A new family of string pattern matching algorithms

Bruce W. Watson & Richard E. Watson
Ribbit Software Systems Inc.
(IST Technologies Research Group)
Box 24040, 297 Bernard Ave.
Kelowna, B.C., V1Y 9P9, Canada
E-mail: {watson, rwatson}@RibbitSoft.com
Web: www.RibbitSoft.com
Phone: +1-250-762-6591
Fax: +1-250-762-6511

Abstract

Even though the field of pattern matching has been well studied, there are still many interesting algorithms to be discovered. In this paper, we present a new family of single keyword pattern matching algorithms. We begin by deriving a common ancestor algorithm, which naïvely solves the problem. Through a series of correctness preserving predicate strengthenings, and implementation choices, we derive efficient variants of this algorithm. This paper also presents one of the first algorithms which could be used to do a minimal number of match attempts within the input string (by maintaining as much information as possible from each match attempt).

1 Introduction and related work

In this paper, we present a new family of algorithms solving the single keyword string pattern matching problem. This particular pattern matching problem can be described as follows: given an input string S and a keyword p, find all occurrences of p as a continuous substring of S. The field of string pattern matching is generally well-studied, however, it continues to yield new and exciting algorithms, as was seen in Watson's Ph.D. dissertation [8], the recent book by Crochemore and Rytter [2], the more classic paper by Hume and Sunday [7] and book by Gonnet and Baeza-Yates [4]. In the dissertation [8], a taxonomy of existing algorithms was presented, along with a number of new algorithms. Any given algorithm may have more than one possible derivation, leading to different classifications of the algorithm in a taxonomy[1]. Many of the new derivations can prove to be more than just an educational curiosity, possibly leading to interesting new families of algorithms. This paper presents one such family — with some new algorithms and also some alternative derivations of existing ones. While a few of the derivation steps are shared with the presentation in [8], this paper takes a substantially different approach overall and arrives at some completely new algorithms.

The algorithms presented in this paper can be extended to handle some more complex pattern matching problems, including multiple keyword pattern matching, regular pattern matching and multi-dimensional pattern matching.

Our derivation begins with a description of the problem, followed by a naïve first algorithm. We then make incremental (correctness preserving) improvements to these algorithms, eventually yielding efficient variants. Throughout the paper, we first precede each definition with some

[1] This is precisely what happened with the Boyer-Moore type algorithms as presented in the dissertation [8].

intuitive background. The mathematical preliminaries necessary to read this paper can be found in Appendix A.

2 The problem and a first algorithm

Before giving the problem specification (in the form of a postcondition to the algorithms), we define a predicate which will make the postcondition and algorithms easier to read. Keyword p (with the restriction that $p \neq \varepsilon$, where ε is the empty string) is said to *match* at position j in input string S if $p = S_{j \cdots j+|p|-1}$; this is restated in the following predicate:

Definition 2.1 (Predicate *Matches***):** We define predicate *Matches* as

$$Matches(S, p, j) \equiv p = S_{j \cdots j+|p|-1}$$

□

The pattern matching problem requires us to compute the set of all matches of keyword p in input string S. We register the matches as the set O of all indices j (in S) such that $Matches(S, p, j)$ holds.

Definition 2.2 (Single keyword pattern matching problem): Given a common alphabet V, input string S, and pattern keyword p, the problem is defined using postcondition PM:

$$O = \{\, j \mid j \in [0, |S|) \land Matches(S, p, j) \,\}$$

Note that this postcondition implicitly depends upon S and p. □

We can now present a nondeterministic algorithm which keeps track of the set of possible indices (in S) at which a match might still be found (indices at which we have not yet checked for a match). This set is known as the *live zone*. Those indices not in the live zone are said to be in the *dead zone*. This give us our first algorithm (presented in the guarded command language of Dijkstra [3, 1]).

