ r \leq_s t \ : \ dst(r, s) + \delta(b, a) \rangle \downarrow \\
 & \langle \downarrow r \ : \ r \leq_s t \ : \ dst(r, sa) + 1 \rangle \downarrow \\
 & \langle \downarrow r \ : \ r \leq_s tb \ : \ dst(r, s) + 1 \rangle \\
= & \quad \{ + \text{ over } \downarrow \ \} \\
 & (\langle \downarrow r \ : \ r \leq_s t \ : \ dst(r, s) \rangle + \delta(b, a)) \downarrow \\
 & (\langle \downarrow r \ : \ r \leq_s t \ : \ dst(r, sa) \rangle + 1) \downarrow \\
 & (\langle \downarrow r \ : \ r \leq_s tb \ : \ dst(r, s) \rangle + 1) \\
= & \quad \{ \text{ IH } \} \\
 & (M_{s,t} + \delta(b, a)) \downarrow (M_{sa,t} + 1) \downarrow (M_{s,tb} + 1)
\end{array}
$$

□

As shown in the derivation, matrix M is computed by the following function:
$M_{\varepsilon,y} \leftarrow 0$
$M_{x,\varepsilon} \leftarrow |x|$
$M_{xa,yb} \leftarrow (M_{x,y} + \delta(a, b)) \downarrow (M_{x,yb} + 1) \downarrow (M_{xa,y} + 1) \quad (xa \leq_p p, yb \leq_p T)$

3.2.1 Matrix properties

This section gathers the proofs of the properties of the dynamic programming matrix.

Property 3.2.1. *Diagonals are ascending: (s, t, x, y are strings and a, b, c, d are characters)*

$$\langle \forall s, t, c, d \ : \ : \ M_{s,t} \leq M_{sc,td} \rangle \tag{3.2}$$

Proof: This is proven with induction on s.

- base $s = \varepsilon$:
  To prove: $\langle \forall t, c, d \ : \ : \ M_{\varepsilon,t} \leq M_{c,td} \rangle$
  proof with induction on t:
  - base $t = \varepsilon$:

$$
\begin{array}{ll}
 & M_{c,d} \\
= & \quad \{ \text{ def } M \ \}
\end{array}
$$

$$
\begin{array}{ll}
& (M_{\varepsilon,\varepsilon} + \delta(c,d)) \downarrow (M_{\varepsilon,d} + 1) \downarrow (M_{c,\varepsilon} + 1) \\
= & \quad \{ \text{ def } M \ \} \\
& \delta(c,d) \downarrow 1 \downarrow 2 \\
\geq & \quad \{ \ 0 \leq \delta(c,d) \leq 1 \ \} \\
& 0 \\
= & \quad \{ \text{ def } M \ \} \\
& M_{\varepsilon,\varepsilon}
\end{array}
$$

– Induction Hypothesis:

$$\langle \forall t, c, d \ : \ t \leq_p y \ : \ M_{\varepsilon,t} \leq M_{c,td} \rangle \qquad (IH_0)$$

– step: $t = yb$:

$$
\begin{array}{ll}
& M_{c,ybd} \\
= & \quad \{ \text{ def } M \ \} \\
& (M_{\varepsilon,yb} + \delta(c,d)) \downarrow (M_{\varepsilon,ybd} + 1) \downarrow (M_{c,yb} + 1) \\
\geq & \quad \{ \ IH_0 \ \} \\
& (M_{\varepsilon,yb} + \delta(c,d)) \downarrow (M_{\varepsilon,ybd} + 1) \downarrow (M_{\varepsilon,y} + 1) \\
= & \quad \{ \text{ def } M \ \} \\
& \delta(c,d) \downarrow 1 \downarrow 1 \\
\geq & \quad \{ \ 0 \leq \delta(c,d) \leq 1 \ \} \\
& 0 \\
= & \quad \{ \text{ def } M \ \} \\
& M_{\varepsilon,yb}
\end{array}
$$

Which proves the basic case: $\langle \forall t, c, d \ : \ : \ M_{\varepsilon,t} \leq M_{c,td} \rangle$

- Induction Hypothesis:

$$\langle \forall s, t, c, d \ : \ s \leq_p x \ : \ M_{s,t} \leq M_{sc,td} \rangle \qquad (IH_1)$$

- step: $s = xa$:
  To prove: $\langle \forall t, c, d \ : \ : \ M_{xa,t} \leq M_{xac,td} \rangle$
  proof with induction on t:

  – base $t = \varepsilon$:

$$
\begin{array}{ll}
& M_{xac,d} \\
= & \quad \{ \text{ def } M \ \} \\
& (M_{xa,\varepsilon} + \delta(c,d)) \downarrow (M_{xa,d} + 1) \downarrow (M_{xac,\varepsilon} + 1) \\
\geq & \quad \{ \ IH_1 \ \} \\
& (M_{xa,\varepsilon} + \delta(c,d)) \downarrow (M_{x,\varepsilon} + 1) \downarrow (M_{xac,\varepsilon} + 1)
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ def } M \ \} \\
& (|xa| + \delta(c,d)) \downarrow (|x|+1) \downarrow (|xac|+1) \\
\geq & \{ \ 0 \leq \delta(c,d) \leq 1 \ \} \\
& |x| + 1 \\
= & \{ \text{ def } M \ \} \\
& M_{xa,\varepsilon}
\end{array}
$$

– Induction Hypothesis:

$$\langle \forall t, c, d \ : \ t \leq_p y \ : \ M_{xa,t} \leq M_{xac,td} \rangle \qquad (IH_2)$$

