Journal of Computers Vol. 29 No. 5, 2018, pp. 112-128
doi:10.3966/199115992018102905010

An Efficient Adaptive Architecture for Multi-pattern Matching

Zhan Peng, Yuping Wang*, Lijuan Hu, Zhenyan Jing

School of Computer Science and Technology, Xidian University, Xi'an, 710071, China
ywang@xidian.edu.cn

Received 7 June 2017; Revised 7 November 2017; Accepted 7 December 2017

Abstract. Multi-pattern matching is a core technique of many applications. However, many of the existing algorithms cannot efficiently deal with large pattern sets or pattern sets with very short patterns. To address these issues, in this paper, an adaptive architecture for multi-pattern matching (AAMPM), which is based on a new data structure called adaptive matching tree (AMT), is proposed. In particular, each tree node in AMT saves only some pattern fragments of the whole pattern set, and the inner data structure of each tree node is adaptively chosen according to the features of those pattern fragments. Due to this adaptivity, each tree node can take as little memory as possible, additionally, matching the text fragments with the pattern fragments in the tree nodes can be very efficient. Based on AMT, AAMPM adopts an effective approach to search patterns in the text string. The experimental results show that, AAMPM has a strong robustness on pattern sets with short patterns. Moreover, due to the good scalability of AMT, AAMPM offers an excellent support for large pattern sets.

Keywords: dictionary matching, pattern matching, string matching

1 Introduction

Pattern Matching (PM) has been historically one of the key problems of computer science, where the term "pattern" is referred to as the plain strings in this paper. According to the number of patterns to be matched at a time, PM technologies can be classified into the Single-Pattern Matching (SPM) and Multi-Pattern Matching (MPM), and this work mainly focuses on the latter.

The MPM is a primitive but important technique which is used in many applications such as Information Retrieval (IR), Intrusion Detection System (IDS), Anti-virus System (AVS), Bio-informatics, etc. In general, it seeks all locations in a given text that might match any of the patterns in a given pattern set. Efficient MPM algorithms scan the text only once and search for the potential matches in all patterns simultaneously. Traditionally, these algorithms include two major phases, i.e. the preprocessing phase and the matching phase. In particular, the preprocessing phase converts the target patterns in the pattern set into a suitable data structure, and then the matching phase scans and compares the given text with the data structures to find the occurrence positions of the target patterns. It is obvious that in the matching phase, the time/space efficiency of the data structure generated in the preprocessing phase can have a significant impact on the global performance of the MPM algorithm.

Preferably, given a matching task, the matching time of the adopted algorithm should mainly depend on the size of the text and the pattern set, rather than the intrinsic features of the patterns. Unfortunately, many of the existing algorithms, such as the classical Wu-Manber (WM) [1] algorithm and the more recent Pre-filter+AC [2] algorithm, are seriously affected by the length of the patterns, particularly, they cannot efficiently deal with the pattern sets containing very short patterns. What dramatically degrades the performance of these algorithms when encountering with short patterns is the poor robustness of the data structures used by them. Therefore, it is quite meaningful to design a kind of data structure that has a strong robustness on the lengths of the patterns and can further efficiently match various pattern sets.

Nowadays, the number of patterns to be matched becomes huge in some applications, e.g. the signature library of a modern IDS might contain millions of signatures (patterns). Unfortunately, many

* Corresponding Author

existing algorithms, such as the classical AC [3] algorithm and the recent hardware-based algorithms, often fail to process large pattern sets, which is mainly due to their data structure that might lead to the problem of lacking storage space as the number of patterns grows. Therefore, how to design efficient MPM algorithms for large-scale pattern matching has become a challenge in the big data context.

In this paper, we proposed an adaptive architecture for multi-pattern matching (AAMPM) which is able to effectively match the pattern set with various lengths of patterns and with large scale. Similar to the traditional approaches, AAMPM also consists of the preprocessing phase and the matching phase. In the preprocessing phase, the whole pattern set is partitioned gradually into several small parts, and the pattern fragments of each small part is stored in a separate node in an adaptive matching tree (AMT) for subsequent matching, which is a compact tree-like data structure. Then in the matching phase, each position of the text is compared with the nodes in the AMT from the root to find whether there are matched patterns. For any given pattern set, since the inner data structure of each tree node in AMT can be adaptively chosen according to the features of the pattern fragments (which makes the tree nodes as efficient and compact as possible), the whole AMT has a strong robustness for the given pattern set and the efficiency of the pattern matching hereafter can be improved. Moreover, owing to the space-efficiency and scalability of the AMT, AAMPM also provides excellent support for large pattern sets.

2 Related Work

Because of the importance of PM, several efficient algorithms have been proposed in the past decades. Well-known algorithms for PM include Knuth-Morris-Pratt (KMP) [4], Boyer-Moore (BM) [5], Wu-Manber (WM) [1], and Aho-Corasick (AC) [3]. The KMP and BM algorithms are appropriate for SPM, but not suitable for MPM. The AC algorithm is a generalization of the KMP algorithm by preprocessing the patterns and building a deterministic finite automaton (DFA) that can match multiple patterns simultaneously. It is the first algorithm for MPM that has a linear time complexity. However, the AC algorithm is alphabet-sensitive and requires a huge amount of memory to build the DFA, which limits its application in large-scale pattern matching.

To reduce the memory requirement of AC, Tuck et al. [6] proposed an algorithm called Bitmapped AC. This algorithm uses the bitmap data structure to replace the conventional child pointers in each state of DFA. In addition, it uses a path compression technique which can merge successive single-child states into one state. It is reported that, the compression optimization of Bitmapped AC results in a 50 times reduction in memory consumption over the original AC algorithm. However, since traversal from node to node requires checking a bit in a bitmap and then performing a sum up to 256 prior bits in the bitmap, the average performance of this algorithm is not high. Another algorithm to reduce the space requirement is the Compressed AC algorithm proposed by Bremeler-Barr et al. [7]. In addition to use a similar path compression method, this algorithm also develops a technique that is able to eliminate the leaf states. The experiment shows that Compressed AC can reduce at most 60% memory requirement of the traditional AC algorithm. But this method suffers from the similar problem, i.e. it works well only for a few pattern sets. Lee and Huang [2] proposed an algorithm called Pre-filter+AC. As the name shows, this algorithm consists of a prefilter and a verification engine. The function of the prefilter is to find the starting positions of potential pattern occurrences. Once a suspicious starting position is found, the verification engine confirms true pattern occurrence. The prefilter uses a bit vector, called “master bitmap”, with simple bitwise *and* and *shift* operations to accumulate query results, while the verification engine, which is a modification of the AC automaton, checks all candidate patterns simultaneously. However, as the authors said, the prefilter is suitable for patterns of moderate or large lengths. For short patterns, its performance degrades seriously. Similarly, Kandhan et al. [8] proposed an algorithm called sigMach which is also based on the prefilter-verification framework. This algorithm develops a cache-efficient *q*-gram index structure called sigTree which can filter out a large number of patterns that are impossible to be matched.

The WM algorithm is another efficient algorithm for MPM which is known to be faster than AC in practical matching. The WM algorithm is an adaptation of the BM algorithm to multiple patterns. It uses a similar “shift idea” of the BM algorithm, but the shift of WM is not based on a single character as the BM does, instead, it is based on a *B-gram* that is a character block consists of *B* characters (*B* is usually set to 2 or 3 in practice). In a typical search, its time complexity is $O(B \cdot n / lsp)$, where *lsp* is the length of the shortest pattern in the pattern set. Therefore, the performance of WM is seriously affected by *lsp*.