Algorithm 2.3:

$live, dead := [0, |S|), \emptyset;$
$O := \emptyset;$
$\{$ invariant: $live \cup dead = [0, |S|) \land live \cap dead = \emptyset$
$\quad\quad \land\ O = \{\, j \mid j \in dead \land Matches(S, p, j) \,\} \ \}$
do $live \neq \emptyset \rightarrow$
  let $j : j \in live;$
  $live, dead := live \setminus \{j\}, dead \cup \{j\};$
  if $Matches(S, p, j) \rightarrow O := O \cup \{j\}$
  $[\!]\ \neg Matches(S, p, j) \rightarrow$ **skip**
  fi
od$\{$ postcondition: PM $\}$

□

The invariant specifies that *live* and *dead* are disjoint and account for all indices in S; additionally, any match at an element of *dead* has already been registered. Thanks to this relationship between *live* and *dead*, we could have written the repetition condition $live \neq \emptyset$ as $dead \neq [0, |S|)$, and the j selection condition $j \in live$ as $j \notin dead$. It should be easy to see that the invariant and the termination condition of the repetition implies the postcondition — yielding a correct algorithm. Note that this algorithm is highly over-specified by keeping both variables *live* and *dead* to represent the live and dead zones, respectively. For efficiency, only one of these sets would normally be kept.

Some of the rightmost positions in S cannot possibly accommodate matches — no match can be found at any point $j \in [|S| - |p| + 1, |S|)$ since $|S_{j \cdots |S|-1}| \leq |S_{|S|-|p|+1 \cdots |S|-1}| < |p|$ (the match attempt begins too close to the end of S for p to fit). For this reason, we safely change the initializations of *live* and *dead* to

$$live, dead := [0, |S| - |p|], [|S| - |p| + 1, |S|)$$

In the next section, give a deterministic (more realistically implemented) version of the last algorithm.

3 A more deterministic algorithm

In the last algorithm, our comparison of p with $S_{j \cdots j+|p|-1}$ is embedded within the evaluation of predicate *Matches*. In this section, we make this comparison explicit. We begin by noting that $p = S_{j \cdots j+|p|-1}$ is equivalent to comparing the individual symbols p_k of p with the corresponding symbols S_{j+k} of S (for $k \in [0, |p|)$). In fact, we can consider the symbols in any order whatsoever. To determine the order in which they will be considered, we introduce *match orders*:

Definition 3.1 (Match order): We define a match order mo as a permutation on $[0, |p|)$. □

Using mo, we can restate our match predicate.

Property 3.2 (Predicate *Matches*): Predicate *Matches* is restated as

$$Matches(S, p, j) \equiv (\forall\ i : i \in [0, |p|) : p_{mo(i)} = S_{j+mo(i)})$$

□

This rendition of the predicate will be evaluated by a repetition which uses a new integer variable i to step from 0 to $|p| - 1$, comparing $p_{mo(i)}$ to the corresponding symbol of S. As i increases, the repetition has the following invariant:

$$(\forall\ k : k \in [0, i) : p_{mo(k)} = S_{j+mo(k)})$$

and terminates as early as possible.

In the following algorithm, we use the match order mo, the new repetition and our previous optimization to the initializations of *dead* and *live*.

Algorithm 3.3:

$live, dead := [0, |S| - |p|], [|S| - |p| + 1, |S|);$
$O := \emptyset;$
$\{$ invariant: $live \cup dead = [0, |S|) \wedge live \cap dead = \emptyset$
$\qquad\qquad \wedge\ O = \{ j \mid j \in dead \wedge Matches(S, p, j) \}\ \}$
do $live \neq \emptyset \rightarrow$
  let $j : j \in live;$
  $live, dead := live \setminus \{j\}, dead \cup \{j\};$
  $i := 0;$
  $\{$ invariant: $(\forall\ k : k \in [0, i) : p_{mo(k)} = S_{j+mo(k)})\ \}$
  do $i < |p|$ **cand** $p_{mo(i)} = S_{j+mo(i)} \rightarrow$
    $i := i + 1$
  od;
  $\{$ postcondition: $(\forall\ k : k \in [0, i) : p_{mo(k)} = S_{j+mo(k)})$
  $\qquad\qquad \wedge\ (i < |p| \Rightarrow p_{mo(i)} \neq S_{j+mo(i)})\ \}$
  if $i = |p| \rightarrow O := O \cup \{j\}$
  $[\!]$ $i < |p| \rightarrow$ **skip**
  fi
od$\{$ postcondition: PM $\}$

□

The operator P **cand** Q appears in the guard of the inner loop of the above algorithm. This operator is similar to conjunction $P \wedge Q$ except that if the first conjunct evaluates to *false* then the second conjunct is not even evaluated. This proves to be a useful property in cases such as the loop guard since, if the first conjunct ($i < |p|$) is *false* (hence $i >= |p|$, and indeed $i = |p|$), then the term $mo(i)$ appearing in the second conjunct is not even defined. Note that the implication within the second conjunct of the loop postcondition is derived from the loop guard, forcing the implication operator to be conditional as well (that is, if $i < |p|$ is determined to be *false*, then $p_{mo(i)} \neq S_{j+mo(i)}$ is not even evaluated).