– step $t = yb$:

$$
\begin{array}{ll}
& M_{xac,ybd} \\
= & \{ \text{ def } M \ \} \\
& (M_{xa,yb} + \delta(c,d)) \downarrow (M_{xa,ybd}+1) \downarrow (M_{xac,yb}+1) \\
\geq & \{ \ IH_2 : M_{xa,y} \leq M_{xac,yd} (\forall d) \ \} \\
& (M_{xa,yb} + \delta(c,d)) \downarrow (M_{xa,ybd}+1) \downarrow (M_{xa,y}+1) \\
\geq & \{ \ IH_1 : M_{x,t} \leq M_{xc,td} (\forall t,c,d) \ \} \\
& (M_{xa,yb} + \delta(c,d)) \downarrow (M_{x,yb}+1) \downarrow (M_{xa,y}+1) \\
\geq & \{ \ 0 \leq \delta(c,d) \leq 1 \ \} \\
& M_{xa,yb} \downarrow (M_{x,yb}+1) \downarrow (M_{xa,y}+1) \\
= & \{ \ 0 \leq \delta(a,b) \leq 1 \ \} \\
& M_{xa,yb} \downarrow (M_{xa,yb} + \delta(a,b)) \downarrow (M_{x,yb}+1) \downarrow (M_{xa,y}+1) \\
\geq & \{ \ IH_1 : M_{x,t} \leq M_{xc,td} (\forall t,c,d) \ \} \\
& M_{xa,yb} \downarrow (M_{x,y} + \delta(a,b)) \downarrow (M_{x,yb}+1) \downarrow (M_{xa,y}+1) \\
= & \{ \text{ def } M \ \} \\
& M_{xa,yb} \downarrow M_{xa,yb} \\
= & \{ \ \downarrow \text{ is idempotent } \} \\
& M_{xa,yb}
\end{array}
$$

Remarks: In the last derivation IH_1 has been used twice. The proof can be shortened by postponing the first time it is applied, but I believe the current version is easier to follow.

Motivation for the above steps. Both IH_1 and IH_2 will be needed in the proof, or there would not have been the need for induction, so apply them first. This results in:

$$(M_{xa,yb} + \delta(c,d)) \downarrow (M_{x,yb}+1) \downarrow (M_{xa,y}+1)$$

The target is $M_{xa,yb}$ which is by definition of M equal to:

$$(M_{x,y} + \delta(a,b)) \downarrow (M_{x,yb}+1) \downarrow (M_{xa,y}+1)$$

So the first term must be manipulated: remove $\delta(c,d)$, introduce $\delta(a,b)$, etc...

□

3.3 APM Automaton

This section formally describes the approximate pattern matching automaton (APM) and proves several related properties.

For an informal description of the automaton refer to Section 2.9 on page 29 and Figure 2.1 on page 29.

The APM automaton is formally defined as the five tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$ with

- State set Q where each state in Q is identified by row and column indices (state (i, j) is the j^{th} state on the i^{th} row) (i.e. $Q = \langle \textbf{set } i, j \ : \ 0 \leq i \leq k \ \wedge \ 0 \leq j \leq m \ : \ (i, j) \rangle$).
- Σ is the alphabet. The choice of alphabet is not important for the algorithms described here, typically the set of ASCII characters, (subsets of) all unicode characters or {a,c,g,t} (in DNA processing) is used.
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function (see below).
- The initial state q_0 is equal to $(0, 0)$.
- The set of final states F is equal to the last column (i.e. $F = \langle \textbf{set } i \ : \ 0 \leq i \leq k \ : \ (i, m) \rangle$).

Since the ε-closure are all states reachable by following zero or more ε-transitions, the following equation holds:

$$\varepsilon\text{-closure}((i, j)) = \langle \textbf{set } l \ : \ 0 \leq l \leq ((k - i) \downarrow (m - j)) \ : \ (i + l, j + l) \rangle$$

The following relation follows from Figure 2.1 on page 29, where $q_0 \in Q, w \in \Sigma^*$ and $a \in \Sigma$:

$$\begin{array}{lcll}
(0,0) \in \delta^*(q_0, wa) & \equiv & true & \\
(0,j) \in \delta^*(q_0, wa) & \equiv & (0, j-1) \in \delta^*(q_0, w) \ \wedge \ p[j-1] = a & (\text{for } 1 \leq j \leq m) \\
(i,0) \in \delta^*(q_0, wa) & \equiv & (i-1, 0) \in \delta^*(q_0, w) & (\text{for } 1 \leq i \leq k) \\
(i,j) \in \delta^*(q_0, wa) & \equiv & (i-1, j-1) \in \delta^*(q_0, w) \ \vee \ (i-1, j) \in \delta^*(q_0, w) & \\
 & & \vee \ ((i, j-1) \in \delta^*(q_0, w) \ \wedge \ p[j-1] = a) & \\
 & & \vee \ (i-1, j-1) \in \delta^*(q_0, wa) & (\text{for } 1 \leq i \leq k \\
 & & & \wedge \ 1 \leq j \leq m)
\end{array}$$

The first formula states that the initial state is always enclosed in the result which follows from the Σ loop on state $(0, 0)$. Formula two and three are special cases of formula four for the first row and column respectively. Formula four has four components:

- $(i - 1, j - 1) \in \delta^*(q_0, w)$: the diagonal Σ transition (this is a substitution operation)
- $(i - 1, j) \in \delta^*(q_0, w)$: the vertical Σ transition (this is a deletion)
- $(i, j - 1) \in \delta^*(q_0, w) \ \wedge \ p[j - 1] = a$: the horizontal transition (this is a match)
- $(i - 1, j - 1) \in \delta^*(q_0, wa)$: the vertical ε-transition (this is an insertion).

3.3.1 Automaton properties

In this section several properties of the automaton described above will de described and proven. These properties will be used in several proofs in the following sections.

Property 3.3.1. *For $0 \leq i \leq k$ and $w \in \Sigma^*$:*

$$(i, m) \in \delta^*(q_0, w) \equiv \langle \exists x, y \ : \ xy = w \ : \ dst(y, p) = i \rangle$$

Proof:

$\Rightarrow$ Suppose $(i, m) \in \delta^*(q_0, w)$
To prove: $\langle \exists x, y \ : \ xy = w \ : \ dst(y, p) = i \rangle$

If $(i, m) \in \delta^*(q_0, w)$ then there exists a path $(0, 0) \xrightarrow{w} (i, m)$. This path is of the form: $(0, 0) \xrightarrow{u} (0, 0) \xrightarrow{v} (i, m)$ with $uv = w$.