To improve the WM algorithm, Zhou et al. [9] proposed an algorithm called MDH. Instead of using the first *m* characters as the signature of each pattern, MDH adopts a heuristic strategy to select the optimum *m* consecutive characters as the signature of each pattern. Similarly, Zhan et al. [10] proposed a different way to search for the optimal signatures of the patterns, additionally, this algorithm designs an index structure to reduce the time for searching the candidate patterns in the hash table. These algorithms can improve the performance of WM for common patterns but they do not solve the inefficiency caused by short patterns. To address this problem, Zhang et al. [11] proposed the High Concurrence WM algorithm (HCWM), which separates the patterns whose length is less than 4 from the pattern set. For these short patterns, the algorithm establishes independent data structures and uses different matching routines. This method can reduce the effect of short patterns, since short patterns are matched independently and concurrently. Another algorithm for this problem is the L^{+1}-MWM proposed by Choi et al. [12], which minimizes the performance degradation caused by the short patterns by appending characters to these patterns. It is reported that, the L^{+1}-MWM improves the performance of WM by as much as 20% in average, moreover, when $lsp < 5$ the L^{+1}-MWM gains a 38.87% enhancement.

For large scale pattern matching, which becomes more and more important in big data context, Le and Prasanna [13] presented a memory-efficient architecture for large-scale string matching (MASM) based on a pipelined binary search tree. It uses a technique called "leaf-attaching" to compress the given pattern set. The compressed pattern set are then transformed to a pipelined binary tree which will be used in the matching phase. It is reported that, the MASM achieves a memory efficiency (defined as the ratio of the amount of the required storage in bytes and the size of the pattern set in number of characters) of 0.56 for the Rogets dictionary and 1.32 for the Snort rule set. Moraru and Andersen [14] also presented a memory-efficient and cache-optimized algorithm for large pattern sets. This algorithm builds upon the Rabin-Karp [15] SPM algorithm and incorporates a new feed-forward bloom filter which takes into account the memory hierarchy of modern computers. This algorithm is also well suited for implementation on GPUs which enables the matching to be done in parallel.

There are also studies that exploited the capacity of hardware to accelerate the proposed MPM engines. Agarwal and Polig [16] proposed a hardware architecture which enables high throughout for Information Extraction applications. Instead of using the common DFA based approach, this architecture employs a novel hashing based scheme. The DFA based approaches (such as AC) typically process one character every cycle, while the proposed hash based scheme can process a string token of several characters every cycle, thus achieves higher throughout than the DFA based approaches. Zhang et al. [17] proposed a GPU-based parallel algorithm G-PEBF which uses the Extended Bloom Filter (EBF). It divides the pattern set into *N* subsets where the lengths of the patterns in each subsets are the same. Then it constructs an EBF for each subset and uses *N* threads to simultaneously process the subsets in parallel on the GPU. The performance of G-PEFB highly relies on the number of threads used during matching, and it is also not applicable for pattern sets containing very short patterns.

Nowadays, some variants of the classical MPM problem have been proposed. Tomohiro et al. [18] introduced a variant of MPM, where the pattern set is given in a compressed form that can be represented by a straight line program (SLP). For a given SLP-compressed pattern set of size *n* and height *h*, which represents *m* patterns of total length *N,* they present an $O\ (n^2 log^N)$-size variant of AC automaton that recognizes all occurrences of the patterns in $O\ (h + m)$ running time per character. Khancome and Boonjing [19] proposed an algorithm for the dynamic MPM problem. This algorithm uses the inverted lists data structure to allow the pattern set to be updated dynamically in an optimal time. Amir et al. [20] introduced an algorithm for the gapped MPM problem, where each pattern in the pattern set is a sequence of sub-patterns separated by bounded sequences of do not cares. Neuburger and Sokol [21] presented the first efficient algorithm that operates in small space for the 2-dimensional MPM problem.

3 Notations

Let $\sum$ be an *alphabet* consisting of a finite number of character symbols. (In this paper we will only focus on the case that $\sum$ is the ASCII character set, i.e. $|\sum| = 256$ and each character requires one byte in memory.) Given the alphabet $\sum$, a *string* as well as its *substrings* over $\sum$ can be defined as follows:

Definition 1. A *string* over $\sum$ is a sequence consisting of a finite number of characters from $\sum$. A length-*n* string is represented by $S = c_1c_2\ldots c_n$, where the *i*-th character of *S* is: $S[i] = c_i \in \sum\ (1 \leq i \leq n)$. The length of *S* is denoted by $|\ S\ |$. A *substring* of *S* is a sequence consisting of any consecutive characters of *S*.

The substring of S, which starts at position i and has length of len, is denoted by $S\,[i,\ len]$.

Given a string S, its *prefix* and *suffix*, which are both special substrings of S, are defined as follows:

Definition 2. The length-*m prefix* of a string S, which is denoted by S^{+m}, is the substring that consists of the first m characters of S, that is: $S^{+m} = S\,[1,\ m]$. While the *suffix* of S, which starts at position $m + 1$, is the substring left after removing the length-*m* prefix of S. It is denoted by S^{-m}, that is: $S^{-m} = S\,[m+1, |\,S\,| - m]$.

Note that, a string S can be actually regarded as a prefix/suffix of itself, i.e. $S = S^{+|S|} = S^{-0}$. Finally, the MPM problem can be formally defined as follows:

Definition 3. Given a text string T and a pattern set $P = \{p_1, p_2, \ldots, p_k\}$, where T and $p_i\ (1 \le i \le k)$ are strings over $\sum$. The *multi-pattern matching* is to find all occurrences of every pattern of P in T, more formally, to find the result set $R = \{(i, p_j) \mid p_j \in P \text{ and } T\,[i, |\,p_j\,|] = p_j\}$.

4 The Adaptive Architecture for Multi-pattern Matching

The procedure of AAMPM consists of the preprocessing phase and the matching phase. In the preprocessing phase, the whole pattern set is transformed to a compact tree-like structure called Adaptive Matching Tree (AMT), in which each tree node is built adaptively to save the information of a small part of the pattern set. While in the matching phase, the built AMT is compared against the text string to search patterns in that string. The framework of constructing the AMT from the pattern set is introduced in Section 4.1, and using the AMT to search patterns in the text string is described in Section 4.2, finally, adaptively creating the tree nodes is presented in Section 4.3.

4.1 AMT Construction

In the preprocessing phase, the given pattern set will be transformed to an AMT. This transformation includes five major steps:

(1) Create a suffix set *SF*, and put each pattern of the pattern set into *SF*. Each pattern is regarded as a suffix of itself.

(2) Compute the length of the shortest suffix (*lss*) in *SF*. For each suffix in *SF*, remove its length-*lss* prefix, and all the removed prefixes are collected together to form a prefix set *PF*. Discard the repetitions in *PF*, and calculate $|PF|$ which refers to the number of distinct prefixes (*ndp*).

(3) Create a tree node t, which is actually a kind of index, to hold the prefixes in *PF*. Each prefix serves as a key in t. The structure of t is selected adaptively according to *ndp* and *lss* which have been computed in step 2.

(4) Divide *SF* by putting together its suffixes, whose removed prefixes are the same, to form a sub-suffix-set. Associate each sub-suffix-set with the corresponding prefix in the tree node t.

(5) For each newly created sub-suffix-set in step 3, repeat the same procedure from step 2.