As we will see in the next section, the particular choice for mo can make a difference in the performance of the algorithm. Some possible match orders include 'forward' (mo is the identity permutation) and 'reverse' ($mo(i) = |p| - i - 1$). The permutation chosen could even be devised according to some theoretical expectations or statistical analysis for a particular application. For instance, if p contained a subsequence of characters which are known to appear very rarely within the type of input string, then the permutation would be chosen in order to check for a match within that subsequence first (since this may result in discovering a mismatch sooner). This approach is standard fare, and is used to find fast variants of the Boyer-Moore algorithms (as described in [7]).

Yet another possibility which could prove interesting is that mo is chosen on-the-fly, that is, $mo(i)$ could be allowed to depend upon $mo(i-1)$, $mo(i-2)$, $\ldots$, $mo(0)$ and even upon other factors such as how much of the input string we have already processed. Such an choice of permutation would be highly specialized to a particular instance of this problem and we do not explore it any further in this paper. In the next section, we outline some precomputation on p which speeds up the algorithm tremendously but also depends upon the choice of mo, meaning that if we devised the permutation on-the-fly, we would be forced to perform the precomputation for each of the possible unique permutations that our algorithm could produce (a maximum of $|p|!$).

4 Reusing match information

On each iteration of the outer repetition, index j is chosen and eliminated from the live zone in the statement:

$$live, dead := live \setminus \{j\}, dead \cup \{j\}$$

The performance of the algorithm can be improved if we remove more than just j in some of the iterations. To do this, we can use some of the match information, such as i, which indicates how far through mo the match attempt proceeded before finding a mismatching symbol. The information most readily available is the postcondition of the inner repetition:

$$(\forall\ k : k \in [0, i) : p_{mo(k)} = S_{j+mo(k)}) \wedge (i < |p| \Rightarrow p_{mo(i)} \neq S_{j+mo(i)})$$

We denote this postcondition by $Result(S, p, i, j)$. Since this postcondition holds, we may be able to deduce that certain indices in S cannot possibly be the site of a match. It is such indices which we could also remove from the live zone. They are formally characterized as:

$$\{\, x \mid x \in [0, |S|) \wedge (Result(S, p, i, j) \Rightarrow \neg Matches(S, p, x)) \,\}$$

Determining this set at pattern matching time is inefficient and not easily implemented. We wish to derive a safe approximation of this set which can be precomputed, tabulated and indexed (at pattern matching time) by i. In order to precompute it, the approximation must be independent of j and S. We wish to find a strengthening of the range predicate since this will allow us to still remove a safe set of elements from set *live*, thanks to the property that, if $P \Rightarrow Q$ (P is a *strengthening* of Q, and Q is a *weakening* of P), then

$$\{\, x \mid P(x) \,\} \subseteq \{\, x \mid Q(x) \,\}$$

As a first step towards this approximation, we can normalize the ideal set (above), by subtracting j from each element. The resulting characterization will be more useful for precomputation reasons:

$$\{\, x \mid x \in [-j, |S| - j) \wedge (\mathit{Result}(S,p,i,j) \Rightarrow \neg \mathit{Matches}(S,p,j+x)) \,\}$$

Note that this still depends upon j, however, it will make some of the derivation steps shown in Appendix B easier. Condensed, the derivation appears as:

$$\begin{array}{ll}
 & (\mathit{Result}(S,p,i,j) \Rightarrow \neg \mathit{Matches}(S,p,j+x)) \\
\Leftarrow & \qquad \{\, \text{Appendix B} \,\} \\
 & \neg((\forall\ k : k \in [0,i) \wedge mo(k) \in [x, |p| + x) : p_{mo(k)} = p_{mo(k)-x}) \\
 & \wedge\ (i < |p| \wedge mo(i) \in [x, |p| + x) \Rightarrow p_{mo(i)} \neq p_{mo(i)-x})) \\
\equiv & \qquad \{\, \text{define the predicate } \mathit{Approximation}(p,i,x) \,\} \\
 & \mathit{Approximation}(p,i,x)
\end{array}$$

Note that we define the predicate $\mathit{Approximation}(p,i,x)$ which depends only on p and i and hence can be precomputed and tabulated. It should be mentioned that this is one of several possible useful strengthenings which could be derived. We could even have used the strongest predicate, *false*, instead of $\mathit{Approximation}(p,i,x)$. This would yield the empty set, $\emptyset$, to be removed form *live* in addition to j (as in the previous algorithm).

