Information Processing Letters 110 (2010) 216–220

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

A faster algorithm for matching a set of patterns with variable length don't cares

Meng Zhang [a], Yi Zhang [b,a], Liang Hu [a,*,1]

[a] College of Computer Science and Technology, Jilin University, Changchun, China
[b] Department of Computer Science, Jilin Business and Technology College, Changchun, China

ARTICLE INFO

Article history:
Received 9 July 2009
Received in revised form 14 December 2009
Accepted 16 December 2009
Available online 21 December 2009
Communicated by M. Yamashita

Keywords:
Algorithms
String matching
Variable length don't cares

ABSTRACT

We present a simple and faster solution to the problem of matching a set of patterns with variable length don't cares. Given an alphabet Σ, a pattern p is a word $p_1@p_2\cdots@p_m$, where p_i is a string over Σ called a keyword and $@\notin\Sigma$ is a symbol called a variable length don't care (VLDC) symbol. Pattern p matches a text t if $t = u_0p_1u_1\ldots u_{m-1}p_mu_m$ for some $u_0,\ldots,u_m\in\Sigma^*$. The problem addressed in this paper is: given a set of patterns $\mathcal{P}$ and a text t, determine whether one of the patterns of $\mathcal{P}$ matches t.

Kucherov and Rusinowitch (1997) [9] presented an algorithm that solves the problem in time $O((|t|+|\mathcal{P}|)\log|\mathcal{P}|)$, where $|\mathcal{P}|$ is the total length of keywords in every pattern of $\mathcal{P}$. We give a new algorithm based on Aho–Corasick automaton. It uses the solutions of Dynamic Marked Ancestor Problem of Chan et al. (2007) [5]. The algorithm takes $O((|t|+\|\mathcal{P}\|)\log\kappa/\log\log\kappa)$ time, where $\|\mathcal{P}\|$ is the total number of keywords in every pattern of $\mathcal{P}$, and κ is the number of distinct keywords in $\mathcal{P}$. The algorithm is faster and simpler than the previous approach.

1. Introduction

The problem of matching a pattern with variable length don't cares (VLDC) is first studied by Fischer and Paterson [7]. A VLDC pattern p is a word $p_1@p_2\cdots@p_m$, where p_i is a string over an alphabet Σ called a keyword and $@\notin\Sigma$ is the variable length don't care symbol. Pattern p matches a text t if $t = u_0p_1u_1\ldots u_{m-1}p_mu_m$ for some $u_0,\ldots,u_m\in\Sigma^*$. The problem addressed in this paper is: given a set of patterns $\mathcal{P}$ and a text t, determine whether one of the patterns of $\mathcal{P}$ matches t.

By allowing to preprocess the text, [8,10,11] gave efficient solutions. In this paper, we study the multi-VLDC pattern matching problem without preprocessing the text, that is the patterns are known but inputs are unknown beforehand. This problem arises in many applications in practice, including intrusion detection systems [14], anti-virus systems [13] and bioinformatics [10]. For instance, in anti-virus systems, many signatures of malicious codes such as worms or virus can be described by VLDC patterns. For example, the signature of worm Trojan.URLspoof.gen [13] is 6c6f636174696f6e2e687265663d756e6573636170652827@3a2f2f@25303140@2729. In modern anti-virus systems, there are a huge number of such signatures. Efficient multi-VLDC pattern matching algorithms are very useful in these applications.

Kucherov and Rusinowitch [9] proposed an algorithm that solves the problem in time $O((|t|+|\mathcal{P}|)\log|\mathcal{P}|)$, where $|\mathcal{P}|$ is the total length of keywords in every pattern of $\mathcal{P}$, and $|t|$ is the length of input text. This algorithm is different from most of the existing string matching algorithms: Instead of composing of two successive stages: preprocessing the pattern and reading through the text, the algorithm has these two stages interleaved. It uses the DAWG (Directed Acyclic Word Graph) [3,6,4] as a multi-pattern matching automaton. In this algorithm, the pattern set of

* Corresponding author.
E-mail addresses: zhangmeng@jlu.edu.cn (M. Zhang), whdzy2000@vip.sina.com (Y. Zhang), hul@mail.jlu.edu.cn (L. Hu).