Note that, since each pattern is a suffix of itself, the whole pattern set P can be also regarded as a suffix set *SF*. Fig. 1 shows an example of this transformation on a pattern set $P = \{p_1, p_2, \ldots, p_{13}\}$. First, since p_6 is the shortest pattern (suffix) in P, we have $lss = |\,p_6\,| = 2$. Then the length-2 prefix of each pattern (suffix) is removed, and the removed prefixes are formed a prefix set: $PF = \{p_1^{+2} = aa, p_2^{+2} = aa, \ldots, p_{13}^{+2} = cc\}$. After discarding the repetitions in *PF*, only 3 distinct prefixes left: $PF = \{aa, bb, cc\}$, thus, we have $ndp = |\,PF\,| = 3$.

Then a tree node, which is actually the *root* of AMT, is created to hold the prefixes in *PF*. Each prefix serves as a *key* in the root and has a pointer (initialized to *NULL*) to the (future) child node. As will be seen in Section 4.3, the structure of root is selected adaptively according to two parameters: $lss = 2$ and $ndp = 3$.

After creation of the root, the suffixes left in P are grouped based on their lost length-2 prefixes: those suffixes having the same length-2 prefixes removed are grouped together to form a sub-suffix-set. In order to reveal the correspondence between the sub-suffix-set and the lost prefix, the tuple (*node.key, sub-suffix-set*) is used: the suffixes in the *sub-suffix-set* have the same prefix *key* removed, while the *key* is held as a key in the newly created *node*. Based on this notation, there are three tuples created after grouping the suffixes left in P: $tp_1 = (root.aa, SF_1 = \{p_1^{-2}, p_2^{-2}, p_3^{-2}, p_4^{-2}, p_5^{-2}, p_7^{-2}\})$, $tp_2 = (root.bb, SF_2 = \{p_8^{-2}, p_9^{-2}, p_{10}^{-2}\})$ and $tp_3 = (root.cc, SF_3 = \{p_{11}^{-2}, p_{12}^{-2}, p_{13}^{-2}\})$. This indicates that P is

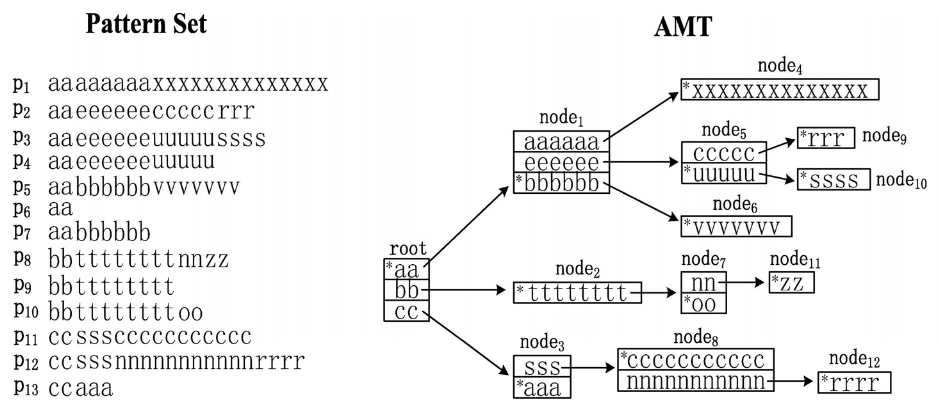

Fig. 1. Constructing the AMT from a pattern set, each arrow indicates a child pointer

divided into three sub-suffix-sets: SF1, SF2, SF3 and each sub-suffix-set corresponds to a prefix held as a key in the root (it also means that the root may have three child nodes). Note that, the shortest suffix (pattern) p6 has disappeared in the sub-suffix-sets, since after removing the length-2 prefix of p6, there is nothing left.

The same procedure is repeated on each of the three created sub-suffix-sets. As a further example, SF_1 of tp_1 is processed. First, the *lss* of SF_1 is computed by: $lss = |p_7^{-2}| = 6$. Then the length-6 prefix of each suffix in SF_1 is removed to form a prefix set: $PF_1 = \{(p_1^{-2})^{+6}, (p_2^{-2})^{+6}, (p_3^{-2})^{+6}, (p_4^{-2})^{+6}, (p_5^{-2})^{+6}, (p_7^{-2})^{+6}\}$. After discarding the duplicates in PF_1, only three distinct prefixes left: PF_1 = $\{a^6, e^6, b^6\}$ (we use the notation c^n to denote a length-n string consisting of the same character c). A new tree node $node_1$ is built adaptively upon PF_1 to hold its prefixes as keys. According to the first component of tp_1 (i.e. *root.aa*), $node_1$ is then associated with the key *aa* in the root by a child pointer, which makes it to be the first child of root. Once again, the suffixes left in SF_1 are grouped based on their lost length-6 prefixes, which formed three new tuples: $tp_4 = (node_1.a^6, SF_4 = \{p_1^{-8}\})$, $tp_5 = (node_1.e^6, SF_5 = \{p_2^{-8}, p_3^{-8}, p_4^{-8}\})$ and $tp_6 = (node_1.b^6, SF_6 = \{p_5^{-8}\})$.

Note that, in order to mark the end of a pattern, once the last part of the pattern has been removed and then stored in a node, that last part is marked with an asterisk in the corresponding node. As a result of this, any path from the root to a key that marked with an asterisk represents a complete pattern. It can be seen that, the patterns are stored implicitly in the AMT and can be reconstructed from the paths ended with an asterisk.

During the construction of AMT, we use a breadth-first strategy to process the sub-suffix-sets and create tree nodes (which means the next sub-suffix-set to be processed is SF_2 rather than SF_4). For the purpose of tracing the order in which the sub-suffix-sets are processed, a first-in-first-out queue is employed to maintain the created tuples. The tuple at the beginning of the queue contains the sub-suffix-set that will be processed next, while the newly created tuples are inserted to the end of the queue in order. In our example, the root node is built initially, then tp_1, tp_2 and tp_3 are inserted to the queue. Next, SF_1 of tp_1 is processed and the newly created tuples tp_4, tp_5 and tp_6 are inserted to the queue in order. Subsequently, $SF_1, SF_2, SF_3, SF_4, \ldots$ are processed in sequence. Once there is no tuple left in the queue, the whole AMT has been constructed. It is worth pointing out that, we can also use a depth-first strategy to build the AMT in a branch by branch way, but regardless of the strategies used, the final built AMTs are the same.

The framework of constructing the AMT from a pattern set is illustrated in Algorithm 1. Firstly, an empty queue *Q* is created in line 2 for maintaining the tuples. Then the tuple (*NULL*, *P*) is inserted to *Q* in line 3 by function *push_queue* which always inserts an element to the end of a queue. Since *P* is the initial pattern set here, there is no prefix removed from *P* yet, and this is indicated by the *NULL* component of the tuple. By this way, the creation of the root can be combined with the creation of other tree nodes in the following *while* loop.