We can derive a smaller range predicate of x for which we have to check if $\mathit{Approximation}(p,i,x)$ holds. Notice that choosing and x such that $[x, |p| + x) \cap [0, |p|) = \emptyset$ has two important consequences:

- The range of the quantification in first conjunct of $\mathit{Approximation}(p,i,x)$ is empty (hence this conjunct is *true*, by the definition of universal quantification with an empty range).
- The range condition of the second conjunct (the 'implicator') is *false* — hence the whole of the second conjunct is *true* since $\mathit{false} \Rightarrow P$ for all predicates P.

With this choice of x, we see that predicate $\mathit{Approximation}(p,i,x)$ always evaluates to *false*, in which case we need not even consider values of x such that $[x, |p| + x) \cap [0, |p|) = \emptyset$. This simplification can be seen in the following algorithm where we have solved the above range equation for x, yielding the restriction that $x \in [1 - |p|, |p| - 1)$. Intuitively we know that there must be such a range restriction since we can not possibly know from a current match attempt whether or not we will find a match of p in S more than $|p|$ symbols away.

Finally we have the following algorithm (in which we have added the additional update of *live* and *dead* below the inner repetition). Note that we introduce the set *nogood* to accumulate the indices for which $\mathit{Approximation}(p,i,x)$ holds. Also note that we renormalize the set *nogood* by adding j to each of its members and ensuring that it is within the valid range of indices, $[0, |S|)$.

Algorithm 4.1:

$$\begin{array}{l}
\mathit{live}, \mathit{dead} := [0, |S| - |p|], [|S| - |p| + 1, |S|); \\
O := \emptyset; \\
\{\ \text{invariant: } \mathit{live} \cup \mathit{dead} = [0, |S|) \wedge \mathit{live} \cap \mathit{dead} = \emptyset \\
\qquad\qquad\quad \wedge\ O = \{\, l \mid l \in \mathit{dead} \wedge \mathit{Matches}(S,p,l) \,\}\ \} \\
\mathbf{do}\ \mathit{live} \neq \emptyset \rightarrow \\
\quad \mathbf{let}\ j : j \in \mathit{live}; \\
\quad \mathit{live}, \mathit{dead} := \mathit{live} \setminus \{j\}, \mathit{dead} \cup \{j\}; \\
\quad i := 0; \\
\quad \{\ \text{invariant: } (\forall\ k : k \in [0,i) : p_{mo(k)} = S_{j+mo(k)})\ \} \\
\quad \mathbf{do}\ i < |p|\ \mathbf{cand}\ p_{mo(i)} = S_{j+mo(i)} \rightarrow \\
\qquad i := i + 1 \\
\quad \mathbf{od}; \\
\quad \{\ \text{postcondition: } \mathit{Result}(S,p,i,j)\ \}
\end{array}$$

$$
\begin{array}{l}
\quad \textbf{if } i = |p| \rightarrow O := O \cup \{j\} \\
\quad [\!] \;\; i < |p| \rightarrow \textbf{skip} \\
\quad \textbf{fi}; \\
\quad \mathit{nogood} := (\{\, x \mid x \in [1-|p|, |p|-1) \wedge \mathit{Approximation}(p,i,x) \,\} + j) \cap [0,|S|); \\
\quad \mathit{live} := \mathit{live} \setminus \mathit{nogood}; \\
\quad \mathit{dead} := \mathit{dead} \cup \mathit{nogood} \\
\textbf{od}\{ \text{ postcondition: } \mathit{PM} \;\}
\end{array}
$$

□

5 Choosing j from the live zone

In this section, we discuss strategies for choosing the index j (from the live zone) at which to make a match attempt. In the last algorithm, the way in which j is chosen from set *live* is nondeterministic. This leads to the situation that *live* (and, of course, *dead*) is fragmented, meaning that an implementation of the algorithm would have to maintain a set of indices for live. If we can ensure that *live* is contiguous, then an implementation would only need to keep track of the (one or two) boundary points between *live* and *dead*. There are several ways to do this, and we discuss some of them in the following subsections section. Each of these represents a particular *policy* to be used in the selection of j.

