Multiple Keyword Pattern Matching using Position Encoded Pattern Lattices

Fritz J. Venter[1], Bruce W. Watson[2], and Derrick G. Kourie[1]

[1] University of Pretoria, {fritz, dkourie}@fastar.org
[2] Stellenbosch University, bruce@fastar.org

Abstract. Formal concept analysis is used as the basis for two new multiple keyword string pattern matching algorithms. The algorithms addressed are built upon a so-called *position encoded pattern lattice* (PEPL). The algorithms presented are in conceptual form only; no experimental results are given. The first algorithm to be presented is easily understood and relies directly on the PEPL for matching. Its worst case complexity depends on both the length of the longest keyword, and the length of the search text. Subsequently a finite-automaton-like structure, called a *PEPL automaton*, is defined which is derived from the PEPL, and which forms the basis for a second more efficient algorithm. In this case, worst case behaviour depends only on the length of the input stream. The second algorithm's worst case performance is the same as the *matching phase* of the well-known (advanced) Aho-Corasick multiple-keyword pattern matching algorithm—widely regarded as the multiple keyword pattern matching algorithm of choice in contexts such as network intrusion detection. The first algorithm's performance is comparable to that of the *matching phase* of the lesser-known *failure-function* version of Aho-Corasick.

1 Overview

The *(multiple) keyword pattern matching problem* (in which the patterns are finite strings, or 'keywords') consists of finding *all occurrences* (including overlapping ones) of the keywords within an *input* string. Typically, the input string is much larger than the set of keywords and the set of keywords are fixed, meaning they can be preprocessed to produce data-structures for later use while processing the input string. We also make these assumptions in this paper, as this problem variant corresponds to many real-life applications in security, computational biology, etc [8]. Several decades of keyword pattern matching research have yielded many well-known algorithms, such as Knuth-Morris-Pratt, Boyer-Moore, Aho-Corasick, and Commentz-Walter. Overview articles are typically more accessible than the original literature—see [3, 4, 7] for comprehensive overviews and [10, 2] for taxonomies and correctness proofs of such algorithms.

In this text, the idea of "position encoding" of a set of patterns is introduced. This strategy serves as an alternative to traditional algorithms used to match

ISBN 978–84–695–5252–0, Universidad de Málaga (Dept. Matemática Aplicada)

a *set* of patterns. These algorithms typically rely on common pattern prefixes, suffixes and/or factors in general. The Aho-Corasick [1] algorithm (AC) is probably the best known and most widely used of these algorithms. Its best- and worst-case performance are both being linear in the size of the input stream and independent of the number of patterns to be matched.

Formal concept analysis (FCA) is used to leverage the potential benefits of position encoding. A formal context in which O is the set of *objects*, A is the set of *attributes*, and I is an *incidence relation* between the objects and the attributes will be denoted by $\mathbb{K} = \langle O, A, I\rangle$. The concept lattice, $\mathfrak{B}$, derived from the context $\langle O, A, I\rangle$ will be denoted by $\mathfrak{B}(\langle O, A, I\rangle)$. The extent, intent and set of own objects of a concept c in a concept lattice will be denoted by $\mathbf{extent}(c), \mathbf{intent}(c)$, and $\mathbf{ownobj}(c)$ respectively. The infimum of a set of concepts C will be denoted by $\mathbf{inf}(C)$. Finally, $\top(b)$ denotes the *attribute top* of attribute b—i.e. $\top(b)$ is the largest concept whose intent contains b.

In Section 2, it is shown how FCA can be used to construct a concept lattice from a position encoded set of patterns. Such a lattice is called a position encoded pattern lattice (PEPL). A first algorithm, called *PMatch*, is developed in Section 3, which takes such a PEPL together with text stream to be searched as input and produces the desired match occurrences as output, albeit in a rather inefficient way. As an alternative, a so-called *PEPL automaton* is defined in Section 4, based on the information in a PEPL. A second algorithm given in Section 5 uses this automaton and the text stream to be searched as input and also produces the desired match occurrences as output. However, in this instance the theoretical performance of the algorithm corresponds to that of Aho-Corasick. In a final section, we reflect on the implications of these results.

2 Position Encoded Pattern Lattices (PEPLs)

The length of string p will be denoted by $|p|$ and its $(i+1)^{st}$ element by p_i for $i \in [0, |p|)$. A *match occurrence* of a single pattern p in target s is a pair $\langle p, t\rangle$, such that $\forall\ k \in [0, |p|)$, $p_k = s_{t+k}$. The problem of matching a *set* of patterns P on target s can be defined as the requirement to construct the set of *all* match occurrences, denoted by MO in our algorithms.