Algorithm 1. Constructing the AMT

Input:

```
    The pattern set P
```

Output:

```
    The corresponding AMT
1:
2: Q ← Create an empty queue
3: push_queue((NULL, P), Q)
4:
5: while Q is not empty do
6:     (parent_node.key, SF) ← pop_queue(Q)
7:     lss ← |$suf_{st}$|, where $suf_{st}$ is the shortest suffix in SF
8:     for each suf ∈ SF do
9:         if |suf| = lss then
10:            Mark $suf^{+lss}$ to be the pattern end
11:        Remove $suf^{+lss}$ of suf, and put $suf^{+lss}$ into PF
12:    Discard the repetitions in PF, and let ndp ← |PF|
13:    According to lss and ndp, adaptively create a new_node to hold the
       prefixes in PF
14:    if (parent_node.key = NULL) then
15:        root ← new_node
16:    else
17:        Associate new_node with parent_node.key by a child pointer
18:        TP ← {(new_node.pf, SSF) | SSF ⊆ SF and ∀ p, q ∈ SSF: p, q
               have the same length-lss prefix pf removed}
19:    for each tp ∈ TP in order do
20:        push_queue(Q, tp)
21:
22: return root
```

If *Q* is not empty, the *while* loop body from line 5 to line 20 constructs a tree node based on the first tuple in *Q*. The tuple at the beginning of *Q* is fetched by the *pop_queue*(*Q*) function in line 6: the suffix set to be processed is assigned to *SF*, and the corresponding prefix of *SF* in the parent node is denoted by *parent_node.key*. Then the *lss* of *SF* is computed in line 7. In the inner *for* loop from line 8 to line 11, the length-*lss* prefix of each suffix in *SF* is removed, and the prefixes are collected to form the prefix set *PF*. If a prefix is the last part of some pattern, it is marked as a pattern end. Line 12 discards the repetitions in *PF*, after that a new tree node is created adaptively to hold the prefixes *PF* in line 13. The *if* statement in line 14 determines whether the newly created node is the root node or not. If the prefix component of the current tuple is *NULL*, the new node is the root node; otherwise, the new node is a child node which will be then associated with the corresponding prefix in its parent node. Next in line 18, the suffixes left in *SF* are grouped based on their lost prefixes to form tuples. Finally the created tuples are inserted to the queue in order. Once there is no tuple left in *Q*, the whole AMT has been completely constructed, and the root of AMT is returned.

4.2 Text Matching

In the matching phase, an algorithm based on the built AMT is used to search the patterns in the text string. The algorithm checks every position of the text to verify whether there is a potential pattern starting at that position. For each position *i* of the text, the algorithm executes a *matching round* to search the various sequential substrings of the text, where the first substring starts at *i*, in the corresponding nodes in AMT from the root. If some substring of the text successfully matched a key that is marked as a pattern end in some node, a pattern is declared to be found at position *i*. However, if there is a mismatch or the searching goes beyond the leaves of AMT, the current matching round terminates and the algorithm goes forward to the next position of the text and restarts a new matching round. Once every position of the text has been checked, the whole algorithm terminates.

The pseudocode of the matching phase of AAMPM is shown in Algorithm 2. Suppose *i* is the current matching position in the text. The *m_len* is the total length of the successfully matched sub-strings of the text in the current matching round. The *node* points to the current matching node in the AMT. These three variables are initialize to 1, 0 and the root of AMT respectively from line 2 to line 4. In addition, since all the keys in a node have the same length, the notation *key_len*(*node*) is used to denote the length of the keys in *node*.

Algorithm 2. Matching the text string

```
Input:
    The text string $T$
    The AMT built from the pattern set $P$
Output:
    The result set $R = \{(i, p) \mid 1 \le i \le |T|, p \in P: T[i, |p|] = p\}$
1:
2: $i \leftarrow 1$
3: $m\_len \leftarrow 0$
4: $node \leftarrow root$ of AMT
5:
6: while $i \le |T|$ do
7:   while $node \ne NULL$ and $\exists$ key$\in node$: $key = T[i+m\_len, key\_len(node)]$ do
8:       $m\_len \leftarrow m\_len + key\_len(node)$
9:       if $key$ is marked as a pattern end then
10:          Put $(i, T[i, m\_len])$ into $R$
11:      $node \leftarrow$ The child of $node.key$
12:   $i \leftarrow i + 1$
13:   $node \leftarrow root$
14:   $m\_len \leftarrow 0$
15:
16: return $R$
```

The major part of the pseudocode is a double *while* loop from line 6 to line 14. The outer *while* loop checks every position of the text. For each position *i*, the inner *while* loop performs a specific matching round to determine the potential patterns starting at *i*. The matching starts form the root of AMT: if the corresponding sub-string *T* [*i*+*m_len*, *key_len*(*node*)] matches some *key* in the current *node*, the totally matched length *m_len* is increased by the length of the key. At the same time, if the matched *key* is also marked as a pattern end, which means *T* [*i*, *m_len*] matches some pattern in the pattern set, this pattern as well as its starting position *i* are inserted into the result set *R*. Then the matching transfers to the child of *node.key* and makes that child node to be the current node. Once there is a mismatch or the current node goes beyond the leaves of AMT (i.e. *node* = *NULL*), the current matching round terminates. And the algorithm starts a new matching round for position *i*+1 after resetting the variables *node* and *m_len*. If all the positions of *T* have been checked, the whole algorithm terminates and returns the result set *R*.

Note that, since the nodes in AMT have various types of inner structures, the search of the target string in a node, must use the search routine specified to the type of that node.

Next, an example is given in Fig. 2 to illuminate a matching round at position *i* of a given text string *T*, using the AMT built in section 4.1. The matching round starts from the root of AMT: according to *key_len*(*root*) = 2, the same length sub-string starting at position *i* of *T*, which is *T* [*i*, 2] = *aa*, is taken to check whether it is one of the keys in root. The result is that: *aa*∈*root*, and also *aa* is marked as a pattern end. Therefore the pattern *T* [*i*, 2] = *aa* (actually p_6 in *P*) is found at position *i*. Then the matching transfers to $node_1$, which is the child node of *root.aa*.

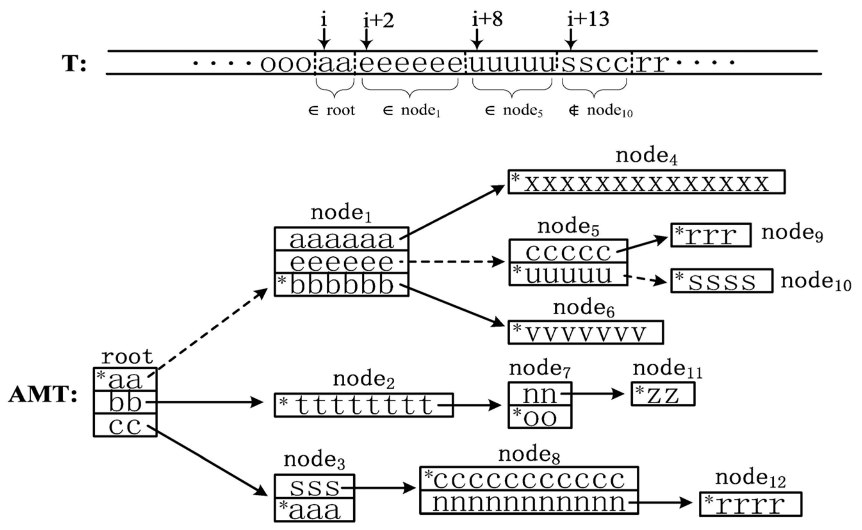

Fig. 2. The matching round that searches patterns starting at position *i* of the text

At $node_1$, since $key\_len(node_1) = 6$, the same length sub-string $T[i+2, 6] = e^6$, which appears following $T[i, 2]$, is trying to match with some key in $node_1$. The result is that: $e^6 \in node_1$ but it is not marked as a pattern end. Then the matching simply transfers to $node_5$, which is the child of $node_1.e^6$.

At $node_5$, since the next sub-string $T[i + 8, 5] = u^5 \in node_5$ and u^5 is marked as a pattern end, another pattern $T[i, 13] = a^2 e^6 u^5$ (actually p_4 in P) is found at position *i*. Then the matching goes to the child of $node_5.u^5$, i.e. $node_{10}$.

At $node_{10}$, since $T[i+13, 4] = sscc \notin node_{10}$, the current matching round for position *i* terminates. The algorithm restarts a new matching round for the next position $i + 1$. Once all the positions of the text have been checked, the whole algorithm terminates.