[1] Supported by National Natural Science Foundation of China (0473099).

0020-0190/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.ipl.2009.12.007

DAWG is changing over time. It makes use of the dynamic tree data structure introduced by Sleator and Tarjan [12] for dynamic dictionary matching. This data structure solves the problem of *dynamic nearest marked ancestor query*, by which finding the closest marked ancestor of a node takes $O(\log n)$ time in an n-node tree.

Two balanced parentheses based approaches have been proposed in [2,5]. In [2], the *marking* and *unmarking* of a node can be done in time $O(\log n/\log\log n)$. So does the operation of finding the closest marked ancestor of a node. In [5] Chan et al. presented the first $O(n)$-bit structure that supports $O(\log n/\log\log n)$ time per operation. The structure matches the performance of the best dynamic result, but with reduced space requirements. By plugging these solutions, the Kucherov–Rusinowitch algorithm can be improved by the $\log\log|\mathcal{P}|$ factor.

In this paper, we give an algorithm based on Aho–Corasick automaton (AC for short) [1]. It solves the problem of multi-VLDC pattern matching in time $O((|t| + \|\mathcal{P}\|)\log\kappa/\log\log\kappa)$, where $\|\mathcal{P}\|$ is the total number of keywords in every pattern of $\mathcal{P}$, and κ is the number of distinct keywords in $\mathcal{P}$. The algorithm uses an AC automaton for all the keywords in $\mathcal{P}$. In the run of the algorithm, we don't delete or insert any node of the AC automaton. The inserting and deleting of a keyword are done by marking or unmarking a corresponding node. We use the dynamic data structure of Chan et al. [5] to organize the marked nodes and use a circular queue to determine when a keyword should be marked. By this approach, we need not load or unload the keywords letter by letter in the run as in [9]. The improvement on the time complexity of the problem is achieved partly due to this strategy.

The essential improvement on Kucherov–Rusinowitch algorithm of our approach is reducing $|P|$ to $\|P\|$ in the time complexity. In practise, some applications will benefit from this improvement. In ClamAV [13] anti-virus system, the string patterns in signatures are quite long (an average of 124 bytes), so, $\|P\|$ and κ are significantly small compared to $|P|$. For this application, our algorithm has advantages over the Kucherov–Rusinowitch algorithm.

2. Notions

The exact multi-pattern matching problem is defined as finding all the occurrences of any pattern in a set of strings K in a text t. This problem was firstly solved by Aho and Corasick [1]. They created a finite state machine for matching the patterns with one pass scanning of the text. An Aho–Corasick automaton is an uncompressed trie of the set of patterns. The root node is denoted by R. For each node v, let $L(v)$ denote the label along the path from the root to v. $L(v)$ represents a prefix of one or more patterns; if $L(v)$ is a pattern, we add a *terminal* flag to v. The trie is represented by *GoTo* function. The *GoTo* function maps a pair consisting of a state and a symbol to a state or the message fail. In detail, if $L(v)a$ is a prefix of some patterns, then $GoTo(v,a) = v'$ where $L(v') = L(v)a$; otherwise $GoTo(v,a) = \text{fail}$.

Each node of AC automaton is associated with a function *failure*. For a node v, $failure(v)$ returns the node v', such that $L(v')$ is the longest proper suffix of $L(v)$ that is also a prefix of some patterns. The series $failure(v)$, $failure^2(v), \ldots$ ends with R, called the *failure chain* of v. For every AC trie node, we associate it with a pointer to the closest terminal node in its failure chain. The terminal function of a node v, say $terminal(v)$, is the node v', such that $L(v')$ is the longest proper suffix of $L(v)$ that is also a pattern. We also call $terminal(v)$ the *terminal link* of v. The series $terminal(v), terminal^2(v), \ldots$ ends with R, called the *terminal chain* of v. These three functions, *GoTo*, *failure* and *terminal*, completely define an AC automaton. We take the root node of the AC automaton as a terminal node. The nodes and terminal links of the AC automaton form a tree called *terminal tree*.