The distance between v and p is the minimum number of edit operations to make them equal. Since there is a path $(0, 0) \xrightarrow{v} (i, m)$ and the arrows in the automaton correspond to the different edit operations, there exists a series of edit operations which make v and p equal with distance i. However, this might not be minimal. So one can only conclude that $dst(v, p) \leq i$.

Consider the following two cases:

- $dst(v, p) = i$. Strings u and v are witnesses which prove the existential quantification.
- $dst(v, p) < i$. Property 3.1.3 on page 74 guarantees the existance of x and y such that $xy = v$ and $dst(y, p) = i$. Strings ux and y are witnesses which prove the existential quantification.

$\Leftarrow$ Suppose $xy = w \ \wedge \ dst(y, p) = i$
To prove: $(i, m) \in \delta^*(q_0, w)$

Obviously $(0, 0) \xrightarrow{x} (0, 0)$. If it is also true that $(0, 0) \xrightarrow{y} (i, m)$ then the property has been proven.

Because $dst(y, p) = i$, there exists a series of edit operations such that y becomes equal to p. By following the corresponding arrows in the automaton a path can be found to state (i, m).

□

Property 3.3.2. *Suppose, $i \leq k$ and $j \leq m - k$*

$$(i, i + j) \in \delta^*(q_0, w) \equiv \langle \forall l \ : \ i \leq l \leq k \ : \ (l, l + j) \in \delta^*(q_0, w) \rangle$$

This lemma states that whenever a state on a diagonal is active, all subsequent states on the same diagonal are active as well. This follows from the transition function of the automaton (note the ε-transition on the diagonals).

The proof of this lemma is trivial. The right hand side of the equation clearly implies the left hand side. The left hand side implies the right hand side due to the definition of δ^*.

Property 3.3.3.

$$\begin{array}{l}\langle \forall i,j \ : \ 0 \leq i \leq k \ \wedge \ i \leq j \leq m \ : \\ \quad (i,j) \in \delta^*(q_0,w) \equiv \langle \forall a \ : \ i \leq a \leq (k \downarrow j) \ : \ (a,j) \in \delta^*(q_0,w) \rangle \\ \rangle\end{array}$$

This predicate states that if a state is active, then all states in the same column which correspond with a greater distance are active as well (with the exception of all states below the diagonal through the starting state).

Proof:

$$\begin{aligned} hc &= \text{horizontal transitions} \\ dc &= \text{diagonal } (\Sigma) \text{ transitions} \\ ec &= \varepsilon - \text{transitions} \\ vc &= \text{vertical transitions} \end{aligned}$$

The right side of the equivalence clearly implies the left (note that $i \leq (k \downarrow j)$ because $i \leq k$ and $i \leq j$). Therefore it is sufficient to prove: $\langle \forall a \ : \ i \leq a \leq (k \downarrow j) \ : \ (a,j) \in \delta^*(q_0,w) \rangle$ provided $(i,j) \in \delta^*(q_0,w)$.

consider two cases:

- case $i = k$: This case is trivial, the quantification has a one-point-domain ($(k \downarrow j) = k$ because $i \leq j$) which makes the formula equal to the assumption that $(i,j) \in \delta^*(q_0,w)$.
- case $0 \leq i < k$):
  Suppose $(i,j) \in \delta^*(q_0,w)$ for some $0 \leq i < k$ and $i \leq j \leq m$.

  If $(i,j) \in \delta^*(q_0,w)$ then there exists a path from $(0,0)$ —the initial state (q_0)—to (i,j). This path travels i steps in the vertical direction by either: vertical transitions, diagonal transitions or $\varepsilon-$transitions: $i = vc + dc + ec$.

  The path also travels j transitions in the horizontal direction by either: horizontal transitions, diagonal transitions or epsilon transitions: $j = hc + dc + ec$.

  To prove the lemma, it must be shown that there exists a path from $(0,0)$ to (a,j) for $i \leq a \leq (k \downarrow j)$.

  There are at least $a - i$ horizontal transitions:

$$\begin{array}{ll} & j = hc + dc + ec \\ \Rightarrow & \quad \{ \ 0 \leq vc \ \} \\ & j \leq hc + vc + dc + ec \\ \equiv & \quad \{ \ i \ \} \\ & j \leq hc + i \\ \Rightarrow & \quad \{ \ a \leq (k \downarrow j), \text{ therefore } a \leq j \ \} \\ & a \leq hc + i \\ \equiv & \quad \{ \ \text{math} \ \} \\ & a - i \leq hc \end{array}$$

Note that for some $(x, y), (u, v)$ and $c \in \Sigma$ on a path ending in (i, j) with $i < k$ and $(x, y) \xrightarrow{c} (u, v)$ it is also true that $(x + 1, y) \xrightarrow{c} (u + 1, v)$. Due to this property, a (partial) path in the automaton can be moved down to yield another valid (partial) path (provided the original path ends in a row smaller than row k).

Consider a path $(0, 0) \xrightarrow{\Sigma^*} (u, v) \xrightarrow{c} (u, v+1) \xrightarrow{\Sigma^*} (i, j)$. By moving the segment $(u, v+1) \xrightarrow{\Sigma^*} (i, j)$ one row downwards and replacing the horizontal transition $(u, v) \xrightarrow{c} (u, v + 1)$ by a diagonal transition $(u, v) \xrightarrow{c} (u + 1, v + 1)$ the following (valid) path is found: $(0, 0) \xrightarrow{\Sigma^*} (u, v) \xrightarrow{c} (u + 1, v + 1) \xrightarrow{\Sigma^*} (i + 1, j)$.