In our approach, each matching round actually corresponds to a *matching path* in the AMT, whose tree nodes have been compared with the text in the matching round. In particular, the matching path of the given example is: $root \rightarrow node_1 \rightarrow node_5 \rightarrow node_{10}$, which is indicated by the dashed arrows in Fig. 2.

4.3 Adaptive Creation of Tree Nodes

As stated in section 4.1, each node of AMT has a specific structure to hold the prefixes of a prefix set as the keys. The type of the node structure is chosen adaptively according to the features of the prefix set. Since all the keys in the prefix set have the same length, the features of the prefix set can be mainly characterized by two parameters: the *length* and *number* of the keys in the prefix set. These two parameters are respectively symbolized by *lss* and *ndp* in Section 4.1. In this work, three kinds of structures, i.e. *character map*, *string array* and *hash table* are adopted for prefix sets with different *lss* and *ndp* combinations. Next, the detailed descriptions of them will be presented.

Character Map. Given a prefix set, if the length of keys is equal to 1 ($lss = 1$), i.e. each key in the prefix set is just a single character, we adopt the space-efficient *character map* structure, which was proposed by Leis et al. [22], as the tree node. There are four sub-types of character maps with different capacities for different number of keys (*ndp*), where $1 \leq ndp \leq 256$.

Fig. 3 illustrates the four sub-types of character maps which are named according to their maximum capacity. Instead of using an array of (*key, child pointer*) pairs, the keys and the child pointers are stored in separate arrays, which is able to keep the node structure compact while supporting efficient search.

- *Map 4* ($1 \leq ndp \leq 4$): The smallest map type can store up to 4 keys. It uses an array of 4 entries for keys and another array of the same length for child pointers. The keys and child pointers are stored at corresponding positions in their arrays, and the keys are sorted according to their ACSII values. Once the target character is found in the key array, its child pointer can be located through the same position in the pointer array. Fig. 3(a) shows a Map 4 structure with three keys: *a*, *b* and *c*, where the triangles with the keys inside represent the corresponding child nodes.

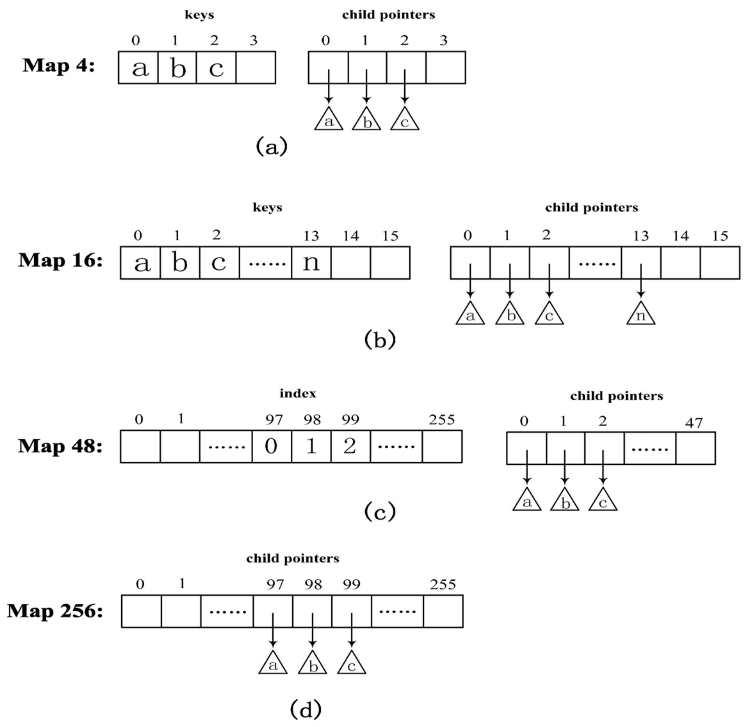

Fig. 3. The four sub-types of character maps, the triangles represent the corresponding child nodes

- *Map 16* (5 ≤ *ndp* ≤ 16): This map type is used for storing 5 to 16 keys. It has the similar structure as Map 4, but both arrays have up to 16 entries. A target character can be retrieved efficiently by a binary search in the key array. Fig. 3(b) shows a Map 16 structure with 14 keys: *a*~ *n*.
- *Map 48* (17 ≤ *ndp* ≤ 48): As the number of keys increases, searching in the key array becomes expensive. Therefore, maps with more than 16 (but less than 49) keys do not store the keys explicitly. Instead, a 256-element index array is used, which can be directly indexed by the ASCII value of the target character. This array stores only the array indexes (small inters in the range of [0, 47]) of another pointer array that contains up to 48 child pointers. In this way, the storage space can be saved comparing with storing pointers directly, because each array index only requires one byte. Fig. 3(c) shows a Map 48 structure, where the ASCII values of *a*, *b* and *c* are 97, 98 and 99 respectively.
- *Map 256* (49 ≤ *ndp* ≤ 256): The largest map type is simply an array of 256 child pointers with each pointer initialized to *NULL*. It is used for storing 49 to 256 keys. In this kind of character map, the child node can be found directly through the array index which is the ASCII value of the target character. Different from other sub-types of character maps, in Map 256, there is only one array and not necessary to carry out the additional indirect access. Therefore, if most entries are not empty, this representation is also very space efficient. Fig. 3(d) shows a Map 256 structure, where only the pointer array is needed.

For the sake of clarity, the pseudocode of searching in a node, whose type is character map, is given in Algorithm 3.

Algorithm 3. Searching in a node whose type is character map

Input:
```
    A tree node whose type is character map
    The target character t_ch.
```
Output:
```
    The child node of node.t_ch (possibly NULL).
1:
2: count ← Number of keys in node
3: switch (the sub-type of the node)
4:   case Map 4:
```

```
5:        for i ← 0 to count - 1 do
6:          if keys[i] = t_ch then
7:            return child_pointers[i].
8:        return NULL.
9:   case Map 16:
10:       low ← 0, high ← count - 1
11:       while low ≤ high do
12:         $mid \leftarrow \lfloor (low + high) / 2 \rfloor$
13:         if t_ch = keys[mid] then
14:            return child_pointers[mid]
15:         else if t_ch < keys[mid] then
16:            high ← mid - 1
17:         else
18:            low ← mid + 1
19:       return NULL
20:   case Map 48:
21:       if index[t_ch] ≠ NULL then
22:          return child_pointers[index[t_ch]]
23:       else
24:          return NULL
25:   case Map 256:
36:       return child_pointers[t_ch]
```

String Array. For the prefix set whose length of keys is greater than 1 (*lss* > 1) and the number of keys is not greater than 100 (*ndp* ≤ 100), the string array structure is constructed as a tree node. Similar to the character map, the keys are stored in lexicographical order in a separate key array of *ndp*×*lss* bytes, in which each key takes *lss* bytes. The child pointers are stored at the corresponding positions in another pointer array of 8×*ndp* bytes, where each pointer takes 8 bytes in a typical x86_64 architecture. Fig. 4 illustrates a string array structure holding three keys: *aaa*, *bbb* and *ccc*. The key and pointer arrays occupy 9 and 24 bytes respectively.