We use $|K|$ to denote the total size of all the strings in a string set K, and use Σ to denote the alphabet and σ to denote the number of distinct characters that appear in K. The AC algorithm builds the AC automaton in time $O(|K|\log\sigma)$ and searches a text of length n in $O((n + tocc)\log\sigma)$ time, where $tocc$ is the total number of occurrences reported. The time complexity of this algorithm is considered linear as the $\log\sigma$ factor is usually implicit while stating the time bounds.

3. The pattern matching algorithm

We use $\mathcal{P}$ to denote a set of VLDC patterns, $\mathcal{K}$ to denote the set of distinct keywords in $\mathcal{P}$. We say that pattern $p = p_1@p_2\cdots@p_m$ matches a text t if $t = u_0p_1u_1\ldots u_{m-1}p_mu_m$ for some $u_0,\ldots,u_m \in \Sigma^*$.

Let $t = u_1p_1u_2\ldots u_xp_xu_{x+1}$, where $x \leqslant m$. This decomposition of t, say Oc, is called a *prefix occurrence* of p in t. Denote $|Oc| = x$. Let $end(Oc,i) = \sum_{j=1}^{i}(|u_j| + |p_j|)$, if $i \leqslant |Oc|$; otherwise $end(Oc,i) = \infty$. There may be many prefix occurrences of p in t. We can identify a prefix occurrence Oc by the set $\{end(Oc,i) \mid 1 \leqslant i \leqslant m\}$. The *leftmost* prefix occurrence of pattern p in t is a prefix occurrence, say $LmOcc$, such that $|LmOcc|$ is the maximal and $end(LmOcc,i)$ is the minimal for $1 \leqslant i \leqslant m$ among all the prefix occurrences of p in t. In other words, the number of matched keywords of p in $LmOcc$ is maximal; every matched keyword of p in $LmOcc$ is not on the right of the corresponding keyword in other prefix occurrences.

We first give the idea of the algorithm. It reads the text from left to right. After processing a letter in the text, the algorithm records the leftmost prefix occurrences of every pattern in the text scanned so far. It searches for a set of keywords that is the set of the first unmatched keywords of every pattern. Denote this set by *Cur_K_Set*. That is, if any keyword in *Cur_K_Set* is matched, the number of matched keywords of some patterns will increase by one. Then the leftmost occurrence of a pattern in $\mathcal{P}$ will be found.

We give the formal description of the algorithm. The pseudo code of the algorithm is given in Fig. 1. Initially, the algorithm builds an AC automaton for $\mathcal{K}$ where only R is marked as terminal. Let $t[1:l] = t[1]\ldots t[l]$ be a prefix of t scanned so far. Let Oc be the leftmost prefix occurrence of $p \in \mathcal{P}$ in $t[1:l]$, denote $l - \sum_{i=1}^{|Oc|}(|u_i| + |p_i|)$ by $tail\_Len(p,l)$, and denote $|Oc|$ by $last(p,l)$. For each keyword k in $\mathcal{K}$, we define a set of patterns associated with

Procedure $Update\_State(s, a)$
Input: s is a state; a is a symbol.

```
1: while GoTo(s, a)=fail do
2:     s ← failure(s)
3: end while
4:  return GoTo(s, a)
```