Hence, by replacing a horizontal transition with a diagonal transition, a new valid path is acquired which ends in a row with a distance which is one higher than the end of the original path.

Now replace $a-i$ horizontal transitions by diagonal transitions, which is possible because there are at least $a - i$ horizontal transitions.

this gives a new path with:

$$\begin{aligned} hc' &= hc - (a - i) \\ dc' &= dc + (a - i) \\ ec' &= ec \\ vc' &= vc \end{aligned}$$

This new path ends in a state with the following coordinate:

$$\begin{aligned} & (vc' + dc' + ec', hc' + dc' + ec') \\ \equiv & \quad \{ \text{ def } \} \\ & (vc + dc + (a - i) + ec, hc - (a - i) + dc + (a - i) + ec) \\ \equiv & \quad \{ \text{ math } \} \\ & (vc + dc + (a - i) + ec, hc + dc + ec) \\ \equiv & \quad \{ \text{ def } i \text{ and } j \ \} \\ & (i + (a - i), j) \\ \equiv & \quad \{ \text{ math } \} \\ & (a, j) \end{aligned}$$

Thus the new path is a path from $(0, 0)$ to (a, j) which proves the lemma.

□

Lemma 3.3.4. *(Indirect Equality)*

$$\langle \forall v \ : \ : \ a = v \Rightarrow b = v \rangle \equiv a = b$$

Proof:

$\langle \forall v \ : \ : \ a = v \Rightarrow b = v \rangle \equiv a = b$

$\equiv$ { predicate calculus }

$\langle \forall v \ : \ a = v \ : \ b = v \rangle \equiv a = b$

$\equiv$ { one-point rule }

$b = a \equiv a = b$

$\equiv$ { predicate calculus }

$true$

□

The following property states that the ε-transitions can be ignored when computing the minimal active state on a diagonal.

Property 3.3.5.

$$\langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ (l, l+j) \in \delta^*(q_0, uc) \ : \ l \rangle = \langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ g(j, l, c) \ : \ l \rangle$$

where function g is defined as follows (for $1 \leq l \leq k$):

$$\begin{aligned} g(j, 0, uc) =& (0, j-1) \in \delta^*(q_0, u) \ \wedge \ p[j-1] = c \\ g(j, l, uc) =& (l-1, l+j-1) \in \delta^*(q_0, u) \ \vee \\ & (l-1, l+j) \in \delta^*(q_0, u) \ \vee \\ & ((l, l+j-1) \in \delta^*(q_0, u) \ \wedge \ p[l+j-1] = c) \end{aligned}$$

Due to indirect equality (lemma (3.3.4)), it is sufficient to prove ($\forall v$):

$$\langle \downarrow l \ : \ 0 \leq l \leq k \wedge (l, l+j) \in \delta^*(q_0, uc) \ : \ l \rangle = v \Rightarrow \langle \downarrow l \ : \ 0 \leq l \leq k \wedge g(j, l, uc) \ : \ l \rangle = v \quad (3.3)$$

Proof: Proof with case analysis on v:

- case $v = 0$:

  $\langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ (l, l+j) \in \delta^*(q_0, uc) \ : \ l \rangle = 0$

  $\equiv$ { $\downarrow$ }

  $(0, j) \in \delta^*(q_0, uc)$

  $\equiv$ { δ^*, def g }

  $g(j, 0, uc)$

  $\equiv$ { min }

  $\langle \downarrow l \ : \ 0 \leq l \leq k \ \wedge \ g(j, l, uc) \ : \ l \rangle = 0$

- case $v = b + 1$ $(b + 1 \leq k)$:

$$
\begin{aligned}
& \langle \downarrow l \ : \ 0 \le l \le k \ \wedge \ (l, l+j) \in \delta^*(q_0, uc) \ : \ l \rangle = b+1 \\
\equiv \quad & \{ \ \downarrow \ \} \\
& \langle \forall l \ : \ 0 \le l \le b \ : \ (l, l+j) \notin \delta^*(q_0, uc) \rangle \ \wedge \ (b+1, b+1+j) \in \delta^*(q_0, uc) \\
\equiv \quad & \{ \ \delta^*, \text{ def } g \ \} \\
& \langle \forall l \ : \ 0 \le l \le b \ : \ (l, l+j) \notin \delta^*(q_0, uc) \rangle \ \wedge \ (g(j, b+1, uc) \ \vee \ (b, b+j) \in \delta^*(q_0, uc)) \\
\equiv \quad & \{ \ (b, b+j) \notin \delta^*(q_0, uc) \ \} \\
& \langle \forall l \ : \ 0 \le l \le b \ : \ (l, l+j) \notin \delta^*(q_0, uc) \rangle \ \wedge \ g(j, b+1, uc) \\
\equiv \quad & \{ \ \delta^*, \text{ def } g, \text{ domain split } \} \\
& \langle \forall l \ : \ 1 \le l \le b \ : \ \neg g(j, l, uc) \ \wedge \ (l-1, l+j-1) \notin \delta^*(q_0, uc) \rangle \\
& \ \wedge \ \neg g(j, 0, uc) \ \wedge \ g(j, b+1, uc) \\
\Rightarrow \quad & \{ \text{ pred. calc. } \} \\
& \langle \forall l \ : \ 1 \le l \le b \ : \ \neg g(j, l, uc) \rangle \ \wedge \ \neg g(j, 0, uc) \ \wedge \ g(j, b+1, uc) \\
\equiv \quad & \{ \text{ combine domains } \} \\
& \langle \forall l \ : \ 0 \le l \le b \ : \ \neg g(j, l, uc) \rangle \ \wedge \ g(j, b+1, uc) \\
\equiv \quad & \{ \ \downarrow \ \} \\
& \langle \downarrow l \ : \ 0 \le l \le k \ \wedge \ g(j, l, uc) \ : \ l \rangle = b+1
\end{aligned}
$$