Definition 1 (Position encoding of a set of patterns). *The* position encoding *of string w is the set of position-symbol pairs denoted by $\overrightarrow{w}^{\bullet}$ and is given by $\overrightarrow{w}^{\bullet} = (\bigcup\ k : k \in [1, |w|] : \{\langle k, w_{k-1}\rangle\})$.*

The position encoding *of a set of strings P is denoted $\overrightarrow{P}^{\bullet}$ and is given by $\overrightarrow{P}^{\bullet} = (\bigcup\ w : w \in P : \overrightarrow{w}^{\bullet})$*

For example, the position encoding of "pack" is $\overrightarrow{pack}^{\bullet} = \{\langle 1, p\rangle, \langle 2, a\rangle, \langle 3, c\rangle, \langle 4, k\rangle\}$, and of "packet" it is $\overrightarrow{packet}^{\bullet} = \{\langle 1, p\rangle, \langle 2, a\rangle, \langle 3, c\rangle, \langle 4, k\rangle, \langle 5, e\rangle, \langle 6, t\rangle\}$. In this

case, the position encoding of the set of patterns $P = \{\text{pack}, \text{packet}\}$ and of "packet" happens to be the same, i.e. $\overrightarrow{P}^{\bullet} = \overrightarrow{packet}^{\bullet}$.

Given any set of patterns, we can now constitute a formal context $K^{\#}$ along the following lines. Regard the words in the set of patterns as a set of objects. Let the position-symbol pairs of the position encoding of the set of patterns serve as attributes of these objects: a given word has as its attributes all the position-symbol pairs that make up its position-encoding.

As an example, consider the set of patterns $P = \{abc, aabc, abcc\}$. Table 1 shows the cross table that represents the position encoded formal context derived from P. This context can be denoted by $\langle P, \overrightarrow{P}^{\bullet}, I^{\overrightarrow{P}^{\bullet}} \rangle$, where $I^{\overrightarrow{P}^{\bullet}}$ is the incidence relation between objects and attributes depicted in the cross table. The formal concept lattice to be derived from such a context will be called a *Position Encoded Pattern Lattice* (PEPL), denoted by $\overrightarrow{\mathfrak{P}}^{\bullet}(\langle P, \overrightarrow{P}^{\bullet}, I^{\overrightarrow{P}^{\bullet}} \rangle)$ or, more concisely, by $\overrightarrow{\mathfrak{P}}^{\bullet}$. The cover graph of the underlying PEPL is shown in Figure 2.

$\langle P, \overrightarrow{P}^{\bullet}, I^{\overrightarrow{P}^{\bullet}} \rangle$	$\langle 1, a \rangle$	$\langle 2, a \rangle$	$\langle 2, b \rangle$	$\langle 3, b \rangle$	$\langle 3, c \rangle$	$\langle 4, c \rangle$
abc	×		×		×	
aabc	×	×		×		
abcc	×		×		×	×

Fig. 1: Position encoded context for $P = \{abc, aabc, abcc\}$.

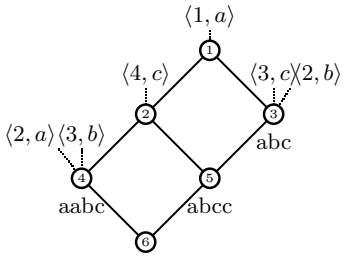


Fig. 2: Cover graph of PEPL for $P = \{abc, aabc, abcc\}$.

If a search of text s is currently at position $s[t]$, and it is found that $s[t+n] = a$, then attribute $\langle n, a \rangle$ is said to *positively check against* s at t.

It is evident that the intent of a PEPL concept has the following property: If all the attributes in the intent have been positively checked against a search text s at position t, then the (one or more) words that are own objects of the concept match the text, starting at position t. This is clearly the case for concepts 3, 4 and 5 with own objects "abc", "aabc" and "abcc" respectively.

3 PEPL-based Matching Using *PMatch*

Algorithm 1 described below is based on the insights of the previous section. Its top level procedure is called *PMatch*, which takes as input a PEPL $\overrightarrow{\mathfrak{P}}^{\bullet}(\langle P, \overrightarrow{P}^{\bullet}, I^{\overrightarrow{P}^{\bullet}} \rangle)$ (or simply $\overrightarrow{\mathfrak{P}}^{\bullet}$) and a text, s. It then finds in s all match occurrences of words

in P, recording them in MO. The algorithm is articulated in Dijkstra's guarded command language (GCL), widely used for its conciseness and precision [5]. The definition of *PMatch* assumes constant *minlength(P)* as the length of the shortest keyword in P. To avoid notational clutter, $\vec{\mathfrak{P}}$, s and MO are assumed to be globally accessible to all procedures. A special symbol, nil, is used to designate a non-existent concept, specifically the parent of the top concept.