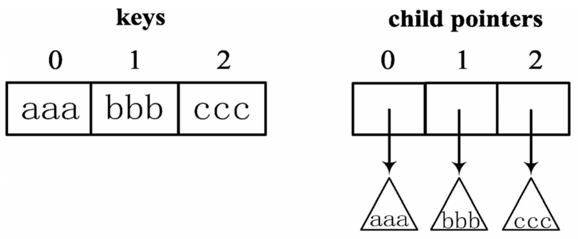

Fig. 4. The string array structure with three keys: *aaa, bbb, ccc*

Algorithm 4. Searching in a node whose type is string array

Input:

```
    A tree node whose type is string array;
    The target string t_str.
```

Output:

```
    The child node of node.t_str (possibly NULL).
1:
2:  count ← The number of keys in node
3:
4:  if count < 5 then
5:     for i  ← 1 to count - 1 do
6:       if  keys[i] = t_str then
```

```
7:          return child_pointers[i]
9:      return NULL
9:  else
10:     low ← 0, high ← count - 1
11:     while low ≤ high do
12:         $mid \leftarrow \lfloor (low + high)/2 \rfloor$
13:         if t_str = keys[mid] then
14:             return child_pointers[mid]
15:         else if t_str < keys[mid] then
16:             high ← mid - 1
17:         else
18:             low ← mid + 1
19:     return NULL
```

For efficiency, if the number of keys is less than 5, a naive linear search routine is used to search the target string in the key array; otherwise, a more efficient (but complicated) binary search routine is adopted. The pseudocode of the searching in a node whose type is string array is depicted in Algorithm 4. For the string array whose number of keys is less than 5, the target string is compared sequentially with the keys in the key array. If the target string matched some key, return the child pointer at the corresponding position in the pointer array; otherwise, return *NULL*. On the other hand, if the number of keys is greater than 4, the target string is searched by a binary search routine from the middle element of the key array.

Hash Table. For the prefix set whose number of keys is greater than 100 (*lss* > 1 and *ndp* > 100), searching in the string array structure becomes inefficient even using a binary search. In this case, a more fast (but heavy) data structure — *hash table* is adopted to deal with large number of keys. In this work, a hash table is an array of child pointers with each pointer initialized to *NULL*, and we utilize a *string hash function* which transforms a string to a positive integer. It is worth noting that, the procedure of building a hash table is directly based on the suffix set rather than the prefix set which is the basis of building other structures.

In particular, given a suffix set *SF*, the size of the hash table (denoted by *table_size*) is determined by *ndp* and a given *load factor lf* (ratio of *ndp* to *table_size*), i.e. table_size =$\lceil ndp/lf \rceil$. For example, with the *ndp* of 1000 and a load factor of 70%, the *table_size* is$\lceil 1000/0.7 \rceil = 1429$. Note that, the *ndp* here is defined to be the number of distinct length-*lss* prefixes of the suffixes in *SF*, which is equal to the size of the prefix set derived from *SF* (after removing the repetitions).

Algorithm 5 shows the pseudocode of building a hash table based on *SF*. *SF* is firstly partitioned into small suffix sets by hashing: for each suffix $suf \in SF$, its prefix suf^{+lss} is hashed to an integer *i* between 0 and *table_size* – 1 by the string hash function, then *suf* is associated with the *i*-th slot of the hash table. After that, the suffixes whose length-*lss* prefixes are hashed to the same value are associated with the same slot of the hash table. Then, to address the hash collisions, for each slot that is not *NULL*, the associated suffixes form a small suffix set *SSF*, and a new tree node is created adaptively based on the prefix set derived from *SSF*, as shown in the **while** loop of Algorithm 5. The newly created tree node is then associated with the corresponding slot of the hash table. As a result, many tree nodes with various types can be associated with one hash table.

Algorithm 5. Building a hash table

```
Input:
    The suffix set SF and the load factor lf.
Output:
    The hash table.
1:
2: ndp ← The number of distinct length-lss prefixes of suffixes in SF
3: $table\_size \leftarrow \lceil ndp/lf \rceil$
```

```
4: hash_table ← Create an array of table_size pointers with each
               pointer initialized to NULL
5:  for each suf ∈ SF do
6:      i ← Hash(suf^{+lss})
7:      Associate suf with hash_table[i]
8:
9:  for i ← 0 to table_size - 1 do
10:    if hash_table[i] ≠ NULL then
11:       SSF ← {suf | suf ∈ SF and Hash(suf^{+lss}) = i}
12:       new_node ← Adaptively create a tree node from SSF, as shown in
                     the while loop in Algorithm 1
13:       Associate new_node with hash_table[i]
14:
15: return hash_table
```

In general, for a large pattern set, the root of the corresponding AMT is usually a hash table, and the core function of the root can be regarded as a “filter”, which is able to filter out a great number of positions of the text that are impossible to match any pattern. Given a target string, the hash value of that string is computed and used as the index of the hash table. If the corresponding slot is *NULL*, which means the target string fails to match any key in the hash table, the current matching round terminates and the algorithm moves to the next position of the text; otherwise, the process moves to the tree node associated with that slot and compare the node with the corresponding substring in the text.

The string hash function adopted can have a significant effect on the performance of matching. In this work, for each matching task, the hash function is selected randomly from a hash function family *H*, at the beginning of the matching phase. The hash function family adopted in our implementation is the fast *shift-add-xor* hash family proposed by Ramakrishna [23], which is claimed to be both uniform and universal.

5 Experimental Results

In this section, we evaluate the performance of AAMPM, and compare it with other four algorithms: the AC [3] and WM [1] algorithms, which are two classic baseline algorithms for MPM; the MASM [13] and Pre-filter+AC [2] algorithms, which are two state-of-the-art MPM algorithms with good efficiency in practically matching. All these algorithms are evaluated in two aspects: the robustness of these algorithms under pattern sets with various *lsp*s and the scalability of the algorithms under pattern sets with various number of patterns.

5.1 Simulation Settings

The experiments are conducted on a PC with an Intel Core i7 2.93GHz CPU, 8GB of RAM and 1TB Disk Driver; the operating system is Ubuntu Linux 16.04 (64-bits). All the testing algorithms are implemented in C/C++ and complied by gcc with –O2 option. The test data used in the experiments is the real-world English text from the *Pizza & Chili corpus* (http://pizzachili.dcc.uchile.cl/). The patterns are extracted randomly from the text to form the pattern sets.

Next, the parameter configuration in AAMPM in our experiment is given as follows. For the String Arrays, if the number of strings exceeds *4*, the binary search is used to search the target string in the key array; otherwise, the linear search is adopted (for String Arrays containing no more than *4* strings, the simple linear search is faster than binary search due to the complexity of binary search). For the Hash Tables, the load factor is set to *0.5,* which is a good balance between time and space; the seed of the hash function is generated randomly in the range of *1~50*; the shift values *L* and *R* are set to *2* and *6* respectively (the parameters about the hash function are chosen from Ramakrishna [23] which are claimed to be very efficient). Once the *ndp* is larger than *100*, the Hash Table becomes more efficient than the String Array, thus the Hash Table is built once *ndp>100*. These choices of parameters have been extensively tested on a wide range of pattern sets, and are considered to be very efficient for a typical match.

Table 1. The characteristics of pattern sets with various LSPs

PS	LSP	LLP	ALP	SD	TLP	Count
P_1	2	50	25.9	14.1	2,599,020	10^5
P_2	3	50	26.5	13.9	2,646,762	10^5
P_3	4	50	27.0	13.6	2,704,091	10^5
P_4	5	50	27.6	13.2	2,745,545	10^5
P_5	6	50	27.9	13.0	2,793,178	10^5
P_6	7	50	28.4	12.7	2,843,580	10^5
P_7	8	50	29.0	12.4	2,903,343	10^5
P_8	9	50	29.5	12.1	2,948,372	10^5
P_9	10	50	30.0	11.8	2,998,992	10^5

5.2 Evaluation under Various *LSP*s

As we've mentioned before, many MPM algorithms are sensitive to the length of the patterns, particularly, the length of the shortest pattern (*lsp*) in the pattern set. Therefore, it is necessary to measure the performance of the AAMPM and compare it with the other investigated algorithms under pattern sets with various *lsp*s. In this evaluation, there are totally 9 pattern sets for testing with *lsp*s ranging from 2 to 10, and every pattern set has a fixed number of 10^5 patterns. The size of text string is fixed to 200 MB. The characteristics of the pattern sets (PS) are shown in Table 1, which includes: the length of the shortest pattern (LSP), the length of the longest pattern (LLP), the average length of the patterns (ALP), the standard deviation of the pattern lengths (SD), the total length of the patterns (TLP) and the number of patterns in the pattern set (Count).