Algorithm $MATCH(t, \mathcal{P})$
Input: t is the input word; $\mathcal{P}$ is the pattern set.

```
1: active ← root; pool_curr ← 0; l ← 1
2: Pool_Len ← Length of the longest keyword in K
3: for each k ∈ K do
4:     wait[k] ← ∅
5: end for
6: for i ← 0 up to Pool_Len − 1 do
7:     pool[i] ← ∅
8: end for
9: for each p ∈ P do
10:    pool[|p_1| − 1] ← pool[|p_1| − 1] ∪ {⟨p, 1⟩}
11: end for
12: while l ≤ |t| do
13:    for each pair ⟨p, i⟩ ∈ pool[pool_curr] do
14:        Mark(p_i)
15:        pool[pool_curr] ← pool[pool_curr] \ {⟨p, i⟩}
16:        wait[p_i] ← wait[p_i] ∪ {⟨p, i⟩}
17:    end for
18:    active ← Update_State(active, t[l])
19:    for each marked node k ≠ root in terminal chain
       of active do
20:        for each ⟨p, j⟩ ∈ wait[k] do
21:            if j equals the number of keywords in p
               then
22:                output(find a match of p) and STOP
23:            end if
24:            wait[k] ← wait[k] \ {⟨p, j⟩}
25:            d ← (pool_curr + |p_{j+1}|) mod Pool_Len
26:            pool[d] ← pool[d] ∪ {⟨p, j + 1⟩}
27:        end for
28:        UnMark(k)
29:    end for
30:    l ← l + 1
31:    pool_curr ← (pool_curr + 1) mod Pool_Len
32: end while
33: output(t doesn't have any matching of P)
```

Fig. 1. Algorithm *MATCH*. Lines 13–18 are stage 1, lines 19–31 are stage 2.

k when scanning $t[l]$: If k is a suffix of $t[1:l]$ then for every p in this set, the number of matched keywords of p in $t[1:l]$ is increased by one compared with that in $t[1:l-1]$, *i.e.* $last(p,l) = last(p,l-1)+1$. We precisely define the set $WAIT(k,l)$ as follows.

$$WAIT(k,l) = \big\{\langle p, x+1\rangle \;\big|\; p \in \mathcal{P},\ x = last(p,l-1),\ p_{x+1} = k \text{ and } tail\_Len(p,l-1)+1 \geqslant |k|\big\}. \tag{1}$$

In the algorithm, we use array **wait**$[k]$ for every $k \in \mathcal{K}$ to represent $WAIT(k,l)$ in each lth iteration. Let $Cur\_K\_Set(l)$ denote the set of keywords whose *WAIT* set is not empty in each lth iteration, called *waiting* keywords set. Let $y = last(p,l-1)+1$. Then p_y is a waiting keyword. Assume that p_y is matched when scanning $t[l]$. According to the definition of *WAIT*, pair $\langle p, y+1\rangle$ belongs to $WAIT(p_{y+1}, l+|p_{y+1}|)$, but doesn't belong to $WAIT(p_{y+1}, l')$ for $l < l' < l+|p_{y+1}|$. This is because the adjacent keywords in the occurrences of a pattern are not overlapped.

We use a circular queue named **pool** to guarantee **wait**$[k] = WAIT(k,l)$ in each lth iteration. Each entry of **pool** is a set of pairs consisting of a pattern and an index of its keyword. The length of **pool** is $Pool\_Len = \max(|k| \mid k \in \mathcal{K})$. The current focusing position in **pool** is denoted by *pool_curr* when scanning $t[l]$. Position *pool_curr* moves rightward synchronously with the focusing position of the text. Then when a waiting keyword k is matched at the moment of scanning $t[l]$, for every pair $\langle p, y\rangle$ in $WAIT(k,l)$, we add a pair $\langle p, y+1\rangle$ to **pool**$[(pool\_curr + |p_{y+1}|) \bmod Pool\_Len]$. By lines 13–18 of *MATCH*, the algorithm fetches each pair, say $\langle p, i\rangle$, from set **pool**$[pool\_curr]$, then add $\langle p, i\rangle$ to **wait**$[p_i]$. Thus, after $|p_{y+1}|$ times of text scanning, the pair $\langle p, y+1\rangle$ will be inserted to **wait**$[p_{y+1}]$ after line 18 in the $(l+|p_{y+1}|)$th iteration. This delay of inserting guarantees **wait**$[k] = WAIT(k,l)$ in each lth iteration.