PMatch calls *matchIntent* for each character in s where a match could possibly start (i.e. the tail is ignored). The condition in the associated for-loop is intended to signify that these probes are from left to right. In each call the intent of the top of the lattice and its non-existent parent, nil, are used as parameters.

matchIntent takes a string position t, and two concepts, c and p, as parameters. It is assumed that c is a child of p and the set difference, Δ, between their intents is computed. The special case of $\top$, which has no parent, is catered for. A loop checks whether all the attributes in Δ indicate positional matches in the text s as offset by the current search position, t—i.e. the loop removes from Δ all attributes of the form $\langle i, \alpha\rangle$ such that $s[t+i] = \alpha$. If this reduces Δ to the empty set, then a match occurrence is considered to have been found for each own object at c. Moreover, $match(c, t)$ can be called to investigate whether further match occurrences at t can be inferred by considering c's children. $match(p, t)$, in turn, simply sweeps through the children of p, recursively invoking *matchIntent* in each case.

Algorithm 1 *PEPL Based Matching*

proc *PMatch*($\vec{\mathfrak{P}}, s$)
  $MO, j := \varnothing, minlength(P)$;
  { *Traverse target string s from left to right* }
  for $(t \in [0, |s| - j + 1)) \rightarrow$
    matchIntent$(t, \top, nil)$
  rof
corp{ ***post*** : *MO is the set of match occurrences of P in s* }
proc *matchIntent*(t, c, p)
  if $(p = nil) \rightarrow \Delta :=$ **intent**(c)
  [] $(p \neq nil) \rightarrow \Delta :=$ **intent**$(c) \setminus$ **intent**(p)
  fi;
  do $(\exists \langle i, \alpha\rangle : \langle i, \alpha\rangle \in \Delta : (s[t + i - 1] = \alpha)) \rightarrow$
    $\Delta := \Delta \setminus \{\langle i, \alpha\rangle\}$
  od;
  if $(\Delta = \varnothing) \rightarrow MO := MO \cup$ **ownobj**$(c) \times \{t\}$;
    for ***all*** $c' \in children(c) \rightarrow matchIntent(t, c', c)$ **rof**
  [] $(\Delta \neq \varnothing) \rightarrow$ **skip**
  fi
corp

To illustrate how Algorithm 1 works, consider the keywords to match $P = \{abc, aabc, abcc\}$ and the target $s = aaabcdabccd$. The formal context $\langle P, \overrightarrow{P}^{\bullet}, I^{\overrightarrow{P}^{\bullet}} \rangle$ is given in Fig. 1 and the cover graph for the corresponding PEPL, $\overrightarrow{\mathfrak{P}}^{\bullet}$, is in Fig. 2. For convenience, the intents and own object sets of each concept are made explicit in Table 1. Table 2 provides a trace summary of calls to *matchIntent*. The first

Id of c	**intent**(c)	**ownobj**(c)
$1 = \top$	$\{\langle 1, a\rangle\}$	
2	$\{\langle 1, a\rangle, \langle 4, c\rangle\}$	
3	$\{\langle 1, a\rangle, \langle 3, c\rangle, \langle 2, b\rangle\}$	abc
4	$\{\langle 1, a\rangle, \langle 4, c\rangle, \langle 3, b\rangle, \langle 2, a\rangle\}$	aabc
5	$\{\langle 1, a\rangle, \langle 4, c\rangle, \langle 3, c\rangle, \langle 2, b\rangle\}$	abcc
$6 = \bot$	$\{\langle 1, a\rangle, \langle 4, c\rangle, \langle 3, b\rangle, \langle 3, c\rangle, \langle 2, b\rangle, \langle 2, a\rangle\}$	

Table 1: Details of concepts of the PEPL in Fig. 2

column shows t, the offset into s from which matching positions are calculated. The second and third columns show the lattice concept visited and its concept participating in the call to *matchIntent*. The fourth column marked Δ gives the set difference between the intent of the child and parent concept. A column per symbol in the string *aaabcdabccd* then follows. The last column gives the own object set to be to be used to update *MO* when a match has been found. Note that since $minlength(P) = 3$ and $|s| = 11$, the trace ranges over $t \in [0, 9)$. Each