Moreover, the performance of AAMPM is stable and reliable. In particular, once the length of the text and number of patterns are fixed, the matching time changes little for various *lsp*s. As shown in Table. 2, the matching time of AAMPM is always about 10 seconds for a text of 200MB and a pattern set with 10^5 patterns; for *lsp*s ranging from 2 to 10, the change in the matching time is less than 2 seconds. In fact, as the *lsp* increases, the average length of the paths from the root to the leaves in AMT will increase slightly. Accordingly, in the matching phase, the matching paths for some text positions may get a little longer, which might slightly slow down the matching speed. The statistics about node types of corresponding AMTs are shown in Table 3. The Map 1 and Single String Array types in the table are actually the Map 4 and String Array types which contain only one element, respectively. Obviously, as the *lsp* of the pattern set changes, the number of nodes of the each type in the AMT changes accordingly, which reflects the adaptivity of AMT.

Table 2. Matching times for various LSPs (seconds)

LSP	AAMPM	MASM	Pre-filter+AC	WM	AC
2	9.20	16.04	43.26	79.30	60.26
3	9.25	16.99	26.08	44.08	55.08
4	9.65	18.01	19.41	31.00	56.41
5	9.99	19.45	15.97	28.87	53.62
6	10.56	19.08	14.78	29.50	52.25
7	10.03	20.45	14.01	29.09	51.68
8	10.65	20.99	13.54	28.99	51.28
9	10.54	21.59	13.23	28.67	51.12
10	10.99	21.89	13.21	28.40	51.23

Table 3. The statistics about the node type in the AMTs under various LSPs

LSP	Map 1	Map 4	Map 16	Map 48	Map 256	Single String Array	String Array	Hash Table	Total
2	2,464	2,096	663	133	0	63,274	21,627	10	90,267
3	2,379	1,636	385	90	0	64,701	22,551	10	91,752
4	2,273	1,329	201	57	0	66,162	22,736	1	92,759
5	2,045	1,104	112	45	0	68,152	22,048	1	93,507
6	1,758	899	52	34	0	70,179	20,818	1	93,741
7	1,560	807	29	33	0	72,284	19,280	1	93,994
8	1,515	803	20	31	0	73,447	18,274	1	94,091
9	1,441	788	24	30	0	74,584	17,120	1	93,988
10	1,361	794	24	29	0	75,094	16,427	1	93,730

5.3 Evaluation under Various Numbers of Patterns

In this section, we evaluate the scalability of the investigated algorithms by testing them under pattern sets with various numbers of patterns. As before, the size of the text is fixed to 200MB, and there are two groups of pattern sets for testing — the small group and the large group. The small group contains 9 pattern sets, with the sizes (number of patterns) increasing from 1×10^5 to 9×10^5 in steps of 10^5, while the large group contains 10 pattern sets, with the sizes increasing from 10^6 to 10^7 in steps of 10^6. The range of the pattern length of each pattern set is fixed to 5 ~ 50.

For the small group, the mean matching times of the investigated algorithms under 5 independent runs are shown in Table 4. The experimental results indicate that, the performance of the AC algorithm is relative stable (but not high) as the number of patterns growing, but it cannot deal with pattern sets whose size is larger than 7×10^5 in this testing due to lack of memory. Although the WM algorithm performs better than AC, it is less stable, for instance, the matching time only rises about 1*s* as the number of patterns increasing from 2×10^5 to 3×10^5, but when the number of patterns increases from 8×10^5 to 9×10^5, the matching time rises roughly up to 9*s*. Thus, we cannot estimate the matching time of WM based on the size of the pattern set. The Pre-filter+AC performs well as the number of patterns growing, but its matching time is also unpredictable, e.g. the matching times are nearly the same as the number of patterns increasing from 4×10^5 to 5×10^5. The instability of WM and Pre-filter+AC is mainly due to that: the effect of the filters used by these two algorithms highly depends on the contents of the text and patterns themselves, particularly, for some pattern sets whose patterns occurs frequently in the text, the effect of the filters degrades. On the other hand, as the number of patterns grows, the matching time of MASM grows more slowly than that of the Pre-filter+AC, which is due to that the matching time of MASM mainly relies on the depth of the pipelined binary search tree and that depth grows very slow as the number of patterns increases.

Table 4. Matching times for the small group (seconds)

Size	AAMPM	MASM	Pre-filter+AC	WM	AC
1×10^5	13.94	24.94	18.26	30.30	63.26
2×10^5	16.65	27.71	21.08	33.08	65.08
3×10^5	17.65	28.20	21.94	33.00	64.41
4×10^5	19.99	30.53	25.32	38.50	65.62
5×10^5	21.69	32.98	25.45	39.39	67.25
6×10^5	22.54	33.27	28.30	44.78	69.68
7×10^5	23.78	33.79	29.88	46.98	70.28
8×10^5	24.82	36.88	33.09	47.79	—
9×10^5	25.32	37.54	34.51	55.39	—

Table 5. The statistics about the node type in the AMTs built for the small group

Size	Map 1	Map 4	Map 16	Map 48	Map 256	Single String Array	String Array	Hash Table	Total
1×10^5	2,000	1,006	72	18	0	68,268	21,971	103	93,438
2×10^5	4,628	2,808	307	42	0	130,440	46,514	244	184,983
3×10^5	7,414	4,858	574	84	0	190,653	71,757	403	275,746
4×10^5	10,555	7,282	112	139	0	247,656	97,967	544	365,089
5×10^5	13,703	10,651	946	238	4	302,350	124,528	17	452,994
6×10^5	17,335	13,536	1,503	298	0	356,644	151,435	31	531,233
7×10^5	21,080	16,872	1,954	388	4	408,165	179,621	35	628,570
8×10^5	24,357	20,094	2,405	436	3	457,674	208,426	37	714,004
9×10^5	28,356	23,924	2,977	513	9	504,779	238,632	42	799,696

Among all these algorithms, the AAMPM is the most efficient as well as stable one. The high performance of AAMPM is mainly due to the *root* node of AMT, which is almost always the *hash table* structure as the number of patterns grows. As mentioned before, the root plays the role of a "filter", by which a large number of positions of the text that are impossible to match with any pattern can be quickly filtered out. Moreover, the increasing in the number of patterns mainly leads to the growing of the width rather than the depth of AMT. Therefore, for each position of the text, the checking time changes slightly. From the experimental results, we can even give a rough estimate on the matching time based on the size of pattern set as follows: given the text of 200 MB, an increase of 10^5 in the number of patterns will lead to roughly one more second taken in the matching time. The statistics about the node types of the corresponding AMTs for the small group are shown in Table 5. We can see that an increase of 10^5 in the number of patterns will yield about 9×10^4 tree nodes.