We keep a state of AC automaton called *active*, such that $L(active)$ is the longest prefix of keywords in $\mathcal{K}$ in the suffixes of $t[1:l]$. The *Update_State* procedure updates *active*. In order to make the algorithm neat, we import the node $\perp$ such that $GoTo(\perp, a) = root$, for any $a \in \Sigma$ and $failure(R) = \perp$. An example of the run of the algorithm is given in Fig. 2.

3.1. Dynamic marked terminal tree

Our algorithm only cares about the matchings of keywords in Cur_K_Set of each moment of text scanning. Other than keeping an AC automaton for Cur_K_Set by inserting and deleting keywords dynamically as in [9], the AC automaton we use is of set $\mathcal{K}$, but only the keywords in Cur_K_Set are marked. Then only the matchings of these keywords will be reported. By the dynamically marking of nodes of the AC automaton, the times of report of the matchings of keywords are bounded by $\|\mathcal{P}\|$. Function $Mark(k)$ marks the node corresponding to k, which means adding keyword k to Cur_K_Set. Function $UnMark(k)$ clears the mark of the node corresponding to k, which means deleting keyword k from Cur_K_Set.

String matching for a dynamic pattern set is called *Dynamic Dictionary Matching* [2] in the literature. In order to report the matchings of keywords in Cur_K_Set, we need to find all the marked keywords in the ancestors of a node in the terminal tree of the AC automaton for $\mathcal{K}$. The problem amounts to finding the closest marked ancestor in a dynamically marked tree.

In this paper, we use the solution of Chan et al. [5] for this problem. By this method, finding the closest marked ancestor of a node in a dynamic tree can be done in time

