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 \(O(B^n / lsp)\), 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 $L^{11}$-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 $L^{11}$-MWM improves the performance of WM by as much as 20% in average, moreover, when $lsp < 5$ the $L^{11}$-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 Roget’s 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-PEBF 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 $O(n^2 \log^2)$-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 $\Sigma$ be an alphabet consisting of a finite number of character symbols. (In this paper we will only focus on the case that $\Sigma$ is the ASCII character set, i.e. $|\Sigma| = 256$ and each character requires one byte in memory.) Given the alphabet $\Sigma$, a string as well as its substrings over $\Sigma$ can be defined as follows:

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

---

An Efficient Adaptive Architecture for Multi-pattern Matching
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 $P = \{p_1, p_2, \ldots, p_i\}$, where $T$ and $p_i (1 \leq i \leq k)$ are strings over $Σ$. The multi-pattern matching is to find all occurrences of every pattern of $P$ in $T$, more formally, to find the result set $R = \{(i, p) | p \in P \text{ and } T[i, |p|] = p_i\}$.

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, SF1 of tp1 is processed. First, the lss of SF1 is computed by: lss = |p1^2| = 6. Then the length-6 prefix of each suffix in SF1 is removed to form a prefix set: PF1 = \{a^6, e^6, b^6\}. After discarding the duplicates in PF1, only three distinct prefixes left: PF1 = \{a, e, b\} (we use the notation cn to denote a length-n string consisting of the same character c). A new tree node node1 is built adaptively upon PF1 to hold its prefixes as keys. According to the first component of tp1 (i.e. root.aa), node1 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 SF1 are grouped based on their lost length-6 prefixes, which formed three new tuples: tp4 = (node1, a^6, SF4 = \{p1^8\}), tp5 = (node1, e^6, SF5 = \{p2^8, p3^8, p4^8\}) and tp6 = (node1, b^6, SF6 = \{p5^8\}).

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 SF2 rather than SF3). 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 tp1, tp2 and tp3 are inserted to the queue. Next, SF1 of tp1 is processed and the newly created tuples tp4, tp5 and tp6 are inserted to the queue in order. Subsequently, SF1, SF2, SF3, SF4, ... 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 the 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: $Q \leftarrow$ Create an empty queue
2: $push\_queue((NULL, P), Q)$
3:
4: while $Q$ is not empty do
5:   $(parent\_node.key, SF) \leftarrow pop\_queue(Q)$
6:   $lss \leftarrow |suf_{st}|$, where $suf_{st}$ is the shortest suffix in $SF$
7:   for each $suf \in SF$ do
8:     if $|suf| = lss$ then
9:        Mark $suf^{+lss}$ to be the pattern end
10:       Remove $suf^{+lss}$ of $suf$, and put $suf^{+lss}$ into $PF$
11:       Discard the repetitions in $PF$, and let $ndp \leftarrow |PF|$
12:   According to $lss$ and $ndp$, adaptively create a $new\_node$ to hold the
13:   prefixes in $PF$
14:   if $(parent\_node.key = NULL)$ then
15:      root $\leftarrow new\_node$
16:   else
17:      Associate $new\_node$ with $parent\_node.key$ by a child pointer
18:      $TP \leftarrow \{(new\_node.pf, SSF) \mid SSF \subseteq SF \text{ and } \forall \ p, q \in SSF: \ p, q$
19:         have the same length-$lss$ prefix $pf$ removed}$
20:      for each $tp \in TP$ in order do
21:         $push\_queue(Q, tp)$
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 AAMP 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 \( \text{node} \) 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 \( \text{key_len}(\text{node}) \) is used to denote the length of the keys in \( \text{node} \).