For the large group, the mean matching times of the investigated algorithms are shown in Table 6. From the results we can see that, as the number of patterns grows from 10^6 to 10^7, the increases in the matching time of the investigated algorithms are respectively: 408% (WM), 371% (Pre-filter+AC), 233% (MASM) and 117% (AAMPM). For every increase of 10^6 in the number of patterns, the mean increment in the matching time of the investigated algorithms are: 22.5*s* (WM), 16.5*s* (Pre-filter+AC), 11.8*s* (MASM), 3.9*s* (AAMPM) respectively. From the results we can see that, Pre-filter+AC and WM perform not well for large pattern sets. This is because, as the number of patterns grows, the chances for every position of the text to successfully match a pattern increases, which will reduce the effect of the filters of these algorithms and results in performance degradation. The MASM algorithm outperforms Pre-filter+AC for pattern sets whose size are larger than 3×10^6, which is due to its pattern set compression strategy and the good scalability of its pipelinded binary search tree.

Table 6. Matching times for the large group (seconds)

Size	AAMPM	MASM	Pre-filter+AC	WM
1×10^6	28.20	45.21	37.26	57.21
2×10^6	33.25	56.33	48.08	70.33
3×10^6	34.65	68.78	62.41	105.27
4×10^6	40.99	80.98	90.62	120.33
5×10^6	44.26	90.99	100.25	140.40
6×10^6	50.73	106.76	124.68	180.89
7×10^6	53.65	113.76	134.28	198.24
8×10^6	54.52	123.97	150.12	227.65
9×10^6	58.99	140.51	172.00	243.54
1×10^7	59.78	152.56	190.12	270.33

On the other hand, among all the algorithms, AAMPM is the most efficient algorithm for large pattern sets. An increase of 10^6 in the number of patterns, only require 3 more seconds to match, this is mainly due to the good scalability of the AMT which enables the matching time grows very slow as the number of patterns explodes. The statistics about the AMTs for the large group are shown in Table 7. In particular, every 10^6 increase in the number of patterns will yield only about 8×10^5 tree nodes.

Table 7. The statistics about the node type in the AMTs built for the large group

Size	Map 1	Map 4	Map 16	Map 48	Map 256	Single String Array	String Array	Hash Table	Total
1×10^6	38,935	27,428	6,468	851	4	588,3437	3,986,683	1,027	903,942
2×10^6	85,900	68,973	17,084	2,141	20	1,091,527	496,001	2,015	1,763,661
3×10^6	136,054	117,527	28,092	3,410	17	1,547,659	764,359	2,749	2,599,867
4×10^6	189,926	171,155	40,328	4,716	23	1,966,326	1,045,558	3,450	3,421,482
5×10^6	247,707	230,163	83,043	5,989	38	2,354,308	1,332,896	3,885	4,228,029
6×10^6	304,863	293,591	65,475	7,169	49	2,706,094	1,625,261	4,379	5,006,881
7×10^6	366,861	361,695	77,899	8,434	53	3,030,673	1,925,027	4,733	5,775,380
8×10^6	429,121	433,708	90,280	9,660	62	3,336,765	2,226,879	5,070	6,531,545
9×10^6	494,278	509,992	102,402	10,833	72	3,623,407	2,537,413	5,224	7,283,621
1×10^7	558,241	591,455	115,012	11,922	79	3,986,683	2,847,277	5,505	8,026,174

6 Conclusion and Future Work

In this paper, we propose an Adaptive Architecture for Multi-Pattern Matching (AAMPM). The AAMPM used a data structure called Adaptive Matching Tree (AMT) which can be constructed to fit the feature of the given pattern set. Using the AMT to match with the text can improve the robustness of AAMPM for pattern sets with various *lsp*s. Moreover, owing to the scalability and compactness of AMT, AAMPM offers a good support for large-scale pattern sets. In the future, we will try to design more efficient inner structures of the tree nodes; furthermore, since AAMPM needs to check every position of the text string, we will further improve this by designing a “skip scheme” for AAMPM.

Acknowledgments

This work is supported by National Natural Science Foundation of China (No. 61472297).

References

[1] S. Wu, U. Manber, A fast algorithm for multi-pattern searching. <http://webglimpse.net/pubs/TR94-17.pdf>, 1994.

[2] T.-H. Lee, N.-L. Huang, A pattern-matching scheme with high throughput performance and low memory requirement, IEEE/ACM Transactions on Networking 21(4)(2013) 1104-1116.

[3] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Commun. ACM 18(6)(1975) 333-340.

[4] D.E. Knuth, Jr., J.H. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM Journal on Computing 6(2)(1977) 323-350.

[5] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Commun. ACM 20(10)(1977) 762-772.

[6] N. Tuck, T. Sherwood, B. Calder, G. Varghese, Deterministic memory-efficient stringmatching algorithms for intrusion detection, in: Proc. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies, 2004.

[7] A. Bremler-Barr, Y. Harchol, D. Hay, Space-time tradeoffs in software-based deep packet inspection, in: Proc. IEEE 12th International Conference on High Performance Switching and Routing (HPSR), 2011.

[8] R. Kandhan, N. Teletia, J.M. Patel, Sigmatch: fast and scalable multi-pattern matching, Proceedings of the VLDB Endowment 3(1)(2010) 1173-1184.

[9] Z. Zhou, Y. Xue, J. Liu, W. Zhang, J. Li, MDH: a high speed multi-phase dynamic hash string matching algorithm for large-scale pattern set, in: Proc. Information and Communications Security Springer, 2007.

[10] P. Zhan, W. Yuping, X. Jinfeng, An improved multi-pattern matching algorithm for large-scale pattern sets, in: Proc. Tenth International Conference on Computational Intelligence and Security (CIS), 2014.

[11] B. Zhang, X. Chen, X. Pan, Z. Wu, High concurrence wu-manber multiple patterns matching algorithm, in: Proc. the

International Symposium on Information Process, 2009.

[12] Y.-H. Choi, M.-Y. Jung, S.-W. Seo, A fast pattern matching algorithm with multi-byte search unit for highspeed network security, Computer Communications 34(14)(2011) 1750-1763.

[13] H. Le, V.K. Prasanna, A memory-efficient and modular approach for large-scale string pattern matching, IEEE Transactions on Computers 62(5)(2013) 844-857.

[14] I. Moraru, D.G. Andersen, Exact pattern matching with feed-forward bloom filters, Journal of Experimental Algorithmics 17(2013) 3-4.

[15] R.M. Karp, M.O. Rabin, Efficient randomized patternmatching algorithms, IBM Journal of Research and Development 31(2)(1987) 249-260.

[16] K. Agarwal, R. Polig, A high-speed and large-scale dictionary matching engine for information extraction systems, in: IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2013.

[17] H. Zhang, D. Xu, Z. Tian, Y. Fan, An efficient parallel algorithm for exact multi-pattern matching, Security And Communication Networks 8(9)(2015) 1688-1697.

[18] I. Tomohiro, T. Nishimoto, S. Inenaga, H. Bannai, M. Takeda, Compressed automata for dictionary matching, Theoretical Computer Science 578(2015) 30-41.

[19] C. Khancome, V. Boonjing, A new linear-time dynamic dictionary matching algorithm, Computing And Informatics 32(5)(2013) 897-923.

[20] A. Amir, A. Levy, E. Porat, B.R. Shalom, Dictionary matching with a few gaps, Theoretical Computer Science 589(2015) 34-46.

[21] S. Neuburger, D. Sokol, Succinct 2d dictionary matching, Algorithmica 65(3)(2012) 662-684.

[22] V. Leis, A. Kemper, T. Neumann, The adaptive radix tree: artful indexing for main-memory databases, in: Proc. IEEE 29th International Conference on Data Engineering (ICDE), 2013.

[23] M.V. Ramakrishna, J. Zobel, Performance in practice of string hashing functions, in: Proc. the Fifth International Conference on Database Systems for Advanced Applications (DASFAA), 1997.