5.1 Minimal element

We could use the policy of always taking the minimal element of *live*. In that case, we can make some simplifications to the algorithm (which, in turn, improve the algorithm's performance):

- We need only store the minimal element of *live*, instead of sets *live* and *dead*. We use $\widehat{\mathit{live}}$ to denote the minimal element.
- The dead zone update could be modified as follows: we will have considered all of the positions to the left of j and so we can ignore the negative elements of the update set:

  $$\{\, x \mid x \in [1-|p|, 0) \wedge \mathit{Approximation}(p,i,x) \,\}$$

  Indeed, we can just add the maximal element (which is still contiguously in the update set and greater than j) of the update set to $\widehat{\mathit{live}}$ for the new version of our new update of *live* and *dead*.

Depending upon the choice of weakening, and the choice of match order, the above policy yields variants of the classical Boyer-Moore algorithm (see [8, 2, 7]):

Algorithm 5.1:

$$
\begin{array}{l}
\widehat{\mathit{live}} := 0; \\
O := \emptyset; \\
\textbf{do } \widehat{\mathit{live}} \leq |S| - |p| \rightarrow \\
\quad j := \widehat{\mathit{live}}; \\
\quad \widehat{\mathit{live}} := \widehat{\mathit{live}} + 1; \\
\quad i := 0; \\
\quad \{ \text{ invariant: } (\forall\, k : k \in [0,i) : p_{mo(k)} = S_{j+mo(k)}) \;\} \\
\quad \textbf{do } i < |p| \textbf{ cand } p_{mo(i)} = S_{j+mo(i)} \rightarrow \\
\qquad i := i + 1 \\
\quad \textbf{od}; \\
\quad \{ \text{ postcondition: } \mathit{Result}(S,p,i,j) \;\}
\end{array}
$$

$\quad$ **if** $i = |p| \to O := O \cup \{j\}$
$\quad$ $\| \ \ i < |p| \to$ **skip**
$\quad$ **fi**;
$\quad$ $\widehat{nogood} := (\mathbf{MAX}\ x : x \in [0, |p| - 1) \wedge (\forall\ h : h \in [0, x] : Approximation(p, i, x)) : x)$;
$\quad$ $\widehat{live} := \widehat{live} + \widehat{nogood}$
od{ postcondition: PM }

□

5.2 Maximal element

We could always choose the maximal element of *live*. This would yield the dual of the previous algorithm.

5.3 Randomization

We could randomize the choice of j. Given the computational cost of most reasonable quality pseudo-random number generators, it is not clear yet that this would yield an interesting or efficient algorithm. It is conceivable that there exist instances of the problem which could benefit from randomly selected match attempts.

5.4 Recursion

We could also devise a recursive version of the algorithm as a procedure. This procedure receives a contiguous range of live indices (*live*) — initially consisting of the range $[0, |S| - |p|]$.

If the set it receives is empty, the procedure immediately returns. If the set is non-empty, j is chosen so that the resulting dead zone would appear reasonably close to the middle of the current live zone[2]. This ensures that we discard as little information as possible from the *nogood* index set. After the match attempt, the procedure recursively invokes itself twice, with the two reduced live zones on either side of the new dead zone. This yields the following procedure:

Algorithm 5.2:

---

proc $mat(S, p, live, dead) \to$
$\quad$ { $live$ is contiguous }
$\quad$ **if** $live = \emptyset \to$ **skip**
$\quad$ $\| \ \ live \neq \emptyset \to$
$\quad\quad$ $live\_low := (\mathbf{MIN}\ k : k \in live : k)$;
$\quad\quad$ $live\_high := (\mathbf{MAX}\ k : k \in live : k)$;
$\quad\quad$ $j := \lfloor (live\_low + live\_high - |p|)/2 \rfloor$;
$\quad\quad$ $i := 0$;
$\quad\quad$ { invariant: $(\forall\ k : k \in [0, i) : p_{mo(k)} = S_{j+mo(k)})$ }
$\quad\quad$ **do** $i < |p|$ **cand** $p_{mo(i)} = S_{j+mo(i)} \to$
$\quad\quad\quad$ $i := i + 1$
$\quad\quad$ **od**;
$\quad\quad$ { postcondition: $Result(S, p, i, j)$ }
$\quad\quad$ **if** $i = |p| \to O := O \cup \{j\}$
$\quad\quad$ $\| \ \ i < |p| \to$ **skip**
$\quad\quad$ **fi**;
$\quad\quad$ $new\_dead := (\{\ x \mid x \in [1 - |p|, |p| - 1) \wedge Approximation(p, i, x)\ \} + j) \cap [0, |S|)$;
$\quad\quad$ $dead := dead \cup new\_dead$;
$\quad\quad$ $mat(S, p, [live\_low, (\mathbf{MIN}\ k : k \in new\_dead : k)), dead)$;