t	c	p	Δ	a	a	a	b	c	d	a	b	c	c	d	**ownobj**(c)
0	1	nil	$\{\langle 1,a\rangle\}$	T											$\varnothing$
0	2	1	$\{\langle 4,c\rangle\}$				F								
0	3	1	$\{\langle 2,b\rangle, \langle 3,c\rangle\}$		F										
1	1	nil	$\{\langle 1,a\rangle\}$		T										$\varnothing$
1	2	1	$\{\langle 4,c\rangle\}$					T							$\varnothing$
1	4	2	$\{\langle 2,a\rangle, \langle 3,b\rangle\}$			T	T								$\{aabc\}$
1	3	1	$\{\langle 2,b\rangle, \langle 3,c\rangle\}$			F									
2	1	nil	$\{\langle 1,a\rangle\}$			T									
2	2	1	$\{\langle 4,c\rangle\}$						F						
2	3	1	$\{\langle 2,b\rangle, \langle 3,c\rangle\}$				T	T							$\{abc\}$
2	5	3	$\{\langle 4,c\rangle\}$						F						
3	1	nil	$\{\langle 1,a\rangle\}$				F								
4	1	nil	$\{\langle 1,a\rangle\}$					F							
5	1	nil	$\{\langle 1,a\rangle\}$						F						
6	1	nil	$\{\langle 1,a\rangle\}$							T					$\varnothing$
6	2	1	$\{\langle 4,c\rangle\}$										T		$\varnothing$
6	4	2	$\{\langle 2,a\rangle, \langle 3,b\rangle\}$								F				
6	5	2	$\{\langle 2,b\rangle, \langle 3,c\rangle\}$								T	T			$\{abcc\}$
6	3	1	$\{\langle 2,b\rangle, \langle 3,c\rangle\}$								T	T			$\{abc\}$
7	1	nil	$\{\langle 1,a\rangle\}$								F				
8	1	nil	$\{\langle 1,a\rangle\}$									F			

Table 2: Algorithm 1 trace: matching $\{abc, aabc, abcc\}$ in *aaabcdabccd*

row is a matching step of the algorithm—i.e. every row represents a call of the function *matchIntent* . As an example, the first row indicates that the matching position $t = 0$ and the attribute set to match is $\Delta = \{\langle 1,a\rangle\}$. The first (and only)

element of the set is $\langle i,\alpha\rangle = \langle 1,a\rangle$. This means that position $t+i=1$ is checked for the symbol $\alpha = a$, which is indeed the case as indicated by the "T" (for the boolean value *true*) shown in the first column for the target string. All "T" entries in the table indicate that attributes in Δ have been successfully matched in the do-loop of *matchIntent*. Once Δ has been reduced to $\varnothing$, *MO* has to be updated. Of course, if the concept has no own object—as is the case for the top concept marked 1—then nothing is added to *MO* (i.e. $\mathbf{ownobj}(c)\times\{t\} = \varnothing$). Subsequent calls to *match* without updating t, recursively deal with children concepts of the one currently under test. The second row of the table therefore logs the results the call to *matchIntent* made via *match* in respect of concept 2, the leftmost child of concept 1. In this case, the intent difference set is $\Delta = \{\langle 4,c\rangle\}$, and since $\nexists\langle i,\alpha\rangle : \{\langle 4,c\rangle\} : (s[t+i-1]=\alpha)$, (or, more explicitly, $s[0+4-1]=b$ and not c) *matchIntent* cannot reduce Δ to $\varnothing$. This is indicated by "F" (for false) as an entry in the relevant column of the table. Control now returns to *match*, where the next child of concept 1, namely concept 3, is considered. Further rows of the table illustrate the execution steps of Algorithm 1 for the rest of the target string.

PMatch eliminates sets of words from P that do not match in s without ever backing up in s, i.e. t is monotonically increasing. In this sense *PMatch* is an *online* algorithm, similar to the AC algorithm. However, *PMatch* sometimes *revisits* symbols in s. Such revisits are reflected by the multiple entries in various columns representing symbols in *aaabcdabccd* in Table 2.

The execution complexity of the matching process *per position checked in* s is bounded by the size of the PEPL. Table 2 shows how all concepts are visited when $t=6$. An (rather conservative) upper bound of the complexity of Algorithm 1 is therefore $(|\vec{\mathfrak{P}}^{\bullet}|\times|s|)$. The advanced AC algorithm is of course more efficient than this. Not only does it check every symbol in s exactly once; it also avoids the application of the expensive set difference operator that is applied in *matchIntent* of Algorithm 1. Instead, the advanced AC simply makes an automaton transition and considers whether an accepting state has been entered. In the upcoming sections, we refine our algorithm to arrive at a PEPL-based algorithm with similar performance characteristics to advanced AC.