- case $v = \infty$:

$$
\begin{aligned}
& \langle \downarrow l \ : \ 0 \le l \le k \ \wedge \ (l, l+j) \in \delta^*(q_0, uc) \ : \ l \rangle = \infty \\
\equiv \quad & \{ \ \downarrow \ \} \\
& \langle \forall l \ : \ 0 \le l \le k \ : \ (l, l+j) \notin \delta^*(q_0, uc) \rangle \\
\equiv \quad & \{ \ \delta^*, \text{ def } g, \text{ domain split } \} \\
& \langle \forall l \ : \ 1 \le l \le k \ : \ \neg g(j, l, uc) \ \wedge \ (l-1, l+j-1) \notin \delta^*(q_0, uc) \rangle \ \wedge \ \neg g(j, 0, uc) \\
\Rightarrow \quad & \{ \text{ calc } \} \\
& \langle \forall l \ : \ 1 \le l \le k \ : \ \neg g(j, l, uc) \rangle \ \wedge \ \neg g(j, 0, uc) \\
\equiv \quad & \{ \text{ combine domains } \} \\
& \langle \forall l \ : \ 0 \le l \le k \ : \ \neg g(j, l, uc) \rangle \\
\equiv \quad & \{ \ \downarrow \ \} \\
& \langle \downarrow l \ : \ 0 \le l \le k \ \wedge \ g(j, l, uc) \ : \ l \rangle = \infty
\end{aligned}
$$

This concludes the proof of (3.3) and therefore the proof of property (3.3.5).

□

3.4 ABNDM automaton

The ABNDM algorithm uses a special approximate suffix automaton (see Figure 3.1). The approximate suffix automaton is very similar to the approximate pattern matching automaton

(compare with Figure 2.1 on page 29).

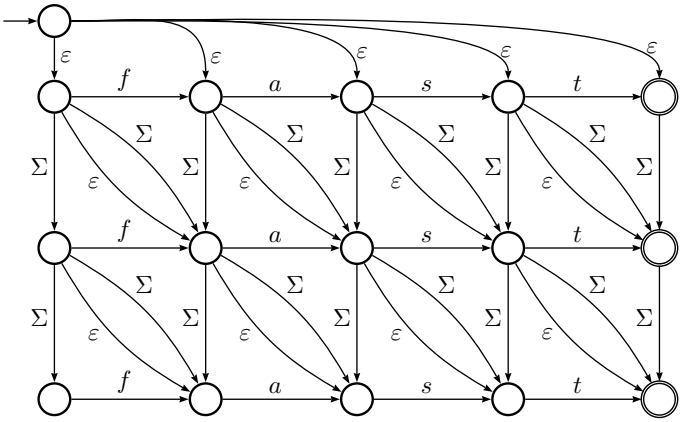


Figure 3.1: approximate suffix automaton on string *fast* with $k = 2$

There are some significant differences:

- There is no Σ-loop on the starting state.
- There is a separate starting state with ε-transitions to all states on the first row.

A suffix automaton accepts all suffixes of the pattern on which it is built. The approximate suffix automaton accepts all strings which have at most k differences with a suffix of the pattern on which it is built. The approximate suffix automaton is characterized by the following formula: (for $0 \leq i \leq k$)

$$(i, m) \in \delta^*(q_0, u) \equiv \langle \exists s \ : \ s \leq_s p \ : \ dst(u, s) = i \rangle$$

The ABNDM algorithm builds an approximate suffix automaton on the reverse pattern and feeds it a reverse substring of the text. By reversing both the pattern and the input, the automaton is transformed into an approximate prefix automaton. Proof:

$$\begin{aligned}
& (i, m) \in \delta^*(q_0, u^R) \\
\equiv \quad & \{ \text{ characterization of the automaton with } p := p^R \ \} \\
& \langle \exists s \ : \ s \leq_s p^R \ : \ dst(u^R, s) = i \rangle \\
\equiv \quad & \{ \ \leq_s \ \} \\
& \langle \exists s, t \ : \ ts = p^R \ : \ dst(u^R, s) = i \rangle \\
\equiv \quad & \{ \text{ property of reverse } \} \\
& \langle \exists s, t \ : \ s^R t^R = p \ : \ dst(u^R, s) = i \rangle \\
\equiv \quad & \{ \ dst(x, y) = dst(x^R, y^R) \ \} \\
& \langle \exists s, t \ : \ s^R t^R = p \ : \ dst(u, s^R) = i \rangle \\
\equiv \quad & \{ \text{ dummy transform } s := s^R, t := t^R, \text{ property of reverse } \}
\end{aligned}$$

$$
\begin{aligned}
& \langle \exists s, t \ : \ st = p \ : \ dst(u, s) = i \rangle \\
\equiv \quad & \{ \ \leq_p \ \} \\
& \langle \exists s \ : \ s \leq_p p \ : \ dst(u, s) = i \rangle
\end{aligned}
$$

From this property follows the validity of the following property of an automaton built on p^R:

$$\delta^*(q_0, u^R) \cap F \neq \varnothing \equiv \langle \exists s \ : \ s \leq_p p \ : \ dst(u, s) \leq k \rangle$$

Chapter 4

Epilogue

4.1 Conclusions

This text has presented a taxonomy of algorithms in the field of approximate pattern matching in strings. The algorithms have been ordered based on their structural relationship into families of similar algorithms as described in the taxonomy tree in Section 2 on page 11.