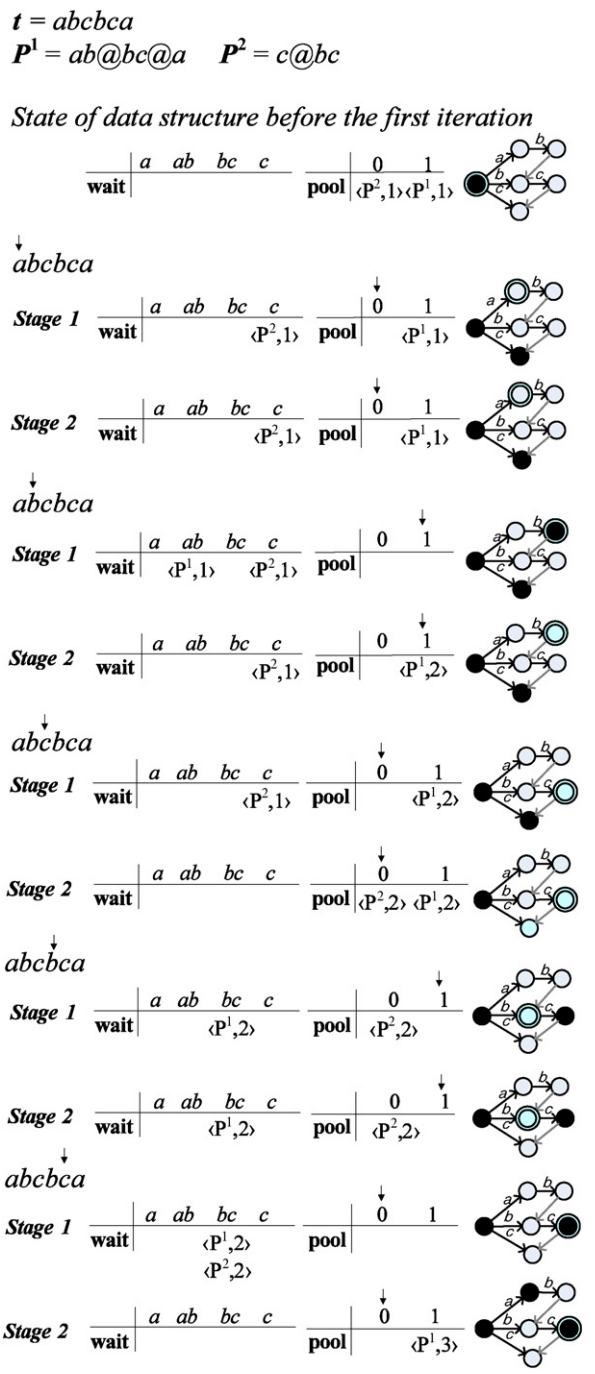

Fig. 2. The run of $MATCH(t, \{P^1, P^2\})$ for $t = abcbca$ and $P^1 = ab@bc@a$, $P^2 = c@bc$. Each line gives the state of **wait**, **pool**, and AC automaton after a stage. In the AC automaton, black nodes are marked nodes, grey edges are fail links, the double circled node is *active*. The fail links of children of root are omitted. We use a downward arrow to indicate *pool_curr* and the current scanning letter.

$O(\log n / \log\log n)$ where n is the number of tree nodes. So does the mark and unmark operations. The space is $o(n)$. The dynamic tree we use is a slight modification of the terminal tree of the AC automaton. It can be constructed by the following steps: First, the initial tree is of all the terminal nodes of the AC automaton and terminal links of these nodes. Then we add a son node to each terminal node, namely *young son*; all the young sons will be marked permanently in the run. The resulting tree is the tree we use, denoted by *FT*. At the beginning of the algorithm, only young sons and the root are marked.

Using the method of [5], we represent *FT* by a list of balanced parentheses and a B-tree built from it. Each *FT* tree node is represented by a pair of parentheses. Marking or unmarking a node amounts to deleting or inserting a pair of parentheses and updating of the B-tree. Finding the closest marked ancestor of a node amounts to finding the nearest enclosing parentheses. We associate $terminal(v)$ of each node v with the open parentheses corresponding to the young son of $terminal(v)$ in *FT*. The number of nodes in *FT* is $2\kappa + 1$. Therefore, each of finding the closest marked ancestor in *FT*, marking and unmarking a node uses time $O(\log\kappa / \log\log\kappa)$.

4. Correctness of the algorithm

To prove the correctness of the algorithm, we verify three conditions hold in the run of algorithm *MATCH*. In each main iteration (lines 12–32) of algorithm *MATCH*, lines 13–18 are stage 1 and lines 19–31 are stage 2. After stage 1 of the lth iteration, the following three conditions imply the correctness of our algorithm:

1. For every keyword k, we have $\mathbf{wait}[k] = WAIT(k, l)$.
2. For node *active* in algorithm *MATCH*, string $L(active)$ is the longest one that is a prefix of keywords in $\mathcal{K}$ and also a suffix of $t[1 : l]$.
3. The set of the marked nodes is $Cur\_K\_Set(l)$.

We prove by induction on l that the whole data structure, *i.e.* **wait**, *active* and *FT*, satisfies conditions l–3 after stage 1 in every iteration.

First, conditions l–3 are true after stage 1 in the first iteration. Assume that $t[1 : l]$ has been scanned, *i.e.* stage 1 of the lth iteration has finished; the current data structure satisfies conditions l–3. We go to prove that the data structure resulting from stage 1 of the $(l+1)$th iteration satisfies conditions l–3.

We first focus on condition 1. Assume that $\mathbf{wait}[k] = WAIT(k, l)$ for each $k \in \mathcal{K}$ after stage 1 of the lth iteration. In the following two stages, **wait** is updated:

(1) **Stage 2 of the lth iteration**. For any matched waiting keyword k in the suffix of $t[1 : l]$, we have $\mathbf{wait}[k] = \emptyset$ after stage 2 of the lth iteration. Since $\mathbf{wait}[k] = WAIT(k, l)$, then after the lth iteration, for each $k \in \mathcal{K}$, $\mathbf{wait}[k]$ is the set of pairs $\langle p, i\rangle$ such that $i = last(p, l) + 1$, $p_i = k$ and $tail\_Len(p, l) \geqslant |k|$.