4 PEPL Automata

For PEPL based matching to achieve the same order-of-magnitude performance as the advanced AC algorithm, this section defines a structure called a PEPL *Automaton*.

Firstly, the position encoded formal context for the set of keywords P is augmented. This augmented context has additional entries to reflect information about each keyword $p \in P$ whose first symbol matches the symbol at the m^{th} index of some other keyword, where $m > 0$.

Definition 2 (Augmentation operator). *For two strings p and y we define the operator denoted $\#$ as*

$$p\#y = \begin{cases} \{(p)y\} & \text{if } y \neq \varepsilon \wedge p \neq \varepsilon \\ \varnothing & \text{otherwise} \end{cases}$$

Definition 3 (Augmentation of a string). *We define the augmentation of string y with respect to string x as*

$$\langle x, y\rangle^{\#} = (\bigcup\ p, s, r, t : ((x = p \cdot s \wedge y = s \cdot r \wedge s \neq \varepsilon) \vee (x = p \cdot y \cdot t)) : p\#y\ )$$

Thus, for each proper suffix[3] s of x that is also prefix of y, we compute the singleton set $p\#y$ (but possibly the empty set) and add all such singleton sets into one big set. Note that there may be several such sets. For example, if $x = aa$ and $y = aaaa$ then the following decompositions of x are relevant :
$x = \varepsilon \cdot aa$; $x = a \cdot a$; so that

$$\begin{aligned} \langle x, y\rangle^{\#} &= \varepsilon\#aaaa \cup a\#aaaa \\ &= \varnothing \cup \{(a)aaaa\}\} \\ &= \{(a)aaaa\} \end{aligned}$$

Definition 4 (String-augmentation of a language). *We define the string-augmentation of language V with respect to string w as follows.*

$$\langle V, w\rangle^{\#} = (\bigcup\ v : v \in V : \langle v, w\rangle^{\#})$$

Then

$$\begin{aligned} \langle\{x, y\}, y\rangle^{\#} &= \{(a)aaaa, (aa)aaaa, (aaa)aaaa\} \\ \langle\{x, y\}, x\rangle^{\#} &= \{(a)aa, (aa)aa, (aaa)aa\} \\ \langle\{x\}, y\rangle^{\#} &= \{(a)aaaa\} \\ \langle\{y\}, x\rangle^{\#} &= \{(a)aa, (aa)aa, (aaa)aa\} \end{aligned}$$

Definition 5. *For a set of patterns P we define the augmented patterns as*

$$P^{\#} = P\ \cup\ (\bigcup\ p : p \in P : \langle P \setminus \{p\}, p\rangle^{\#})$$

Thus,

$$\begin{aligned} \{x, y\}^{\#} &= \{x, y\} \cup \langle\{x\}, y\rangle^{\#} \cup \langle\{y\}, x\rangle^{\#} \\ &= \{aa, aaaa\} \cup \{(a)aaaa\}\{(a)aa, (aa)aa, (aaa)aa\} \\ &= \{aa, aaaa, (a)aaaa, (a)aa, (aa)aa, (aaa)aa\} \end{aligned}$$

[3] By proper suffix, we mean that the empty string is not taken as a suffix.

$K^{\#}$	$\langle 1,a\rangle$	$\langle 2,a\rangle$	$\langle 2,b\rangle$	$\langle 3,c\rangle$	$\langle 4,c\rangle$	$\langle 3,b\rangle$	$\langle 5,c\rangle$
abc	×		×	×			
aabc	×	×			×	×	
abcc	×		×	×	×		
(a)abc	×	×			×	×	
(a)abcc	×	×			×	×	×

Table 3: Context derived by augmenting $P = \{abc, aabc, abcc\}$

Given set of patterns P, we can constitute a formal context denoted $K^{\#}$ for a PEPL using objects from the augmented set of patterns $P^{\#}$ and attributes from $\overrightarrow{P^{\#}}^{\bullet}$.

Example 1. As an example, consider again the set of patterns $P = \{abc, aabc, abcc\}$ augmented with the set of patterns $P^{\#}$ derived as follows:

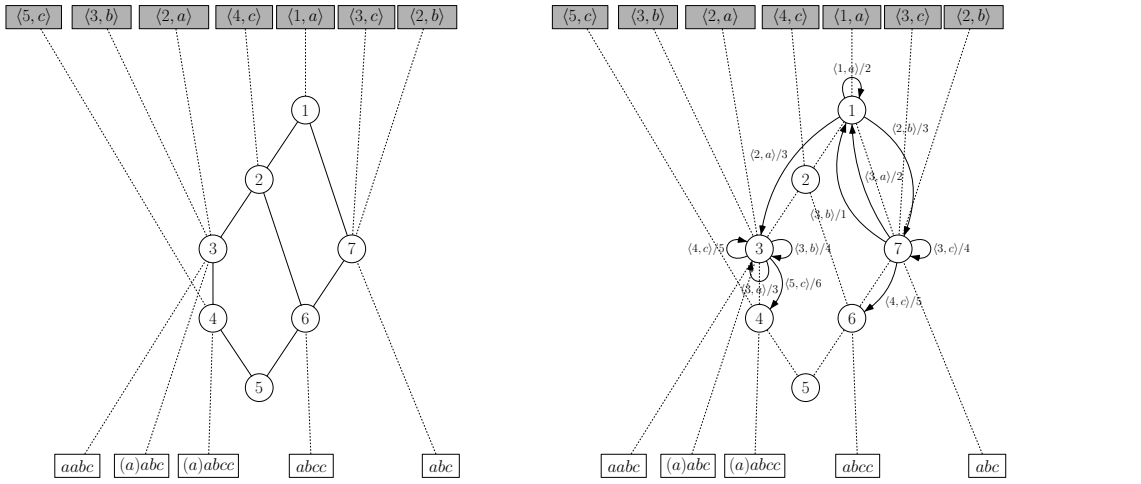
$$\begin{aligned} P^{\#} &= P \cup \langle\{aabc, abcc\}, abc\rangle^{\#} \cup \langle\{abc, abcc\}, aabc\rangle^{\#} \cup \langle\{abc, aabc\}, abcc\rangle^{\#} \\ &= P \cup \{(a)abc\} \cup \varnothing \cup \{(a)abcc\} \\ &= P \cup \{(a)abc, (a)abcc\} \\ &= \{abc, aabc, abcc, (a)abc, (a)abcc\} \end{aligned}$$

Table 3 depicts the context derived from the set of objects $P^{\#}$ and their corresponding position encoding as attributes.

Algorithm 2 discussed in Section 5 uses $\overrightarrow{\mathfrak{P}}^{\bullet}$ to match a set of keywords P against a target string. The algorithm relies on a second pre-processing step after $\overrightarrow{\mathfrak{P}}^{\bullet}$ has been generated from its context. This step traverses $\overrightarrow{\mathfrak{P}}^{\bullet}$ to create a special automaton, called a "Position Encoded Pattern Lattice Automaton" or simply a PEPL Automaton. It is defined as follows.

Definition 6 (PEPL Automaton). *Given $\overrightarrow{\mathfrak{P}}^{\bullet}$, the PEPL that has been derived from set of patterns $P^{\#}$, the associated* PEPL automaton *is a five-tuple $\langle Q^{\overrightarrow{P}^{\bullet}}, V^{\overrightarrow{P}^{\bullet}}, \delta^{\overrightarrow{P}^{\bullet}}, {q_0}^{\overrightarrow{P}^{\bullet}}, F^{\overrightarrow{P}^{\bullet}}\rangle$ such that:*

- $Q^{\overrightarrow{P}^{\bullet}} \subseteq \{0\} \cup (\bigcup\ q, c : q = id(c) \wedge c \in \overrightarrow{\mathfrak{P}}^{\bullet} : \{q\})$ *is regarded as the automaton's set of states, where id(c) simply gives a unique numerical identifier for concept c.*
- $V^{\overrightarrow{P}^{\bullet}} = \overrightarrow{P}^{\bullet}$ *is regarded as the automaton's alphabet, indicating position-symbol pairs.*
- $\delta^{\overrightarrow{P}^{\bullet}} : Q^{\overrightarrow{P}^{\bullet}} \times V^{\overrightarrow{P}^{\bullet}} \nrightarrow Q^{\overrightarrow{P}^{\bullet}} \times |{P^{\#}}_{Max}|$ *is the automaton's transition function. The mapping is generally determined by the recursive relationship:*



(a) Cover Graph for the PEPL for the context of $P^{\#}$

(b) PEPL Automaton superimposed on the Cover Graph

Fig. 3: PEPL derived from $P^{\#}$

$\delta^{\overrightarrow{P\bullet}}(id(c), \langle i, \alpha\rangle) = id(c')/i'$ *where variables* c' *and* i' *are defined as follows:*

$$\langle c', i'\rangle = \begin{cases} \langle c, i\rangle & \text{if } A(c,i,\alpha) \ \wedge B(c,i,\alpha) \\ \langle \top(\langle 1, \alpha\rangle), 2\rangle & \text{if } A(c,i,\alpha) \ \wedge \neg B(c,i,\alpha) \wedge \top(\langle 1,\alpha\rangle) \neq \bot \\ \langle \top, 1\rangle & \text{if } A(c,i,\alpha) \ \wedge \neg B(c,i,\alpha) \wedge \top(\langle 1,\alpha\rangle) = \bot \\ \langle \mathbf{inf}(\{c, \top(\langle i, \alpha\rangle)\}), i+1\rangle & \text{otherwise} \end{cases}$$

where $A(c,i,\alpha) \equiv \mathbf{inf}(\{c, \top(\langle i,\alpha\rangle)\}) = \bot$ *and* $B(c,i,\alpha) \equiv \langle i-1, \alpha\rangle \in \mathbf{intent}(c)$