- The algorithms have all been described in a uniform manner. It has become possible to compare the algorithms and significant similarities and differences have been highlighted.
- Insignificant implementation details have been removed from the algorithms. This has made the core concepts of the algorithms visible and allows readers to understand the algorithms without having to work through irrelevant and/or language dependent details.
- There now exists a structured overview of some very interesting solutions to the approximate pattern matching problem. This makes it easier to find those algorithms that are capable of retrieving the required information from the text. For example, not all algorithms are equally well suited to report both start and ending position of occurrences.
- Formal proofs have been constructed for all algorithms in the taxonomy showing total correctness. In the literature one only encounters operational descriptions of the algorithms with proof of important properties, but never a complete proof of the algorithms.
- A description was presented for a backwards verification function in the Superimposed Algorithm (a multiple keyword approximate pattern matching algorithm). If the backwards verification function is used instead of the forward verification function the result can be enriched to report the actual occurrence (start and end position) instead of only the starting position.
- It has been shown that simulating the entire approximate pattern matching automaton (as opposed to removing all states below the first full diagonal) results in a simpler representation invariant without degrading the performance of the algorithm (see Section 2.11.3 on page 40).

- A simplification was found for the diagonal wise bit parallel algorithm. By reordering statements, an algorithm is found which matches better with the concept of computing the active states in an automaton. Refer to section 2.14.2 on page 54 for details.

The process of deriving the formal proofs of the various algorithms has seen several drastic rewrites of proofs. These changes have considerably simplified the derivation.

Initially all algorithms were derived using indices in strings to denote substrings (e.g. a substring starting at position i and ending at position j was denoted by $T[i..j)$). This had several important disadvantages:

- Long formulas looked very complex, this holds especially for those formulas that contained several substrings.
- Initialization of indices and verifying the bounds became quite difficult (this became extra complicated because some algorithms in the literature indexed strings starting at position 0 while others used strings starting at position 1).
- Boundary cases such as the empty string are easily missed.

These index based derivations were rewritten (and sometimes derived from scratch again) using strings and the take and drop calculus. A lot of problems which were encountered using indices vanished automatically. Complex initializations and special cases became almost trivial. The problem with string based derivations was initial unfamiliarity with string calculus. The simplifications brought by the string based approach were certainly worth the extra effort of getting better acquainted with the string calculus.

Another challenge were the derivations and presentation of bit parallel algorithms. After some experiments, a useful form was found, the ideas behind this form are described in appendix B.

The key observation in finding bit parallel algorithms is discovering assignments which are independent can be executed in parallel. Unfortunately, algorithms are often presented using tricks, optimizations and implementation details such that such parallelism becomes obscured.

4.2 Future Work

Although several algorithms have been described, there are still plenty of extensions possible within the field of approximate pattern matching. Possible future includes the following:

- Create a more fine-grained taxonomy. Several steps in the taxonomy are quite large, especially those steps close to the leaves. This can be done by chopping up the larger derivations into more elementary steps and introducing more intermediate algorithms.
- Add missing algorithms to the taxonomy and use gaps in the taxonomy to attempt to derive new algorithms.
- Proof equality of third and fourth rule in definition of the edit distance (see Section 1.2.1).

- Investigate the initial split in the taxonomy. The taxonomy branches between algorithms which gather all occurrences ending in a prefix of the text and algorithms which gather occurrences starting in a prefix of the text, is probably a split between a prefix and suffix based approach (see next point as well).
- Find a new derivation for the ABNDM algorithm. The ABNDM algorithm is the odd man out in the taxonomy presented in this text. I suspect that the ABNDM algorithm could be presented significantly simpler if it is derived as a suffix algorithm (i.e. consider prefixes of suffixes of the text).
- To adjust the presented algorithms for a different distance function. There are plenty of other distance functions available. Some examples are the simpler Hammington distance (allows only character replacements) and the more complex distance functions which allow character swaps and/or substring replacements.
- Classify the matches which are lost by removing the upper right corner of the Approximate Pattern Matching automaton as described in Section 2.13 on page 43.
- Enrich the information which is returned by the algorithms.

  The dynamic programming algorithms could be extended to also maintain the difference between the number of deletions and character insertions used to compute an element of the dynamic programming matrix (at the cost of doubling the memory usage). This difference can be used to efficiently compute the length (and thus the starting position) of an occurrence.

  The backwards verification used in the superimposed algorithm is ideal for finding both the start and the end of an occurrence (currently only the end position is returned).
- Integrate the taxonomy with the taxonomy of the SPARE Time / SPARE Parts toolkit (see [Cle03]). Note that the exact keyword pattern matching problem is a special case of approximate keyword pattern matching (with $k = 0$).
- The SPARE Time toolkit itself should be extended with the algorithms presented here (possibly requiring some restructuring). A Domain Specific Language (DSL) for approximate pattern matching could function as a nice interface for the combined toolkit.
- Benchmark implementations of the algorithms presented and create a theoretical analysis of memory usage and time complexity.
- Research the generalization of the approximate pattern matching algorithms towards Approximate Tree Pattern Matching, Approximate Graph Pattern Matching or Multi-Dimensional Approximate Pattern Matching.

Appendix A

Notations and definitions

Notation 1. *A Hoare triple* $\{\ P\ \}\ S\ \{\ Q\ \}$ *means: execution of statement* S *starting in a state satisfying* P *terminates in a state satisfying* Q*.*

□

Notation 2. *The following notation is used for quantifications:*

$$\langle \otimes x\ :\ P(x)\ :\ Q(x) \rangle$$

where $\otimes$ *is a quantification operator,* x *is a bound dummy variable,* $P(x)$ *is a range predicate on* x *and* $Q(x)$ *is the quantified expression.*

The following table shows the various quantifications used in this text:

Quantification type	*universal*	*existential*	*minimum*	*maximum*	*union*	*set*
symbol	$\forall$	$\exists$	$\downarrow$	$\uparrow$	$\bigcup$	***set***
unit element	*true*	*false*	∞	$-\infty$	$\varnothing$	$\varnothing$

The set quantifier is used to omit the braces around the quantified expression. For example, the set of all natural numbers can be specified as:

$$\langle \bigcup x\ :\ 0 \leq x\ \wedge\ x \in \mathbb{Z}\ :\ \{x\} \rangle$$

Using the set quantification, this is written as:

$$\langle \textbf{\textit{set}}\ x\ :\ 0 \leq x\ \wedge\ x \in \mathbb{Z}\ :\ x \rangle$$