**Algorithm 2.** Matching the text string

**Input:**
- The text string \( T \)
- The AMT built from the pattern set \( P \)

**Output:**
- The result set \( R = \{(i, p) | 1 \leq i \leq |T|, \ p \in P: \ T[i, |p|] = p\} \)

1: \( i \leftarrow 1 \)
2: \( m_{len} \leftarrow 0 \)
3: \( \text{node} \leftarrow \text{root of AMT} \)
4: \( \text{while} \ i \leq |T| \text{ do} \)
5: \( \text{while} \ \text{node} \neq \text{NULL} \text{ and } \exists \ \text{key} \in \text{node}: \ \text{key} = T[i+m_{len}, \ \text{key_len}(\text{node})] \text{ do} \)
6: \( m_{len} \leftarrow m_{len} + \text{key_len}(\text{node}) \)
7: \( \text{if} \ \text{key} \text{ is marked as a pattern end then} \)
8: \( \text{Put } (i, T[i, m_{len}]) \text{ into } R \)
9: \( \text{node} \leftarrow \text{The child of node.key} \)
10: \( i \leftarrow i + 1 \)
11: \( \text{node} \leftarrow \text{root} \)
12: \( m_{len} \leftarrow 0 \)
13: \( \text{return } R \)

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 \( T[i+m_{len}, \ \text{key_len}(\text{node})] \) matches some \( \text{key} \) in the current \( \text{node} \), the totally matched length \( m_{len} \) is increased by the length of the key. At the same time, if the matched \( \text{key} \) 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 \( \text{node.key} \) 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. \( \text{node} = \text{NULL} \)), the current matching round terminates. And the algorithm starts a new matching round for position \( i+1 \) after resetting the variables \( \text{node} \) 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 \( \text{key_len(root)} = 2 \), 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 the root. The result is that: \( aa \in \text{root} \), 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 \( \text{node}_1 \), which is the child node of \( \text{root.aa} \).
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 $T[i+13, 4] = sccc \notin node_10$, 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 ASCII 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 ints 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: count ← Number of keys in node
2: switch (the sub-type of the node)
3:     case Map 4:
for i ← 0 to count − 1 do
    if keys[i] = t_ch then
        return child_pointers[i].

return NULL.

case Map 16:
    low ← 0, high ← count − 1

    while low ≤ high do
        mid ← ⌊(low + high)/2⌋
        if t_ch = keys[mid] then
            return child_pointers[mid]
        else if t_ch < keys[mid] then
            high ← mid − 1
        else
            low ← mid + 1

    return NULL

case Map 48:
    if index[t_ch] ≠ NULL then
        return child_pointers[index[t_ch]]
    else
        return NULL

case Map 256:
    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](image)

### 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: count ← The number of keys in node

4: if count < 5 then
5:     for i ← 1 to count − 1 do
6:         if keys[i] = t_str then
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 = \( \left\lceil \frac{ndp}{lf} \right\rceil \). For example, with the ndp of 1000 and a load factor of 70%, the table_size is \( \left\lceil \frac{1000}{0.7} \right\rceil = 1429 \). 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 ∈ 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: ndp ← The number of distinct length-lss prefixes of suffixes in SF
2: table_size ← \( \left\lceil \frac{ndp}{lf} \right\rceil \)
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 + \text{lss})
7:     Associate suf with hash_table[i]
8: for i ← 0 to table_size – 1 do
9:     if hash_table[i] ≠ NULL then
10:        SSF ← \{suf | suf ∈ SF and Hash(suf + \text{lss}) = i\}
11:        new_node ← Adaptively create a tree node from SSF, as shown in the while loop in Algorithm 1
12:        Associate new_node with hash_table[i]
13: 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 \( \text{lsp} \)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