The the first level recursion starts off with $c = \top$*. The notation* q/x *means that when a transition to state* q *is made, the automaton produces the additional value* x*.*

- $q_0{}^{\overrightarrow{P\bullet}} = \top$ *is the automaton's start state, which is also the top concept of the PEPL.*
- $F^{\overrightarrow{P\bullet}} = (\bigcup\ q, c : q = id(c) \wedge c \in Q^{\overrightarrow{P\bullet}} \wedge |\mathbf{ownobj}(c)| > 0 : \{q\})$ *is the automaton's set of final states.*

The above definition embeds sufficient information to derive algorithmically a DFA whose transition diagram can be superimposed on the cover graph of the PEPL.

It is assumed below that an function, *getFA*, is available which delivers a PEPL automaton $\mathcal{M}^{\overrightarrow{P}}$ when provided with a PEPL. As an example, *getFA*($\overrightarrow{\mathfrak{P}}$) will return $\mathcal{M}^{\overrightarrow{P}}$, the *PEPL-Automaton* (partially) shown in Fig. 3b, superimposed over the cover graph for $\overrightarrow{\mathfrak{P}}$. Note that in order to avoid clutter, a number of arcs have not been shown in Fig. 3b. For example, the many of transitions to $\top$ have been left out.

5 Matching Using a *PEPL-Automaton*

Viewed as a DFA, a PEPL automaton could be used to test whether a given sequence of its alphabet are in the regular set of patterns that it describes. For example, it can easily be seen in Figure 3 that, starting from $\top$, successive transitions on elements of the string $\langle\langle 1, a\rangle, \langle 2, b\rangle, \langle 3, c\rangle\rangle$ lead to the final state 6, affirming that this sequence is indeed part of the set of patterns described by the automaton, and and since *abc* is an own object of 6, affirming that *abc* is in the original set of patterns, P.

However, the PEPL is not primarily intended to be used in this way. Instead, the PEPL is used in Algorithm 2 to find all match opportunities in P in a text s. The algorithm's **do** processes symbols of s, updating variables c and i to keep track of partial matches in that part of s already processed. This is expressed as loop *invariant* $Inv(c, i, t) \equiv$

- MO contains all matches in $s_{[0,t-i+1)}$. (These are the matches *already processed.*)
- And $s_{[t-i+1,t)}$ matches the first $i-1$ characters of all patterns in **extent**(c). (These are the partial matches *in progress.*)

To illustrate matching as executed by Algorithm 2, consider the steps logged in Table 4 when matching the set of patterns *abc,aabc,abcc* against the target string *aaabcdabccd*. The first five entries of each row in this table shows the values of variables $t, s[t], c, i$ as they have been updated as a result of the statements in the body of the main loop in Algorithm 2. The next set of entries in the respective row are all empty except for the position where $s[t]$ has been matched. At such a position a "T" or "F" is shown depending on the result of the match. The next entry in the row shows the size of the intent of c. The last entry in this row gives the patterns matched at the step represented by the row.

Algorithm 2 *PEPL Automaton Based Matching*

proc *PAutMatch*($\overrightarrow{\mathfrak{P}}, s$)
  $MO := \varnothing$;
  $\mathcal{M}^{\overrightarrow{P}} := getFA(\overrightarrow{\mathfrak{P}})$;
  $\langle c, i\rangle, t := \langle {q_0}^{\overrightarrow{P}}, 1\rangle, 0$; { *Recall that* ${q_0}^{\overrightarrow{P}} = \top$ }

{ ***invariant:*** $Inv(c, i, t)$ }
do $(t < |s|) \rightarrow \langle c, i\rangle, t := \delta^{\overrightarrow{P}}(c, \langle i, s[t]\rangle), t + 1;$
if $(i = |\mathbf{intent}(c)|) \rightarrow MO := MO \cup (\{t\} \times \mathbf{ownobj}(c))$
[] $(i \neq |\mathbf{intent}(c)|) \rightarrow \mathbf{skip}$
fi
od
corp { ***post*** : *MO is the set of match occurrences of P in s* }