□

Notation 3. *The minimum and maximum operators are denoted by* $\downarrow$ *and* $\uparrow$ *respectively.*

□

Definition 4. *Alphabet* Σ *is a finite (non-empty) set of symbols or characters.*

In practice this is usually the set of ASCII or unicode characters (or a subset). In DNA processing the set A, C, T, G *is used, which is a nice example of a very small alphabet used in practice.*

□

Notation 5. *The empty string is denoted by* ε

□

Definition 6. *The following string operators from the take and drop calculus are used:* $\uparrow, \downarrow, \upharpoonright, \downharpoonright$ *(each has signature:* $\Sigma^* \times \mathbb{N} \mapsto \Sigma^*$*).*

- $\uparrow$ *(left take):* $w \uparrow i$ *are the* $i \downarrow |w|$ *leftmost characters of* w*.*
- $\downarrow$ *(left drop):* $w \downarrow i$ *are the* $(|w| - i) \uparrow 0$ *rightmost characters of* w*.*
- $\upharpoonright$ *(right take):* $w \upharpoonright i$ *are the* $i \downarrow |w|$ *rightmost characters of* w*.*
- $\downharpoonright$ *(right drop):* $w \downharpoonright i$ *are the* $(|w| - i) \uparrow 0$ *leftmost characters of* w*.*

Some examples: $hers \uparrow 3 = her$*,* $hers \downarrow 1 = ers$*,* $hers \upharpoonright 5 = hers$*,* $hers \downharpoonright 10 = \varepsilon$*.*

The four operators can also be defined recursively as follows ($0 < i, x \in \Sigma^*, a \in \Sigma$*):*

$$\begin{array}{lll} x \uparrow 0 & = \varepsilon & \qquad x \upharpoonright 0 & = \varepsilon \\ \varepsilon \uparrow i & = \varepsilon & \qquad \varepsilon \upharpoonright i & = \varepsilon \\ ax \uparrow i & = a(x \uparrow (i-1)) & \qquad xa \upharpoonright i & = (x \upharpoonright (i-1))a \end{array}$$

$$\begin{array}{lll} x \downarrow 0 & = x & \qquad x \downharpoonright 0 & = x \\ \varepsilon \downarrow i & = \varepsilon & \qquad \varepsilon \downharpoonright i & = \varepsilon \\ ax \downarrow i & = x \downarrow (i-1) & \qquad xa \downharpoonright i & = x \downharpoonright (i-1) \end{array}$$

□

Notation 7. *For strings* w, x *and character* a*, the notation* $w :: ax$ *is used as a condition which return whether or not* w *is of the form* ax *(i.e. a character followed by a string). As a side effect, this will also assign* $w[0]$ *to* a *and* $w \downarrow 1$ *to* x*.*

For example, the following two selections are equivalent:

if $w :: ax \rightarrow S_0$
$[\!]$ $\neg(w :: ax) \rightarrow S_1$
fi;

if $w \neq \varepsilon \rightarrow a, x := w[0], w \downarrow 1;$
S_0
$[\!]$ $w = \varepsilon \rightarrow S_1$
fi;

□

Definition 8. *The reverse of a string* w *is denoted by* w^R*. Reverse is defined recursively by:*

$$\begin{array}{ll} \varepsilon^R & = \varepsilon \\ (wa)^R & = a(w^R) \end{array}$$

The reverse operator has the obvious property $|w| = |w^R|$ *which can easily be proven using structural induction on* w*.*

□

Definition 9. *Given alphabet* Σ *the function* $Pref : \Sigma^* \mapsto \mathcal{P}(\Sigma^*)$ *and* $Suff : \Sigma^* \mapsto \mathcal{P}(\Sigma^*)$ *are defined as follows:*

$$\begin{array}{ll} Pref(w) & = \langle \textbf{\textit{set}}\ x, y\ :\ xy = w\ :\ x \rangle \\ Suff(w) & = \langle \textbf{\textit{set}}\ x, y\ :\ xy = w\ :\ y \rangle \end{array}$$

Informally, $Pref(w)$ *(*$Suff(w)$*) is the set of all strings which are prefixes (suffixes) of* w*.*

□

Definition 10. *Partial orders* $\leq_p, <_p, \leq_s, <_s$ *over* $\Sigma^* \times \Sigma^*$ *are defined as:*

$$\begin{array}{ll} v \leq_p w & \equiv v \in Pref(w) \\ v <_p w & \equiv v \in Pref(w) \backslash \{w\} \\ v \leq_s w & \equiv v \in Suff(w) \\ v <_s w & \equiv v \in Suff(w) \backslash \{w\} \end{array}$$

□

Definition 11. *A NFA is a five tuple* $\langle Q, \Sigma, \delta, q_0, F \rangle$ *where:*

- Q *is the set of states*
- Σ *is the alphabet*
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ *is the transition function*
- q_0 *is the initial state*
- F *is the set of final states*

A state of an NFA is called active after processing a string s *if it can be reached from the initial state using only transitions on all of the subsequent characters in the input* s *and* ε *transitions.*

□

Definition 12. *The* ε*-closure of a state is the set of all states reachable from that state by following zero or more* ε*-transitions.*

□

Definition 13. *The extended transition function* δ^* *is defined as follows (with* $q \in Q, w \in \Sigma^*$ *and* $a \in \Sigma$*):*

$$\begin{array}{rcl} \delta^*(q, \varepsilon) & = & \varepsilon\text{-}closure(q) \\ \delta^*(q, wa) & = & \langle \bigcup r, s\ :\ r \in \delta^*(q, w)\ \wedge\ s \in \delta(r, a)\ :\ \varepsilon\text{-}closure(s) \rangle \end{array}$$

□

Appendix B

Deriving bit parallel algorithms

This section provides some definitions and guidelines for deriving bit parallel algorithms. A bit parallel algorithm utilizes machine words to speed up computation by storing several variables in a single machine word and computing the next values by updating the machine word as a whole. This effectively computes the values of all variables in a machine word in parallel, hence the name bit parallel algorithms. The fact that these variables are computed in parallel implies that there cannot exist any dependency between these variables new values (it is allowed that the variables depend on earlier computed values, such as the value of the variables in a previous iteration).