[2] The algorithm given in this section makes a simple approximation by taking the middle of the live zone it receives, and subtracting $\lfloor |p|/2 \rfloor$.

$mat(S, p, ((\mathbf{MAX}\ k : k \in new\_dead : k), live\_high], dead)$

fi

corp

□

This procedure is used in the algorithm:

Algorithm 5.3:

$O := \emptyset$;
$mat(S, p, [0, |S| - |p|], [|S| - |p| + 1, |S|))$
{ postcondition: PM }

□

Naturally, for efficiency reasons, the set *live* can be represented by its minimal and maximal elements (since it is contiguous).

6 Further work

The family of algorithms presented in this paper can easily be extended to multiple pattern matching and to regular pattern matching (using regular expressions or regular grammars). In each of these cases, various strengthenings of the update predicate could be explored and specialized methods for choosing the index of the next match attempt determined.

Another branch in this family tree of algorithms could be derived by removing the conjunct $p_{mo(i)} = S_{j+mo(i)}$ from the guard of the inner repetition (that is, do not terminate the match attempt as soon as we encounter a mismatch). This would allow us to accumulate more mismatch information and possibly provide a weaker strengthening than $Approximation(p, i, x)$ (and hence a larger set *nogood*). It is not yet clear that this would lead to an interesting family of algorithms.

7 Conclusions

We have shown that there are still many interesting algorithms to be derived within the field of single keyword pattern matching. The correctness preserving derivation of am entirely new family of such algorithms demonstrates the use of formal methods and the use of predicates, invariants, postconditions and preconditions. It is unlikely that such a family of algorithms could have be devised without the use of formal methods.

Historically, keyword pattern matching algorithms have restricted themselves to processing the input string from left to right, thus discarding half of the useful information which can be determined from previous match attempts. As a new starting point for pattern matching algorithms, this paper proposes pattern matching in the more general manner of making match attempts in a less restricting order within the input string. With the advent of both hardware and software which enable near-constant-time lookup of a random character in a file stream (using memory mapped files, as are available in most newer operating systems), such algorithms will prove useful for typical single keyword pattern matching applications (ones which have a finite input string which can be randomly accessed).

The derivation also yielded a recursive algorithm which appears to be particularly efficient. The algorithm has been implemented, and benchmarking results will be presented in the final paper, comparing the algorithm to the other extensively benchmarked algorithms in [7, 8].

8 Acknowledgements

We would like to thank Nanette Saes and Mervin Watson for proofreading this paper, Ricardo Baeza-Yates for serving as a sounding board, and Ribbit Software Systems Inc. for allowing us to pursue some of our pure research interests.

References

[1] COHEN, E. *Programming in the 1990s,* (Springer-Verlag, New York, NY, 1990).

[2] CROCHEMORE, M. and W. RYTTER. *Text Algorithms,* (Oxford University Press, Oxford, England, 1994).

[3] DIJKSTRA, E.W. *A discipline of programming,* (Prentice Hall, Englewood Cliffs, NJ, 1976).

[4] GONNET, G.H. and R. BAEZA-YATES. *Handbook of Algorithms and Data Structures (In Pascal and C),* (Addison-Wesley, Reading, MA, 2nd edition, 1991).

[5] GRIES, D. and F.B. SCHNEIDER. *A Logical Approach to Discrete Math,* (Springer-Verlag, New York, NT, 1993).

[6] GRIES, D. *The Science of Programming,* (Springer-Verlag, New York, NY, 1981).

[7] HUME, S.C. and D. SUNDAY. "Fast string searching," *Software — Practice & Experience,* **21**(11) 1221–1248.

[8] WATSON, B.W. *Taxonomies and Toolkits of Regular Language Algorithms,* Ph.D dissertation, Faculty of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, September 1995, ISBN 90-386-0396-7.