As an example we present an explanation of the steps up to the first matched patterns being recorded. Consider the first row of Table 4. This row represents the first step of the matching process. After this step, a transition is made from the start state ($c = \top$) to the same state ($c' = \top$). The transition is due to the transition function $\delta^{\overrightarrow{P}}(c, \langle i, s[t]\rangle)$ returning the value $\top/2$ for the offset variable $i = 1$ and symbol $s[t = 0] = a$. The last entry in the row is empty as the top node does not contain any own objects. The next row shows the transition $\delta^{\overrightarrow{P}}(\top, \langle 2, a\rangle) = 3/3$ due to value of the variable t being incremented from its value in the previous step. This process continues until the patterns $aabc, (a)abc$ are recorded when the variables i and t are both (coincidentally) equal to 4 and state 4 is reached in the 5^{th} row of the table. Recall that a match is recorded for a state that is represented by a concept such that the size of such concept's intent (as shown in the second last entry) is the same as the offset variable i.

c_o	i_o	t	$s[t]$	c	i	a	a	a	b	c	d	a	b	c	c	d	$\lvert\mathbf{intent}(c')\rvert$	{Patterns Matched}
$\top$	1	0	a	$\top$	2	T											1	∅
$\top$	2	1	a	3	3		T										2	∅
3	3	2	a	3	3			T									4	∅
3	3	3	b	3	4				T								4	∅
3	4	4	c	4	5					T							4	{aabc,(a)abc}
4	5	5	d	$\top$	1						F						5	∅
$\top$	1	6	a	$\top$	2							T					5	∅
$\top$	2	7	b	7	3								T				3	∅
7	3	8	c	7	4									T			3	{abc}
7	4	9	c	6	5										T		4	{abcc}
6	5	10	d	$\top$	1											F	5	∅

Table 4: Positions visited in *aaabcdabccd* when matching *abc,aabc,abcc*

We have arrived at a particularly efficient algorithm, thanks to two observations about *PAutMatch* . Firstly, transitions in the PEPL-Automaton ($\delta^{\overrightarrow{L}}$) can be done in constant time using a lookup table. Secondly, the **if** statement can be made in constant time, and consists of simple integer arithmetic to advance through the lattice and target s, and an update of MO (only if a match has been found). The latter can be done using a precomputed lookup table, as is done in the advanced AC algorithm. These two characteristics are also found in the advanced AC algorithm, and is unavoidable in pattern matching algorithms, giving us the same exact (worst- and best-case) running time of $|s|$.

The example in Table 4 illustrates how, in contrast to the example in Table 2, each symbol in the string *aaabcdabccd* is visited exactly once to match the keywords $\{abc, aabc, abcc\}$.

6 Conclusion

The application of FCA in pattern matching was first introduced in [9]. There, two-dimensional pattern information was encoded into a concept lattice which was subsequently used as the basis for traversing a two-dimensional space in search of a specific pattern. Here, by contrast, common information about multiple keywords is encoded into a PEPL, to form the basis for discovering positional information about matching instances of those keywords in a linearly streamed text. The two new pattern matching algorithms are shown to have theoretical running-time comparable to the Aho-Corasick family of algorithms.

Ongoing work involves benchmarking the new algorithms against the Aho-Corasick and other multiple keyword pattern matching algorithms. We are also finding ways in which FCA can be effectively used in other stringology contexts [6].

References

1. Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.
2. Loek Cleophas, Bruce W. Watson, and Gerard Zwaan. A new taxonomy of sublinear right-to-left scanning keyword pattern matching algorithms. *Science of Computer Programming*, 75:1095–1112, 2010.
3. Maxime A. Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, 1994.
4. Maxime A. Crochemore and Wojciech Rytter. *Jewels of Stringology*. World Scientific Publishing Company, 2003.
5. Derrick G. Kourie and Bruce W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer Verlag, 2012.
6. Derrick G. Kourie, Bruce W. Watson, Loek Cleophas, and Fritz Venter. Failure deterministic finite automata. In Jan Holub, editor, *Proceedings of the Prague Stringology Conference (PSC)*. Czech Technical University, August 2012.
7. William F. Smyth. *Computing Patterns in Strings*. Addison-Wesley, 2003.
8. George Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann, 2004.
9. Fritz Venter, Derrick G. Kourie, and Bruce W. Watson. FCA-based two dimensional pattern matching. In *Proceedings of the 7th International Conference on Formal Concept Analysis*, 2009.
10. Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Eindhoven University of Technology, September 1995.