For example, a machine word can be used to represent an entire boolean array. If all assignments to elements of the boolean array can be replaced by a bit parallel expression then a linear time operation can be replaced by a constant time operation (provided the array does not exceed the length of a machine word).

How can one derive a bit parallel algorithm? There are three essential steps:

- Identify a part of program state which is independent such that it can be computed in parallel.
- Derive an algorithm storing the program state which is to be computed in parallel in an array.
- Derive an expression to update the array as a whole. This expression must be a function in terms of the array(s) and other program variables, but may not contain individual elements of the array as parameters. If the function is to be implemented efficiently, it should be a composition of basic machine word operations only (see below).

These steps do not necessarily occur in this order. It possible to derive an algorithm (step two) and transform it to a bit parallel implementation afterwards, though this is likely to be harder compared with a direct approach.

The basic machine word operations used in this text are the following (with $0 \leq h$):

$LShiftT : \mathbb{B}^{h+1} \rightarrow \mathbb{B}^{h+1}$
$LShiftT((a_0, \ldots, a_h)) = (a_1, \ldots, a_h, true)$
$RShiftT : \mathbb{B}^{h+1} \rightarrow \mathbb{B}^{h+1}$

$RShiftT((a_0, \ldots, a_h)) = (true, a_0, \ldots, a_{h-1})$
$LShiftF : \mathbb{B}^{h+1} \to \mathbb{B}^{h+1}$
$LShiftF((a_0, \ldots, a_h)) = (a_1, \ldots, a_h, false)$
$RShiftF : \mathbb{B}^{h+1} \to \mathbb{B}^{h+1}$
$RShiftF((a_0, \ldots, a_h)) = (false, a_0, \ldots, a_{h-1})$
$And : \mathbb{B}^{h+1} \times \mathbb{B}^{h+1} \to \mathbb{B}^{h+1}$
$And((a_0, \ldots, a_h), (b_0, \ldots, b_h)) = (a_0 \wedge b_0,\ \ldots,\ a_h \wedge b_h)$
$Or : \mathbb{B}^{h+1} \times \mathbb{B}^{h+1} \to \mathbb{B}^{h+1}$
$Or((a_0, \ldots, a_h), (b_0, \ldots, b_h)) = (a_0 \vee b_0,\ \ldots,\ a_h \vee b_h)$
$Xor : \mathbb{B}^{h+1} \times \mathbb{B}^{h+1} \to \mathbb{B}^{h+1}$
$Xor((a_0, \ldots, a_h), (b_0, \ldots, b_h)) = (a_0\ xor\ b_0,\ \ldots,\ a_h\ xor\ b_h)$

For a really rich set of basic machine word operations see [HSW02]. This book also describes lots of functions which can be constructed with the basic operations.

The third item above states that a bit parallel expression must be derived. The derivation in the 2^{nd} step should specify the values which must be assigned to each of the elements of the array.

To compute the bit parallel expression follow the next steps. First expand the expressions for all elements into a vector. Then rewrite this vector using the following steps:

- make elements of a similar shape
- align indices
- apply bit parallel functions.

For example: Suppose there are three arrays (A, B and C) each have two elements and have a boolean domain. The derivation has shown that $C[0]$ should equal $false$ and $C[1]$ should equal $A[0] \wedge B[1]$.

$$
\begin{aligned}
& C \\
\equiv \quad & \{ \text{ expand } \} \\
& (C[0], C[1]) \\
\equiv \quad & \{ \text{ specification (this is the first step) } \} \\
& (false, A[0] \wedge B[1]) \\
\equiv \quad & \{ \ \wedge \ \text{(introduce a similar shape) } \} \\
& (false \wedge B[0], A[0] \wedge B[1]) \\
\equiv \quad & \{ \ And \ \text{(apply function) } \} \\
& And(\ (false, A[0])\ ,\ (B[0], B[1])\ ) \\
\equiv \quad & \{ \ RShiftF \ \text{(align indices) } \} \\
& And(RShiftF((A[0], A[1])), (B[0], B[1])) \\
\equiv \quad & \{ \text{ simplify } \} \\
& And(RShiftF(A), B)
\end{aligned}
$$

The above derivation shows all common operations applied in a single derivation. Consider the step which introduces a similar shape. The choice to introduce $B[0]$ as a conjunct might not be completely trivial at this point, however there is not really any choice. Element $B[1]$ is already on the correct position (it occurs in an expression with index 1 which equals its element number). The added conjunct can equal either $true$ or $false$ which does not matter $((false \wedge x) \equiv x)$. By choosing $B[0]$, vector B does not need to be modified.

The operation above also shows two new hints: expand and simplify. The expand hint indicates a replacement of the form C by $(C[0], C[1])$ and the simplify hint indicates a replacement the other way around. Hence, expand adds indices, simplify removes indices.

All the steps above occur frequently in the bit parallel derivations in this text.

Bibliography

[Cle03] L.G.W.A. Cleophas. Towards spare time a new taxonomy and toolkit of keyword pattern matching algorithms. Master's thesis, Technical University Eindhoven (TU/e), 2003.

[Hol96] Jan Holub. Reduced nondeterministic finite automata for approximate string matching. *Proceedings of the Prague Stringology Club Workshop*, pages 19–27, 1996.

[HSW02] Jr. Henry S. Warren. *Hacker's Delight*. Addison Wesley, 2002.

[NR02] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings: Practical on-line search algorithms for texts and biological sequences*. Cambridge university press, 2002.

[Ukk85] Esko Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1-3):132–137, 1985.

[Wat95] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Technical University Eindhoven (TU/e), 1995.