(2) **Stage 1 of the $(l+1)$th iteration**. Consider line 15 and lines 25–26, which are the operations on **pool**. After the lth iteration, for every pattern p such that $|p_{last(p,l)+1}| = tail\_Len(p, l) + 1$, pair $\langle p, last(p, l) + 1\rangle$ has been added to the current focusing entry of **pool**. For each pair $\langle p, i\rangle$ in $\mathbf{pool}[pool\_curr]$, we have $i = last(p, l) + 1$ and $tail\_Len(p, l) + 1 = |p_i|$. These pairs are added to $\mathbf{wait}[p_i]$ by the for-loop in lines 13–17 of the $(l+1)$th iteration. Then after this loop, for $k \in \mathcal{K}$, $\mathbf{wait}[k]$ is the

set of pairs $\langle p, i\rangle$ such that $i = last(p, l) + 1$, $p_i = k$ and $tail\_Len(p, l) + 1 \geqslant |k|$.

Therefore, we have $\mathbf{wait}[k] = WAIT(k, l+1)$, for every $k \in \mathcal{K}$. That is, condition 1 is true after stage 1 of the $(l+1)$th iteration.

According to the definition of AC automaton, after executing line 18, $L(active)$ is the longest prefix of $\mathcal{K}$ that is a suffix of $t[1 : l+1]$. So, condition 2 is satisfied.

Every *Mark* operation of a keyword k is followed by an inserting of the set $\mathbf{wait}[k]$ and every *UnMark* operation of k is preceded by the operations to empty the set $\mathbf{wait}[k]$. Since condition 1 is true, the marked keywords are the keywords whose **wait** sets are not empty. Therefore the set of marked keywords is $Cur\_K\_Set(l)$. Thus, condition 3 is verified.

So, the data structure resulting from stage 1 of the $(l+1)$th iteration satisfies conditions l–3. We summarize the result in the following theorem.

Theorem 1. *The algorithm $MATCH(t, \mathcal{P})$ is correct*; *it detects an occurrence of any pattern in $\mathcal{P}$ in t iff there is one.*

5. Complexity of the algorithm

Theorem 2. *The algorithm $MATCH(t, \mathcal{P})$ runs in time $O((|t| + \|\mathcal{P}\|)\log\kappa/\log\log\kappa)$.*

Proof. We first consider the time of the operations that are not on **wait** nor on **pool**.

According to [1], the total number of the state transitions in $MATCH(t, \mathcal{P})$ by line 18 is less than $2|t|$.

Consider the operations on marked nodes. The while-loop of the algorithm *MATCH* will be executed exactly $|t|$ times. After *active* is updated by line 18, *MATCH* enumerates all the marked nodes in the terminal chain of the new *active*. By the structure of [5], only the marked nodes in the terminal chain of *active* will be visited, and each visiting takes $O(\log\kappa/\log\log\kappa)$ time. Even though there may be no marked node in the terminal chain of *active* except R, it also needs $O(\log\kappa/\log\log\kappa)$ time to confirm the case. Thus, the time of the operations on marked nodes is at least $O(|t|\log\kappa/\log\log\kappa)$. The total times of the matching of waiting keywords in the run are not greater than $\|\mathcal{P}\|$, because (1) only the matching of marked keywords will be reported; (2) the total times of the keyword marking are not grater than the number of the pairs inserted into **pool**. And we have that any pair $\langle p, i\rangle$ will be inserted into **pool** at most once. Thus, the time of the operations on marked nodes is $O((|t| + \|\mathcal{P}\|)\log\kappa/\log\log\kappa)$.

Consider the operations on **wait** and **pool**. Let $\langle p, i\rangle$ be a pair such that $p \in \mathcal{P}$ and $2 \leqslant i \leqslant \|p\|$, where $\|p\|$ is the number of keywords in p. For every $\langle p, i\rangle$, it will be inserted into **pool** in an iteration iff $\langle p, i-1\rangle$ has been deleted from **pool** and been inserted in **wait** and p_{i-1} is matched in this iteration. That is, each $\langle p, i\rangle$ will be inserted into and deleted from **pool** at most one time. Hence, the times of the operation on **pool** are not greater than $2\|\mathcal{P}\|$.