<table>
<thead>
<tr>
<th>PS</th>
<th>LSP</th>
<th>LLP</th>
<th>ALP</th>
<th>SD</th>
<th>TLP</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>50</td>
<td>25.9</td>
<td>14.1</td>
<td>2,599,020</td>
<td>10⁵</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>50</td>
<td>26.5</td>
<td>13.9</td>
<td>2,646,762</td>
<td>10⁵</td>
</tr>
<tr>
<td>P3</td>
<td>4</td>
<td>50</td>
<td>27.0</td>
<td>13.6</td>
<td>2,704,091</td>
<td>10⁵</td>
</tr>
<tr>
<td>P4</td>
<td>5</td>
<td>50</td>
<td>27.6</td>
<td>13.2</td>
<td>2,745,545</td>
<td>10⁵</td>
</tr>
<tr>
<td>P5</td>
<td>6</td>
<td>50</td>
<td>27.9</td>
<td>13.0</td>
<td>2,793,178</td>
<td>10⁵</td>
</tr>
<tr>
<td>P6</td>
<td>7</td>
<td>50</td>
<td>28.4</td>
<td>12.7</td>
<td>2,843,580</td>
<td>10⁵</td>
</tr>
<tr>
<td>P7</td>
<td>8</td>
<td>50</td>
<td>29.0</td>
<td>12.4</td>
<td>2,903,343</td>
<td>10⁵</td>
</tr>
<tr>
<td>P8</td>
<td>9</td>
<td>50</td>
<td>29.5</td>
<td>12.1</td>
<td>2,949,372</td>
<td>10⁵</td>
</tr>
<tr>
<td>P9</td>
<td>10</td>
<td>50</td>
<td>30.0</td>
<td>11.8</td>
<td>2,998,992</td>
<td>10⁵</td>
</tr>
</tbody>
</table>

5.2 Evaluation under Various LSPs

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. In this evaluation, there are totally 9 pattern sets for testing with lsp ranging from 2 to 10, and every pattern set has a fixed number of 10⁵ 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. 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 10⁵ patterns; for lsp 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)

<table>
<thead>
<tr>
<th>LSP</th>
<th>AAMPM</th>
<th>MASM</th>
<th>Pre-filter+AC</th>
<th>WM</th>
<th>AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>9.20</td>
<td>16.04</td>
<td>43.26</td>
<td>79.30</td>
<td>60.26</td>
</tr>
<tr>
<td>3</td>
<td>9.25</td>
<td>16.99</td>
<td>26.08</td>
<td>44.08</td>
<td>55.08</td>
</tr>
<tr>
<td>4</td>
<td>9.65</td>
<td>18.01</td>
<td>19.41</td>
<td>31.00</td>
<td>56.41</td>
</tr>
<tr>
<td>5</td>
<td>9.99</td>
<td>19.45</td>
<td>15.97</td>
<td>28.87</td>
<td>53.62</td>
</tr>
<tr>
<td>6</td>
<td>10.56</td>
<td>19.08</td>
<td>14.78</td>
<td>29.50</td>
<td>52.25</td>
</tr>
<tr>
<td>7</td>
<td>10.03</td>
<td>20.45</td>
<td>14.01</td>
<td>29.09</td>
<td>51.68</td>
</tr>
<tr>
<td>8</td>
<td>10.65</td>
<td>20.99</td>
<td>13.54</td>
<td>28.99</td>
<td>51.28</td>
</tr>
<tr>
<td>10</td>
<td>10.99</td>
<td>21.89</td>
<td>13.21</td>
<td>28.40</td>
<td>51.23</td>
</tr>
</tbody>
</table>
Table 3. The statistics about the node type in the AMTs under various LSPs

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

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 $1 \times 10^5$ to $9 \times 10^5$ 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 $7 \times 10^5$ 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 1s as the number of patterns increasing from $2 \times 10^5$ to $3 \times 10^5$, but when the number of patterns increases from $8 \times 10^5$ to $9 \times 10^5$, the matching time rises roughly up to 9s. 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 $4 \times 10^5$ to $5 \times 10^5$. 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)