A Mathematical preliminaries

While most of the mathematical notation and definitions used in this paper are is described in detail in [5], in this appendix we present some more specific notations. Indexing within strings begins at 0, as in the C and C++ programming languages. We use ranges of integers throughout the paper which are defined by (for integers i and j):

$$[i, j) = k | i \leq k < j$$

$$(i, j] = k | i < k \leq j$$

$$[i, j] = [i, j) \cup (i, j]$$

$$(i, j) = [i, j) \cap (i, j]$$

In addition, we define a *permutation* of a set of integers to be a bijective mapping of those integers onto themselves.

B Range predicate strengthening

Here, we present the derivation of a strengthening of the range predicate

$$\mathit{Result}(S, p, i, j) \Rightarrow \neg \mathit{Matches}(S, p, j + x)$$

Being more comfortable with weakening steps, we begin with the negation of part of the above range predicate, and proceed by weakening:

$$
\begin{array}{ll}
& \neg(\mathit{Result}(S,p,i,j) \Rightarrow \neg \mathit{Matches}(S,p,j+x)) \\
\equiv & \qquad \{\text{ definition of } \Rightarrow \} \\
& \neg(\neg \mathit{Result}(S,p,i,j) \vee \neg \mathit{Matches}(S,p,j+x)) \\
\equiv & \qquad \{\text{ De Morgan } \} \\
& \mathit{Result}(S,p,i,j) \wedge \mathit{Matches}(S,p,j+x)) \\
\equiv & \qquad \{\text{ definition of } \mathit{Result} \text{ and } \mathit{Matches} \} \\
& (\forall\ k : k \in [0,i) : p_{mo(k)} = S_{j+mo(k)}) \wedge (i < |p| \Rightarrow p_{mo(i)} \neq S_{j+mo(i)}) \\
& \ \wedge (\forall\ k : k \in [0,|p|) : p_{mo(k)} = S_{mo(k)+j+x}) \\
\equiv & \qquad \{\text{ change range predicate in second quantification and definition of } mo \} \\
& (\forall\ k : k \in [0,i) : p_{mo(k)} = S_{j+mo(k)}) \wedge (i < |p| \Rightarrow p_{mo(i)} \neq S_{j+mo(i)}) \\
& \ \wedge (\forall\ k : mo(k) \in [0,|p|) : p_{mo(k)} = S_{mo(k)+j+x}) \\
\Rightarrow & \qquad \{\text{ change dummy } (mo(k') = mo(k)+x) \text{, restrict range } \} \\
& (\forall\ k : k \in [0,i) : p_{mo(k)} = S_{j+mo(k)}) \wedge (i < |p| \Rightarrow p_{mo(i)} \neq S_{j+mo(i)}) \\
& \ \wedge (\forall\ k' : mo(k') - x \in [0,|p|) : p_{mo(k')-x} = S_{mo(k')+j}) \\
\equiv & \qquad \{\text{ simplify range predicate of second quantification } \} \\
& (\forall\ k : k \in [0,i) : p_{mo(k)} = S_{j+mo(k)}) \wedge (i < |p| \Rightarrow p_{mo(i)} \neq S_{j+mo(i)}) \\
& \ \wedge (\forall\ k' : mo(k') \in [x,|p|+x) : p_{mo(k')-x} = S_{mo(k')+j}) \\
\Rightarrow & \qquad \{\text{ one-point rule: second conjunct and second quantification } \} \\
& (\forall\ k : k \in [0,i) : p_{mo(k)} = S_{j+mo(k)}) \\
& \ \wedge ((i < |p| \wedge mo(i) \in [x,|p|+x)) \Rightarrow p_{mo(i)} \neq p_{mo(i)-x}) \\
& \ \wedge (\forall\ k' : mo(k') \in [x,|p|+x) : p_{mo(k')-x} = S_{mo(k')+j}) \\
\Rightarrow & \qquad \{\text{ combine two quantifications and remove dependency on } S \text{ and transitivity of } = \} \\
& (\forall\ k : k \in [0,i) \wedge mo(k) \in [x,|p|+x) : p_{mo(k)} = p_{mo(k)-x}) \\
& \ \wedge ((i < |p| \wedge mo(i) \in [x,|p|+x)) \Rightarrow p_{mo(i)} \neq p_{mo(i)-x})
\end{array}
$$