All the pairs inserted into **wait** are from **pool**. So, the times of **wait**-inserting are not greater than that of **pool**-inserting. The times of deleting of **wait** are not greater than that of matching of waiting keywords. To sum up, the time spent on **wait** and **pool** is $O(\|\mathcal{P}\|)$.

For the callings of *Mark* are in one to one correspondence of the insert operations on **wait** (see lines 15–16), and each delete operation on $\mathbf{wait}[k]$ is corresponding to a calling of $UnMark(k)$ (see lines 20–28). So, the times of the calling of *Mark* and *UnMark* are not greater than $2\|\mathcal{P}\|$ which take time $O(\|\mathcal{P}\|\log\kappa/\log\log\kappa)$.

Summarizing the complexity of all the parts, we state that the algorithm $MATCH(t, \mathcal{P})$ runs in time $O((|t| + \|\mathcal{P}\|)\log\kappa/\log\log\kappa)$. □

6. Further research

The VLDC patterns can be viewed as a kind of special regular expressions. Efficient multi-regular expression matching is a central function needed by deep packet inspection, intrusion detection and virus scanning. The further research is to implement more regular expression functions, including: (1) The length constraint on variable length don't cares. (2) The content constraints on variable length don't cares. The goals of the algorithm are high speed and low memory costs, which are crucial for the applications mentioned above. The algorithm in [10] can deal with length constraints, but it needs the preprocessing of the text. The direct porting of our algorithm to these problems faces difficulties. All the prefix occurrences of patterns should be recorded, since the leftmost prefix occurrence may be discarded due to range constraints.

References

[1] A.V. Aho, M.J. Corasick, Efficient string matching: An aid to bibliographic search, Comm. ACM 18 (1975) 333–340.
[2] A. Amir, M. Farach, R. Idury, J.L. Poutré, A. Schäfer, Improved dynamic dictionary matching, Inform. and Comput. 119 (2) (June 1995) 258–282.
[3] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, J. Seiferas, The smallest automation recognizing the subwords of a text, Theoret. Comput. Sci. 40 (1985) 31–55.
[4] A. Blumer, J. Blumer, D. Haussler, R. McConnell, A. Ehrenfeucht, Complete inverted files for efficient text retrieval and analysis, J. ACM 34 (3) (1987) 578–595.
[5] Ho-Leung Chan, Wing-Kai Hon, Tak Wah Lam, Kunihiko Sadakane, Compressed indexes for dynamic text collections, ACM Trans. Algorithms 3 (2) (2007).
[6] M. Crochemore, Transducers and repetitions, Theoret. Comput. Sci. 45 (1986) 63–86.
[7] M. Fischer, M. Paterson, String matching and other products, in: R. Karp (Ed.), Proceedings of the 7th SIAM–AMS Complexity of Computation, 1974, pp. 113–125.
[8] Shunsuke Inenaga, Hideo Bannai, Ayumi Shinohara, Masayuki Takeda, Setsuo Arikawa, Discovering best variable-length-don't-care patterns, in: Proc. 5th International Conference on Discovery Science (DS '02), in: Lecture Notes in Comput. Sci., vol. 2534, Springer-Verlag, November 2002, pp. 86–97.
[9] Gregory Kucherov, Michaël Rusinowitch, Matching a set of strings with variable length don't cares, Theoret. Comput. Sci. 178 (1–2) (1997) 129–154.
[10] U. Manber, R. Baeza-Yates, An algorithm for string matching with a sequence of don't cares, Inform. Process. Lett. 37 (1991) 133–136.
[11] M. Sohel Rahman, Costas S. Iliopoulos, Pattern matching algorithms with don't cares, in: SOFSEM (2), 2007, pp. 116–126.
[12] Daniel Dominic Sleator, Robert Endre Tarjan, A data structure for dynamic trees, J. Comput. System Sci. 26 (3) (1983) 362–391.
[13] Clam AntiVirus at: http://www.clamav.net.
[14] Snort Intrusion Detection System at: http://www.snort.org.