<table>
<thead>
<tr>
<th>Size</th>
<th>AAMPM</th>
<th>MASM</th>
<th>Pre-filter+AC</th>
<th>WM</th>
<th>AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \times 10^5$</td>
<td>13.94</td>
<td>24.94</td>
<td>18.26</td>
<td>30.30</td>
<td>63.26</td>
</tr>
<tr>
<td>$2 \times 10^5$</td>
<td>16.65</td>
<td>27.71</td>
<td>21.08</td>
<td>33.08</td>
<td>65.08</td>
</tr>
<tr>
<td>$3 \times 10^5$</td>
<td>17.65</td>
<td>28.20</td>
<td>21.94</td>
<td>33.00</td>
<td>64.41</td>
</tr>
<tr>
<td>$4 \times 10^5$</td>
<td>19.99</td>
<td>30.53</td>
<td>25.32</td>
<td>38.50</td>
<td>65.62</td>
</tr>
<tr>
<td>$5 \times 10^5$</td>
<td>21.69</td>
<td>32.98</td>
<td>25.45</td>
<td>39.39</td>
<td>67.25</td>
</tr>
<tr>
<td>$6 \times 10^5$</td>
<td>22.54</td>
<td>33.27</td>
<td>28.30</td>
<td>44.78</td>
<td>69.68</td>
</tr>
<tr>
<td>$7 \times 10^5$</td>
<td>23.78</td>
<td>33.79</td>
<td>29.88</td>
<td>46.98</td>
<td>70.28</td>
</tr>
<tr>
<td>$8 \times 10^5$</td>
<td>24.82</td>
<td>36.88</td>
<td>33.09</td>
<td>47.79</td>
<td>—</td>
</tr>
<tr>
<td>$9 \times 10^5$</td>
<td>25.32</td>
<td>37.54</td>
<td>34.51</td>
<td>55.39</td>
<td>—</td>
</tr>
</tbody>
</table>
Table 5. The statistics about the node type in the AMTs built for the small group

<table>
<thead>
<tr>
<th>Size</th>
<th>Map 1</th>
<th>Map 4</th>
<th>Map 16</th>
<th>Map 48</th>
<th>Map 256</th>
<th>Single String Array</th>
<th>String Array</th>
<th>Hash Table</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1×10^5</td>
<td>2,000</td>
<td>1,006</td>
<td>72</td>
<td>18</td>
<td>0</td>
<td>68,268</td>
<td>21,971</td>
<td>103</td>
<td>93,438</td>
</tr>
<tr>
<td>2×10^5</td>
<td>4,628</td>
<td>2,808</td>
<td>307</td>
<td>42</td>
<td>0</td>
<td>130,440</td>
<td>46,514</td>
<td>244</td>
<td>184,983</td>
</tr>
<tr>
<td>3×10^5</td>
<td>7,414</td>
<td>4,858</td>
<td>574</td>
<td>84</td>
<td>0</td>
<td>190,653</td>
<td>71,757</td>
<td>403</td>
<td>275,746</td>
</tr>
<tr>
<td>4×10^5</td>
<td>10,555</td>
<td>7,282</td>
<td>112</td>
<td>139</td>
<td>0</td>
<td>247,656</td>
<td>97,967</td>
<td>544</td>
<td>365,089</td>
</tr>
<tr>
<td>5×10^5</td>
<td>13,703</td>
<td>10,651</td>
<td>946</td>
<td>238</td>
<td>4</td>
<td>302,350</td>
<td>124,528</td>
<td>17</td>
<td>452,994</td>
</tr>
<tr>
<td>6×10^5</td>
<td>17,355</td>
<td>13,536</td>
<td>1,503</td>
<td>298</td>
<td>3</td>
<td>356,644</td>
<td>151,435</td>
<td>31</td>
<td>531,233</td>
</tr>
<tr>
<td>7×10^5</td>
<td>21,080</td>
<td>16,872</td>
<td>1,954</td>
<td>388</td>
<td>4</td>
<td>408,165</td>
<td>179,621</td>
<td>35</td>
<td>628,570</td>
</tr>
<tr>
<td>8×10^5</td>
<td>24,357</td>
<td>20,094</td>
<td>2,405</td>
<td>437</td>
<td>3</td>
<td>457,674</td>
<td>208,426</td>
<td>37</td>
<td>714,004</td>
</tr>
<tr>
<td>9×10^5</td>
<td>28,356</td>
<td>23,924</td>
<td>2,977</td>
<td>513</td>
<td>9</td>
<td>504,779</td>
<td>238,632</td>
<td>42</td>
<td>799,696</td>
</tr>
</tbody>
</table>

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: 22.5s (WM), 16.5s (Pre-filter+AC), 11.8s (MASM), 3.9s (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 pipelined binary search tree.

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

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

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 8×10^4 tree nodes.
Table 7. The statistics about the node type in the AMTs built for the large group

<table>
<thead>
<tr>
<th>Size</th>
<th>Map 1</th>
<th>Map 4</th>
<th>Map 16</th>
<th>Map 48</th>
<th>Map 256</th>
<th>Single String Array</th>
<th>String Array</th>
<th>Hash Table</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1×10^6</td>
<td>38,935</td>
<td>27,428</td>
<td>6,468</td>
<td>851</td>
<td>4</td>
<td>588,343</td>
<td>3,986,683</td>
<td>1,027</td>
<td>903,942</td>
</tr>
<tr>
<td>2×10^6</td>
<td>85,900</td>
<td>68,973</td>
<td>17,084</td>
<td>2,141</td>
<td>20</td>
<td>1,091,527</td>
<td>496,001</td>
<td>2,015</td>
<td>1,763,661</td>
</tr>
<tr>
<td>3×10^6</td>
<td>136,054</td>
<td>117,527</td>
<td>28,092</td>
<td>3,410</td>
<td>17</td>
<td>1,547,659</td>
<td>764,359</td>
<td>2,749</td>
<td>2,599,867</td>
</tr>
<tr>
<td>4×10^6</td>
<td>189,926</td>
<td>171,155</td>
<td>40,328</td>
<td>4,716</td>
<td>23</td>
<td>1,966,326</td>
<td>1,045,558</td>
<td>3,450</td>
<td>3,421,482</td>
</tr>
<tr>
<td>5×10^6</td>
<td>247,707</td>
<td>230,163</td>
<td>83,043</td>
<td>5,989</td>
<td>38</td>
<td>2,354,308</td>
<td>1,332,896</td>
<td>3,885</td>
<td>4,228,029</td>
</tr>
<tr>
<td>6×10^6</td>
<td>304,863</td>
<td>293,591</td>
<td>65,475</td>
<td>7,169</td>
<td>49</td>
<td>2,706,094</td>
<td>1,625,261</td>
<td>4,379</td>
<td>5,006,881</td>
</tr>
<tr>
<td>7×10^6</td>
<td>366,861</td>
<td>361,695</td>
<td>77,899</td>
<td>8,434</td>
<td>53</td>
<td>3,030,673</td>
<td>1,925,027</td>
<td>4,733</td>
<td>5,775,380</td>
</tr>
<tr>
<td>8×10^6</td>
<td>429,121</td>
<td>433,708</td>
<td>90,280</td>
<td>9,660</td>
<td>62</td>
<td>3,336,765</td>
<td>2,226,879</td>
<td>5,070</td>
<td>6,531,545</td>
</tr>
<tr>
<td>9×10^6</td>
<td>494,278</td>
<td>509,992</td>
<td>102,402</td>
<td>10,833</td>
<td>72</td>
<td>3,623,407</td>
<td>2,537,413</td>
<td>5,224</td>
<td>7,283,621</td>
</tr>
<tr>
<td>1×10^7</td>
<td>558,241</td>
<td>591,455</td>
<td>115,012</td>
<td>11,922</td>
<td>79</td>
<td>3,986,683</td>
<td>2,847,277</td>
<td>5,505</td>
<td>8,026,174</td>
</tr>
</tbody>
</table>

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 lps. 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

An Efficient Adaptive Architecture for Multi-pattern Matching


